Getting Started with the LLVM System

- Overview
- Getting Started Quickly (A Summary)
- Requirements
  - Hardware
  - Software
  - Host C++ Toolchain, both Compiler and Standard Library
    - Getting a Modern Host C++ Toolchain
- Getting Started with LLVM
  - Terminology and Notation
  - Unpacking the LLVM Archives
  - Checkout LLVM from Git
    - Sending patches
    - For developers to commit changes from Git
    - Reverting a change when using Git
    - Checkout via SVN (deprecated)
  - Local LLVM Configuration
  - Compiling the LLVM Suite Source Code
  - Cross-Compiling LLVM
  - The Location of LLVM Object Files
  - Optional Configuration Items
- Directory Layout
  - **`llvm/examples`**
  - **`llvm/include`**
  - **`llvm/lib`**
  - **`llvm/projects`**
  - **`llvm/test`**
  - **`test-suite`**
  - **`llvm/tools`**
  - **`llvm/utils`**
- An Example Using the LLVM Tool Chain
  - Example with clang
- Common Problems
- Links

Overview

Welcome to the LLVM project! In order to get started, you first need to know some basic information.

First, the LLVM project has multiple components. The core of the project is itself called “LLVM”. This contains all of the tools, libraries, and header files needed to process an intermediate representation and convert it into object files. It contains an assembler, disassembler, bitcode analyzer and bitcode optimizer. It also contains basic regression tests.

Another piece is the Clang front end. This component compiles C, C++, Objective C, and Objective C++ code into LLVM bitcode – and from there into object files, using LLVM.

There are other components as well: the libc++ C++ standard library, the LLD linker, and more.

Getting Started Quickly (A Summary)

The LLVM Getting Started documentation may be out of date. So, the Clang Getting Started page might also be a good place to start.

Here's the short story for getting up and running quickly with LLVM:

1. Read the documentation.
2. Read the documentation.
3. Remember that you were warned twice about reading the documentation.
4. Checkout LLVM (including related subprojects like Clang):
    - `git clone https://github.com/llvm/llvm-project.git`
    - Or, on windows, `git clone --config core.autocrlf=false https://github.com/llvm/llvm-project.git`
5. Configure and build LLVM and Clang:.
    - `cd llvm-project`
    - `mkdir build`
    - `cd build`
    - `cmake -G <generator> [options] ../llvm`

      Some common generators are:

        - `Ninja` — for generating Ninja build files. Most llvm developers use Ninja.
        - `Unix Makefiles` — for generating make-compatible parallel makefiles.
        - `Visual Studio` — for generating Visual Studio projects and solutions.
        - `Xcode` — for generating Xcode projects.

      Some Common options:

        - `-DLLVM_ENABLE_PROJECTS='...'` — semicolon-separated list of the LLVM subprojects you'd like to additionally build. Can include any of: clang, clang-tools-extra, libcxx, libcxxabi, libunwind, lldb, compiler-rt, lld, polly, or debuginfo-tests.

          For example, to build LLVM, Clang, libcxx, and libcxxabi, use `-DLLVM_ENABLE_PROJECTS="clang;libcxx;libcxxabi"`.

        - `-DCMAKE_INSTALL_PREFIX=directory` — Specify for *directory* the full pathname of where you want the LLVM tools and libraries to be installed (default `/usr/local`).
        - `-DCMAKE_BUILD_TYPE=type` — Valid options for *type* are Debug, Release, RelWithDebInfo, and MinSizeRel. Default is Debug.
        - `-DLLVM_ENABLE_ASSERTIONS=On` — Compile with assertion checks enabled (default is Yes for Debug builds, No for all other build types).
    - Run your build tool of choice!
        - The default target (i.e. `ninja` or `make`) will build all of LLVM.
        - The `check-all` target (i.e. `ninja check-all`) will run the regression tests to ensure everything is in working order.

- CMake will generate build targets for each tool and library, and most LLVM sub-projects generate their own `check-<project>` target.
- Running a serial build will be *slow*. Make sure you run a parallel build. That's already done by default in Ninja; for `make`, use `make -j NNN` (with an appropriate value of NNN, e.g. number of CPUs you have.)

- For more information see CMake
- If you get an "internal compiler error (ICE)" or test failures, see below.

Consult the Getting Started with LLVM section for detailed information on configuring and compiling LLVM. Go to Directory Layout to learn about the layout of the source code tree.

Requirements

Before you begin to use the LLVM system, review the requirements given below. This may save you some trouble by knowing ahead of time what hardware and software you will need.

Hardware

LLVM is known to work on the following host platforms:

OS	Arch	Compilers
Linux	x86[1]	GCC, Clang
Linux	amd64	GCC, Clang
Linux	ARM	GCC, Clang
Linux	PowerPC	GCC, Clang
Solaris	V9 (Ultrasparc)	GCC
FreeBSD	x86[1]	GCC, Clang
FreeBSD	amd64	GCC, Clang
NetBSD	x86[1]	GCC, Clang
NetBSD	amd64	GCC, Clang
macOS[2]	PowerPC	GCC
macOS	x86	GCC, Clang
Cygwin/Win32	x86[1, 3]	GCC
Windows	x86[1]	Visual Studio
Windows x64	x86-64	Visual Studio

Note

1. Code generation supported for Pentium processors and up
2. Code generation supported for 32-bit ABI only
3. To use LLVM modules on Win32-based system, you may configure LLVM with `-DBUILD_SHARED_LIBS=On`.

Note that Debug builds require a lot of time and disk space. An LLVM-only build will need about 1-3 GB of space. A full build of LLVM and Clang will need around 15-20 GB of disk space. The exact space requirements will vary by system. (It is so large because of all the debugging information and the fact that the libraries are statically linked into multiple tools).

If you are space-constrained, you can build only selected tools or only selected targets. The Release build requires considerably less space.

The LLVM suite *may* compile on other platforms, but it is not guaranteed to do so. If compilation is successful, the LLVM utilities should be able to assemble, disassemble, analyze, and optimize LLVM bitcode. Code generation should work as well, although the generated native code may not work on your platform.

Software

Compiling LLVM requires that you have several software packages installed. The table below lists those required packages. The Package column is the usual name for the software package that LLVM depends on. The Version column provides "known to work" versions of the package. The Notes column describes how LLVM uses the package and provides other details.

Package	Version	Notes
CMake	>=3.4.3	Makefile/workspace generator
GCC	>=5.1.0	C/C++ compiler[1]
python	>=2.7	Automated test suite[2]
zlib	>=1.2.3.4	Compression library[3]
GNU Make	3.79, 3.79.1	Makefile/build processor[4]

Note

1. Only the C and C++ languages are needed so there's no need to build the other languages for LLVM's purposes. See *below* for specific version info.
2. Only needed if you want to run the automated test suite in the `llvm/test` directory.
3. Optional, adds compression / uncompression capabilities to selected LLVM tools.
4. Optional, you can use any other build tool supported by CMake.

Additionally, your compilation host is expected to have the usual plethora of Unix utilities. Specifically:

- **ar** — archive library builder
- **bzip2** — bzip2 command for distribution generation
- **bunzip2** — bunzip2 command for distribution checking
- **chmod** — change permissions on a file
- **cat** — output concatenation utility
- **cp** — copy files
- **date** — print the current date/time
- **echo** — print to standard output
- **egrep** — extended regular expression search utility
- **find** — find files/dirs in a file system
- **grep** — regular expression search utility
- **gzip** — gzip command for distribution generation
- **gunzip** — gunzip command for distribution checking
- **install** — install directories/files
- **mkdir** — create a directory
- **mv** — move (rename) files
- **ranlib** — symbol table builder for archive libraries
- **rm** — remove (delete) files and directories
- **sed** — stream editor for transforming output
- **sh** — Bourne shell for make build scripts
- **tar** — tape archive for distribution generation
- **test** — test things in file system

- **unzip** — unzip command for distribution checking
- **zip** — zip command for distribution generation

Host C++ Toolchain, both Compiler and Standard Library

LLVM is very demanding of the host C++ compiler, and as such tends to expose bugs in the compiler. We also attempt to follow improvements and developments in the C++ language and library reasonably closely. As such, we require a modern host C++ toolchain, both compiler and standard library, in order to build LLVM.

LLVM is written using the subset of C++ documented in coding standards. To enforce this language version, we check the most popular host toolchains for specific minimum versions in our build systems:

- Clang 3.5
- Apple Clang 6.0
- GCC 5.1
- Visual Studio 2017

Anything older than these toolchains *may* work, but will require forcing the build system with a special option and is not really a supported host platform. Also note that older versions of these compilers have often crashed or miscompiled LLVM.

For less widely used host toolchains such as ICC or xlC, be aware that a very recent version may be required to support all of the C++ features used in LLVM.

We track certain versions of software that are *known* to fail when used as part of the host toolchain. These even include linkers at times.

GNU ld 2.16.X. Some 2.16.X versions of the ld linker will produce very long warning messages complaining that some "`.gnu.linkonce.t.*`" symbol was defined in a discarded section. You can safely ignore these messages as they are erroneous and the linkage is correct. These messages disappear using ld 2.17.

GNU binutils 2.17: Binutils 2.17 contains a bug which causes huge link times (minutes instead of seconds) when building LLVM. We recommend upgrading to a newer version (2.17.50.0.4 or later).

GNU Binutils 2.19.1 Gold: This version of Gold contained a bug which causes intermittent failures when building LLVM with position independent code. The symptom is an error about cyclic dependencies. We recommend upgrading to a newer version of Gold.

Getting a Modern Host C++ Toolchain

This section mostly applies to Linux and older BSDs. On macOS, you should have a sufficiently modern Xcode, or you will likely need to upgrade until you do. Windows does not have a "system compiler", so you must install either Visual Studio 2017 or a recent version of mingw64. FreeBSD 10.0 and newer have a modern Clang as the system compiler.

However, some Linux distributions and some other or older BSDs sometimes have extremely old versions of GCC. These steps attempt to help you upgrade you compiler even on such a system. However, if at all possible, we encourage you to use a recent version of a distribution with a modern system compiler that meets these requirements. Note that it is tempting to install a prior version of Clang and libc++ to be the host compiler, however libc++ was not well tested or set up to build on Linux until relatively recently. As a consequence, this guide suggests just using libstdc++ and a modern GCC as the initial host in a bootstrap, and then using Clang (and potentially libc++).

The first step is to get a recent GCC toolchain installed. The most common distribution on which users have struggled with the version requirements is Ubuntu Precise, 12.04 LTS. For this distribution, one easy option is to install the toolchain testing PPA and use it to install a modern GCC. There is a really nice discussions of this on the ask ubuntu stack exchange and a github gist with updated commands. However, not all users can use PPAs and there are many other distributions, so it may be necessary (or just useful, if you're here you *are* doing compiler development after all) to build and install GCC from source. It is also quite easy to do these days.

Easy steps for installing GCC 5.1.0:

```
% gcc_version=5.1.0
% wget https://ftp.gnu.org/gnu/gcc/gcc-${gcc_version}/gcc-${gcc_version}.tar.bz2
% wget https://ftp.gnu.org/gnu/gcc/gcc-${gcc_version}/gcc-${gcc_version}.tar.bz2.sig
% wget https://ftp.gnu.org/gnu/gnu-keyring.gpg
% signature_invalid=`gpg --verify --no-default-keyring --keyring ./gnu-keyring.gpg gcc-
% if [ $signature_invalid ]; then echo "Invalid signature" ; exit 1 ; fi
% tar -xvjf gcc-${gcc_version}.tar.bz2
% cd gcc-${gcc_version}
% ./contrib/download_prerequisites
% cd ..
% mkdir gcc-${gcc_version}-build
% cd gcc-${gcc_version}-build
% $PWD/../gcc-${gcc_version}/configure --prefix=$HOME/toolchains --enable-languages=c,c
% make -j$(nproc)
% make install
```

For more details, check out the excellent GCC wiki entry, where I got most of this information from.

Once you have a GCC toolchain, configure your build of LLVM to use the new toolchain for your host compiler and C++ standard library. Because the new version of libstdc++ is not on the system library search path, you need to pass extra linker flags so that it can be found at link time (`-L`) and at runtime (`-rpath`). If you are using CMake, this invocation should produce working binaries:

```
% mkdir build
% cd build
% CC=$HOME/toolchains/bin/gcc CXX=$HOME/toolchains/bin/g++ \
  cmake .. -DCMAKE_CXX_LINK_FLAGS="-Wl,-rpath,$HOME/toolchains/lib64 -L$HOME/toolchains
```

If you fail to set rpath, most LLVM binaries will fail on startup with a message from the loader similar to `libstdc++.so.6: version `GLIBCXX_3.4.20' not found`. This means you need to tweak the -rpath linker flag.

When you build Clang, you will need to give *it* access to modern C++ standard library in order to use it as your new host in part of a bootstrap. There are two easy ways to do this, either build (and install) libc++ along with Clang and then use it with the `-stdlib=libc++` compile and link flag, or install Clang into the same prefix (`$HOME/toolchains` above) as GCC. Clang will look within its own prefix for libstdc++ and use it if found. You can also add an explicit prefix for Clang to look in for a GCC toolchain with the `--gcc-toolchain=/opt/my/gcc/prefix` flag, passing it to both compile and link commands when using your just-built-Clang to bootstrap.

Getting Started with LLVM

The remainder of this guide is meant to get you up and running with LLVM and to give you some basic information about the LLVM environment.

The later sections of this guide describe the general layout of the LLVM source tree, a simple example using the LLVM tool chain, and links to find more information about LLVM or to get help via e-mail.

Terminology and Notation

Throughout this manual, the following names are used to denote paths specific to the local system and working environment. *These are not environment variables you need to set but just strings used in the rest of this document below*. In any of the examples below, simply replace each of these names with the appropriate pathname on your local system. All these paths are absolute:

`SRC_ROOT`

This is the top level directory of the LLVM source tree.

`OBJ_ROOT`

This is the top level directory of the LLVM object tree (i.e. the tree where object files and compiled programs will be placed. It can be the same as SRC_ROOT).

Unpacking the LLVM Archives

If you have the LLVM distribution, you will need to unpack it before you can begin to compile it. LLVM is distributed as a number of different subprojects. Each one has its own download which is a TAR archive that is compressed with the gzip program.

The files are as follows, with *x.y* marking the version number:

`llvm-x.y.tar.gz`

Source release for the LLVM libraries and tools.

`cfe-x.y.tar.gz`

Source release for the Clang frontend.

Checkout LLVM from Git

You can also checkout the source code for LLVM from Git. While the LLVM project's official source-code repository is Subversion, we are in the process of migrating to git. We currently recommend that all developers use Git for day-to-day development.

Note

Passing `--config core.autocrlf=false` should not be required in the future after we adjust the .gitattribute settings correctly, but is required for Windows users at the time of this writing.

Simply run:

```
% git clone https://github.com/llvm/llvm-project.git
```

or on Windows,

```
% git clone --config core.autocrlf=false https://github.com/llvm/llvm-project.git
```

This will create an '`llvm-project`' directory in the current directory and fully populate it with all of the source code, test directories, and local copies of documentation files for LLVM and all the related subprojects. Note that unlike the tarballs, which contain each subproject in a separate file, the git repository contains all of the projects together.

If you want to get a specific release (as opposed to the most recent revision), you can check out a tag after cloning the repository. E.g., *git checkout llvmorg-6.0.1* inside the `llvm-project` directory created by the above command. Use *git tag -l* to list all of them.

Sending patches

Please read Developer Policy, too.

We don't currently accept github pull requests, so you'll need to send patches either via emailing to llvm-commits, or, preferably, via Phabricator.

You'll generally want to make sure your branch has a single commit, corresponding to the review you wish to send, up-to-date with the upstream `origin/master` branch, and doesn't contain merges. Once you have that, you can use `git show` or `git format-patch` to output the diff, and attach it to a Phabricator review (or to an email message).

However, using the "Arcanist" tool is often easier. After installing arcanist, you can upload the latest commit using:

```
% arc diff HEAD~1
```

Additionally, before sending a patch for review, please also try to ensure it's formatted properly. We use `clang-format` for this, which has git integration through the `git-clang-format` script. On some systems, it may already be installed (or be installable via your package manager). If so, you can simply run it – the following command will format only the code changed in the most recent commit:

```
% git clang-format HEAD~1
```

Note that this modifies the files, but doesn't commit them – you'll likely want to run

```
% git commit --amend -a
```

in order to update the last commit with all pending changes.

Note

If you don't already have `clang-format` or `git clang-format` installed on your system, the `clang-format` binary will be built alongside clang, and the git integration can be run from `clang/tools/clang-format/git-clang-format`.

For developers to commit changes from Git

A helper script is provided in `llvm/utils/git-svn/git-llvm`. After you add it to your path, you can push committed changes upstream with `git llvm push`. While this creates a Subversion checkout and patches it under the hood, it does not require you to have interaction with it.

```
% export PATH=$PATH:$TOP_LEVEL_DIR/llvm-project/llvm/utils/git-svn/
% git llvm push
```

Within a couple minutes after pushing to subversion, the svn commit will have been converted back to a Git commit, and made its way into the official Git repository. At that point, `git pull` should get back the changes as they were committed.

You'll likely want to `git pull --rebase` to get the official git commit downloaded back to your repository. The SVN revision numbers of each commit can be found at the end of the commit message, e.g.

`llvm-svn: 350914.`

You may also find the `-n` flag useful, like `git llvm push -n`. This runs through all the steps of committing _without_ actually doing the commit, and tell you what it would have done. That can be useful if you're unsure whether the right thing will happen.

Reverting a change when using Git

If you're using Git and need to revert a patch, Git needs to be supplied a commit hash, not an svn revision. To make things easier, you can use `git llvm revert` to revert with either an SVN revision or a Git hash instead.

Additionally, you can first run with `git llvm revert -n` to print which Git commands will run, without doing anything.

Running `git llvm revert` will only revert things in your local repository. To push the revert upstream, you still need to run `git llvm push` as described earlier.

```
% git llvm revert rNNNNNN       # Revert by SVN id
% git llvm revert abcdef123456  # Revert by Git commit hash
% git llvm revert -n rNNNNNN    # Print the commands without doing anything
```

Checkout via SVN (deprecated)

Until we have fully migrated to Git, you may also get a fresh copy of the code from the official Subversion repository.

- `cd where-you-want-llvm-to-live`
- Read-Only: `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
- Read-Write: `svn co https://user@llvm.org/svn/llvm-project/llvm/trunk llvm`

This will create an '`llvm`' directory in the current directory and fully populate it with the LLVM source code, Makefiles, test directories, and local copies of documentation files.

If you want to get a specific release (as opposed to the most recent revision), you can check it out from the '`tags`' directory (instead of '`trunk`'). The following releases are located in the following subdirectories of the '`tags`' directory:

- Release 3.5.0 and later: **RELEASE_350/final** and so on
- Release 2.9 through 3.4: **RELEASE_29/final** and so on
- Release 1.1 through 2.8: **RELEASE_11** and so on
- Release 1.0: **RELEASE_1**

Local LLVM Configuration

Once checked out repository, the LLVM suite source code must be configured before being built. This process uses CMake. Unlinke the normal `configure` script, CMake generates the build files in whatever format you request as well as various `*.inc` files, and `llvm/include/Config/config.h`.

Variables are passed to `cmake` on the command line using the format `-D<variable name>=<value>`. The following variables are some common options used by people developing LLVM.

Variable	Purpose
CMAKE_C_COMPILER	Tells `cmake` which C compiler to use. By default, this will be /usr/bin/cc.

Variable	Purpose
CMAKE_CXX_COMPILER	Tells `cmake` which C++ compiler to use. By default, this will be /usr/bin/c++.
CMAKE_BUILD_TYPE	Tells `cmake` what type of build you are trying to generate files for. Valid options are Debug, Release, RelWithDebInfo, and MinSizeRel. Default is Debug.
CMAKE_INSTALL_PREFIX	Specifies the install directory to target when running the install action of the build files.
PYTHON_EXECUTABLE	Forces CMake to use a specific Python version by passing a path to a Python interpreter. By default the Python version of the interpreter in your PATH is used.
LLVM_TARGETS_TO_BUILD	A semicolon delimited list controlling which targets will be built and linked into llvm. The default list is defined as `LLVM_ALL_TARGETS`, and can be set to include out-of-tree targets. The default value includes: `AArch64, AMDGPU, ARM, BPF, Hexagon, Mips, MSP430, NVPTX, PowerPC, Sparc, SystemZ, X86, XCore`.
LLVM_ENABLE_DOXYGEN	Build doxygen-based documentation from the source code This is disabled by default because it is slow and generates a lot of output.
LLVM_ENABLE_PROJECTS	A semicolon-delimited list selecting which of the other LLVM subprojects to additionally build. (Only effective when using a side-by-side project layout e.g. via git). The default list is empty. Can include: clang, libcxx, libcxxabi, libunwind, lldb, compiler-rt, lld, polly, or debuginfo-tests.
LLVM_ENABLE_SPHINX	Build sphinx-based documentation from the source code. This is disabled by default because it is slow and generates a lot of output. Sphinx version 1.5 or later recommended.
LLVM_BUILD_LLVM_DYLIB	Generate libLLVM.so. This library contains a default set of LLVM components that can be overridden with `LLVM_DYLIB_COMPONENTS`. The default contains most of LLVM and is defined in `tools/llvm-shlib/CMakelists.txt`.
LLVM_OPTIMIZED_TABLEGEN	Builds a release tablegen that gets used during the LLVM build. This can dramatically speed up debug builds.

To configure LLVM, follow these steps:

1. Change directory into the object root directory:

```
% cd OBJ_ROOT
```

2. Run the `cmake`:

```
% cmake -G "Unix Makefiles" -DCMAKE_INSTALL_PREFIX=/install/path
  [other options] SRC_ROOT
```

Compiling the LLVM Suite Source Code

Unlike with autotools, with CMake your build type is defined at configuration. If you want to change your build type, you can re-run cmake with the following invocation:

```
% cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=type SRC_ROOT
```

Between runs, CMake preserves the values set for all options. CMake has the following build types defined:

Debug

These builds are the default. The build system will compile the tools and libraries unoptimized, with debugging information, and asserts enabled.

Release

For these builds, the build system will compile the tools and libraries with optimizations enabled and not generate debug info. CMakes default optimization level is -O3. This can be configured by setting the `CMAKE_CXX_FLAGS_RELEASE` variable on the CMake command line.

RelWithDebInfo

These builds are useful when debugging. They generate optimized binaries with debug information. CMakes default optimization level is -O2. This can be configured by setting the `CMAKE_CXX_FLAGS_RELWITHDEBINFO` variable on the CMake command line.

Once you have LLVM configured, you can build it by entering the *OBJ_ROOT* directory and issuing the following command:

```
% make
```

If the build fails, please check here to see if you are using a version of GCC that is known not to compile LLVM.

If you have multiple processors in your machine, you may wish to use some of the parallel build options provided by GNU Make. For example, you could use the command:

```
% make -j2
```

There are several special targets which are useful when working with the LLVM source code:

`make clean`

Removes all files generated by the build. This includes object files, generated C/C++ files, libraries, and executables.

`make install`

Installs LLVM header files, libraries, tools, and documentation in a hierarchy under `$PREFIX`, specified with `CMAKE_INSTALL_PREFIX`, which defaults to `/usr/local`.

`make docs-llvm-html`

If configured with `-DLLVM_ENABLE_SPHINX=On`, this will generate a directory at `OBJ_ROOT/docs/html` which contains the HTML formatted documentation.

Cross-Compiling LLVM

It is possible to cross-compile LLVM itself. That is, you can create LLVM executables and libraries to be hosted on a platform different from the platform where they are built (a Canadian Cross build). To generate build files for cross-compiling CMake provides a variable `CMAKE_TOOLCHAIN_FILE` which can define compiler flags and variables used during the CMake test operations.

The result of such a build is executables that are not runnable on the build host but can be executed on the target. As an example the following CMake invocation can generate build files targeting iOS. This will work on macOS with the latest Xcode:

```
% cmake -G "Ninja" -DCMAKE_OSX_ARCHITECTURES="armv7;armv7s;arm64"
  -DCMAKE_TOOLCHAIN_FILE=<PATH_TO_LLVM>/cmake/platforms/iOS.cmake
  -DCMAKE_BUILD_TYPE=Release -DLLVM_BUILD_RUNTIME=Off -DLLVM_INCLUDE_TESTS=Off
  -DLLVM_INCLUDE_EXAMPLES=Off -DLLVM_ENABLE_BACKTRACES=Off [options]
  <PATH_TO_LLVM>
```

Note: There are some additional flags that need to be passed when building for iOS due to limitations in the iOS SDK.

Check How To Cross-Compile Clang/LLVM using Clang/LLVM and Clang docs on how to cross-compile in general for more information about cross-compiling.

The Location of LLVM Object Files

The LLVM build system is capable of sharing a single LLVM source tree among several LLVM builds. Hence, it is possible to build LLVM for several different platforms or configurations using the same source tree.

- Change directory to where the LLVM object files should live:

```
% cd OBJ_ROOT
```

- Run `cmake`:

```
% cmake -G "Unix Makefiles" SRC_ROOT
```

The LLVM build will create a structure underneath *OBJ_ROOT* that matches the LLVM source tree. At each level where source files are present in the source tree there will be a corresponding `CMakeFiles` directory in the *OBJ_ROOT*. Underneath that directory there is another directory with a name ending in `.dir` under which you'll find object files for each source.

For example:

```
% cd llvm_build_dir
% find lib/Support/ -name APFloat*
lib/Support/CMakeFiles/LLVMSupport.dir/APFloat.cpp.o
```

Optional Configuration Items

If you're running on a Linux system that supports the binfmt_misc module, and you have root access on the system, you can set your system up to execute LLVM bitcode files directly. To do this, use commands like this (the first command may not be required if you are already using the module):

```
% mount -t binfmt_misc none /proc/sys/fs/binfmt_misc
% echo ':llvm:M::BC::/path/to/lli:' > /proc/sys/fs/binfmt_misc/register
% chmod u+x hello.bc   (if needed)
% ./hello.bc
```

This allows you to execute LLVM bitcode files directly. On Debian, you can also use this command instead of the 'echo' command above:

```
% sudo update-binfmts --install llvm /path/to/lli --magic 'BC'
```

Directory Layout

One useful source of information about the LLVM source base is the LLVM doxygen documentation available at http://llvm.org/doxygen/. The following is a brief introduction to code layout:

`llvm/examples`

Simple examples using the LLVM IR and JIT.

`llvm/include`

Public header files exported from the LLVM library. The three main subdirectories:

`llvm/include/llvm`

All LLVM-specific header files, and subdirectories for different portions of LLVM: `Analysis`, `CodeGen`, `Target`, `Transforms`, etc...

`llvm/include/llvm/Support`

Generic support libraries provided with LLVM but not necessarily specific to LLVM. For example, some C++ STL utilities and a Command Line option processing library store header files here.

`llvm/include/llvm/Config`

Header files configured by `cmake`. They wrap "standard" UNIX and C header files. Source code can include these header files which automatically take care of the conditional #includes that `cmake` generates.

`llvm/lib`

Most source files are here. By putting code in libraries, LLVM makes it easy to share code among the tools.

`llvm/lib/IR/`

Core LLVM source files that implement core classes like Instruction and BasicBlock.

`llvm/lib/AsmParser/`

Source code for the LLVM assembly language parser library.

`llvm/lib/Bitcode/`

Code for reading and writing bitcode.

`llvm/lib/Analysis/`

A variety of program analyses, such as Call Graphs, Induction Variables, Natural Loop Identification, etc.

`llvm/lib/Transforms/`

IR-to-IR program transformations, such as Aggressive Dead Code Elimination, Sparse Conditional Constant Propagation, Inlining, Loop Invariant Code Motion, Dead Global Elimination, and many others.

`llvm/lib/Target/`

Files describing target architectures for code generation. For example, `llvm/lib/Target/X86` holds the X86 machine description.

`llvm/lib/CodeGen/`

The major parts of the code generator: Instruction Selector, Instruction Scheduling, and Register Allocation.

`llvm/lib/MC/`

(FIXME: T.B.D.) ….?

`llvm/lib/ExecutionEngine/`

Libraries for directly executing bitcode at runtime in interpreted and JIT-compiled scenarios.

`llvm/lib/Support/`

Source code that corresponding to the header files in `llvm/include/ADT/` and `llvm/include/Support/`.

llvm/projects

Projects not strictly part of LLVM but shipped with LLVM. This is also the directory for creating your own LLVM-based projects which leverage the LLVM build system.

llvm/test

Feature and regression tests and other sanity checks on LLVM infrastructure. These are intended to run quickly and cover a lot of territory without being exhaustive.

test-suite

A comprehensive correctness, performance, and benchmarking test suite for LLVM. This comes in a `separate git repository <https://github.com/llvm/llvm-test-suite>`, because it contains a large amount of third-party code under a variety of licenses. For details see the Testing Guide document.

llvm/tools

Executables built out of the libraries above, which form the main part of the user interface. You can always get help for a tool by typing `tool_name -help`. The following is a brief introduction to the most important tools. More detailed information is in the Command Guide.

`bugpoint`

`bugpoint` is used to debug optimization passes or code generation backends by narrowing down the given test case to the minimum number of passes and/or instructions that still cause a problem, whether it is a crash or miscompilation. See HowToSubmitABug.html for more information on using `bugpoint`.

`llvm-ar`

The archiver produces an archive containing the given LLVM bitcode files, optionally with an index for faster lookup.

`llvm-as`

The assembler transforms the human readable LLVM assembly to LLVM bitcode.

`llvm-dis`

The disassembler transforms the LLVM bitcode to human readable LLVM assembly.

`llvm-link`

`llvm-link`, not surprisingly, links multiple LLVM modules into a single program.

`lli`

`lli` is the LLVM interpreter, which can directly execute LLVM bitcode (although very slowly...). For architectures that support it (currently x86, Sparc, and PowerPC), by default, `lli` will function as a Just-In-Time compiler (if the functionality was compiled in), and will execute the code *much* faster than the interpreter.

`llc`

`llc` is the LLVM backend compiler, which translates LLVM bitcode to a native code assembly file.

`opt`

`opt` reads LLVM bitcode, applies a series of LLVM to LLVM transformations (which are specified on the command line), and outputs the resultant bitcode. '`opt -help`' is a good way to get a list of the program transformations available in LLVM.

`opt` can also run a specific analysis on an input LLVM bitcode file and print the results. Primarily useful for debugging analyses, or familiarizing yourself with what an analysis does.

`llvm/utils`

Utilities for working with LLVM source code; some are part of the build process because they are code generators for parts of the infrastructure.

`codegen-diff`

`codegen-diff` finds differences between code that LLC generates and code that LLI generates. This is useful if you are debugging one of them, assuming that the other generates correct output. For the full user manual, run `` `perldoc codegen-diff' ``.

`emacs/`

Emacs and XEmacs syntax highlighting for LLVM assembly files and TableGen description files. See the `README` for information on using them.

`getsrcs.sh`

Finds and outputs all non-generated source files, useful if one wishes to do a lot of development across directories and does not want to find each file. One way to use it is to run, for example: `xemacs` `` `utils/getsources.sh` `` from the top of the LLVM source tree.

`llvmgrep`

Performs an `egrep -H -n` on each source file in LLVM and passes to it a regular expression provided on `llvmgrep`'s command line. This is an efficient way of searching the source base for a particular regular expression.

`TableGen/`

Contains the tool used to generate register descriptions, instruction set descriptions, and even assemblers from common TableGen description files.

`vim/`

vim syntax-highlighting for LLVM assembly files and TableGen description files. See the `README` for how to use them.

An Example Using the LLVM Tool Chain

This section gives an example of using LLVM with the Clang front end.

Example with clang

1. First, create a simple C file, name it 'hello.c':

```
#include <stdio.h>

int main() {
  printf("hello world\n");
  return 0;
}
```

2. Next, compile the C file into a native executable:

```
% clang hello.c -o hello
```

> **Note**
>
> Clang works just like GCC by default. The standard -S and -c arguments work as usual (producing a native .s or .o file, respectively).

3. Next, compile the C file into an LLVM bitcode file:

```
% clang -O3 -emit-llvm hello.c -c -o hello.bc
```

The -emit-llvm option can be used with the -S or -c options to emit an LLVM `.ll` or `.bc` file (respectively) for the code. This allows you to use the standard LLVM tools on the bitcode file.

4. Run the program in both forms. To run the program, use:

```
% ./hello
```

and

```
% lli hello.bc
```

The second examples shows how to invoke the LLVM JIT, lli.

5. Use the `llvm-dis` utility to take a look at the LLVM assembly code:

```
% llvm-dis < hello.bc | less
```

6. Compile the program to native assembly using the LLC code generator:

```
% llc hello.bc -o hello.s
```

7. Assemble the native assembly language file into a program:

```
% /opt/SUNWspro/bin/cc -xarch=v9 hello.s -o hello.native   # On Solaris

% gcc hello.s -o hello.native                              # On others
```

8. Execute the native code program:

```
% ./hello.native
```

Note that using clang to compile directly to native code (i.e. when the `-emit-llvm` option is not present) does steps 6/7/8 for you.

Common Problems

If you are having problems building or using LLVM, or if you have any other general questions about LLVM, please consult the Frequently Asked Questions page.

Links

This document is just an **introduction** on how to use LLVM to do some simple things... there are many more interesting and complicated things that you can do that aren't documented here (but we'll gladly accept a patch if you want to write something up!). For more information about LLVM, check out:

- LLVM Homepage
- LLVM Doxygen Tree
- Starting a Project that Uses LLVM