Three Ways to Use Android NDK Cross Compiler

22 Dec 2015

Today I was asked to help compile an Android executable from C source file on Mac OS X platform. Of course it is a cross compile task. Since I have Android NDK installed on my Mac, I thought it would be nice if I can make good use of that. I have used Android NDK to compile shared libraries for Android, so I would like to summary the possible ways to compile an Android executable using Android NDK.

Note: While writing this article, I found the official guide from Andriod was very nice and easy to read. Some of the examples are borrowed from there for the summary purpose.

1. Use the `ndk-build` and an `Android.mk` with `BUILD_EXECUTABLE`

The `Android.mk` and the `Application.mk` files are really tiny GNU makefile fragments that the build system parses once or more. The `Android.mk` file is useful for defining project-wide settings that

`Application.mk`, the build system, and your environment variables leave undefined. It can also override project-wide settings for specific modules.

The `Android.mk` must resides in a subdirectory of your project's `$PROJECT/jni/` directory, and describes your sources and shared libraries to the build system. The `Application.mk` also usually resides under `$PROJECT/jni/`. We can also place it under a sub-directory of the top-level `$NDK/apps/` directory.

Example of using `ndk-build`, an `Android.mk` and an `Application.mk` with `BUILD_EXECUTABLE`:

```
# Filename: Android.mk
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := foo
LOCAL_SRC_FILES := foo.c

include $(BUILD_EXECUTABLE)
```

```
# Filename: Application.mk
APP_ABI: armeabi areabi-v7a
```

Then in the terminal, jut change directory to the `$PROJECT/jni` folder and invoke `ndk-build`:

```
$ ndk-build
[armeabi] Compile thumb  : foo <=
foo.c
```

```
[armeabi] Executable     : foo
[armeabi] Install        : foo =>
libs/armeabi/foo
[armeabi-v7a] Compile thumb  : fo
o <= foo.c
[armeabi-v7a] Executable     : fo
o
[armeabi-v7a] Install        : fo
o => libs/armeabi-v7a/foo
```

We are all set and the compiled executable is installed under `$PROJECT/libs/<APP_ABI>` now.

2. Use Standalone Toolchain

You can use the toolchains provided with the Android NDK independently.

Steps:

1. Selecting Your Toolchain.

   Before anything else, you need to decide which processing architecture your standalone toolchain is going to target. Each architecture corresponds to a different toolchain name, as **Table 1** shows.

   Table 1. `APP_ABI` settings for different instruction sets.

Architecture	Value
ARM-based	`arm-linux-androideabi-<gcc-version>`
x86-based	`x86-<gcc-`

Architecture	Value
	`version>`
MIPS-based	`mipsel-linux-android-<gcc-version>`
ARM64-based	`aarch64-linux-android-<gcc-version>`
x86-64-based	`x86_64-<gcc-version>`
MIPS64-based	`mips64el-linux-android--<gcc-version>`

2. Selecting Your Sysroot

The next thing you need to do is define your sysroot (A sysroot is a directory containing the system headers and libraries for your target). To define the sysroot, you must must know the Android API level you want to target for native support; available native APIs vary by Android API level.

Native APIs for the respective Android API levels reside under `$NDK/platforms/` ; each API-level directory, in turn, contains subdirectories for the various CPUs and architectures. The following example shows how to define a sysroot for a build targeting Android 5.0 (API level 21), for ARM

architecture:

```
SYSROOT=$<ANDROID_NDK>/plat
forms/android-21/arch-arm
```

3. Invoking the Compiler

   The simplest way to build is by invoking the appropriate compiler directly from the command line, using the `--sysroot` option to indicate the location of the system files for the platform you're targeting. For example:

```
export CC="$<ANDROID_NDK>/toolcha
ins/arm-linux-androideabi-<gcc-ve
rsion>/prebuilt/ \
<your_machine_arch>/bin/arm-linux
-androideabi-gcc-<gcc-version> --
sysroot=$SYSROOT"
$CC -o foo.o -c foo.c
```

Caveat

for Clang, you need to perform an additional two steps:

- Add the appropriate -target for the target architecture, as Table 2 shows.
- Add assembler and linker support by adding the -gcc-toolchain option, as in the following example:

```
-gcc-toolchain $NDK/toolchains/ar
m-linux-androideabi-4.8/prebuilt/
```

```
linux-x86_64
```

Ultimately, a command to compile using Clang might look like this:

```
export CC="$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/ \
linux-x86/bin/arm-linux-androideabi-gcc-4.8 --sysroot=$SYSROOT" -target \
armv7-none-linux-androideabi \
-gcc-toolchain $NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/linux-x86_64"
$CC -o foo.o -c foo.c
```

3. Make a Ready-to-use Standalone Toolchain (advanced)

The NDK provides the `make-standalone-toolchain.sh` shell script to allow you to perform a customized toolchain installation from the command line.

The script is located in the `$NDK/build/tools/` directory, where `$NDK` is the installation root for the NDK. An example of the use of this script appears below:

```
$NDK/build/tools/make-standalone-toolchain.sh \
--arch=arm --platform=android-21 --install-dir=/tmp/my-android-toolchain
```

This command creates a directory named `/tmp/my-android-toolchain/` ,

containing a copy of the `android-21/arch-arm` sysroot, and of the toolchain binaries for a 32-bit ARM architecture.

Note that the toolchain binaries do not depend on or contain host-specific paths, in other words, you can install them in any location, or even move them if you need to.

By default, the build system uses the 32-bit, ARM-based GCC 4.8 toolchain. You can specify a different value, however, by specifying `--arch=<toolchain>` as an option. Alternatively, you can use the `--toolchain=<toolchain>` option. See Tables on Toolchain for more details.

You can make these settings directly, as in the following example:

```
export PATH=/tmp/my-android-toolc
hain/bin:$PATH
export CC=arm-linux-androideabi-g
cc   # or export CC=clang
export CXX=arm-linux-androideabi-
g++  # or export CXX=clang++
```

Note that if you omit the `-install-dir` option, the `make-standalone-toolchain.sh` shell script creates a tarball in `tmp/ndk/<toolchain-name>.tar.bz2` . This tarball makes it easy to archive, as well as to redistribute the binaries.

This standalone toolchain provides an

Nosce Te Ipsum

A blog about my Ph.D. life

Home
About
Archives
My Gallery
My Homepage

Currently v1.3

additional benefit, as well, in that it contains a working copy of a C++ STL library, with working exceptions and RTTI support. For more options and details, use `--help` or go to NDK Guides -> Standalone Toolchain.

.

Nosce Te Ipsum

A blog about my Ph.D. life

Home
About
Archives
My Gallery
My Homepage

Currently v1.3

Related Posts

My Memories of the Year 2017 in Photos **31 Dec 2017**

Cross-compile Wireshark for Android **18 Jul 2016**

Cross-compile GLib for Android **17 Jul 2016**

Nosce Te Ipsum

A blog about my Ph.D. life

Currently v1.3