android-cmake

CMake is great, and so is Android. This is a collection of CMake scripts that may be useful to the Android NDK community. It is based on experience from porting OpenCV library to Android: http://opencv.org/platforms/android.html

Main goal is to share these scripts so that devs that use CMake as their build system may easily compile native code for Android.

TL;DR

```
cmake -DCMAKE_TOOLCHAIN_FILE=android.toolchain.cmake \
      -DANDROID_NDK=<ndk_path>                       \
      -DCMAKE_BUILD_TYPE=Release                     \
      -DANDROID_ABI="armeabi-v7a with NEON"          \
      <source_path>
cmake --build .
```

One-liner:

```
cmake -DCMAKE_TOOLCHAIN_FILE=android.toolchain.cmake -DANDROID_NDK=<ndk_path> -
DCMAKE_BUILD_TYPE=Release -DANDROID_ABI="armeabi-v7a with NEON" <source_path> && cmake
--build .
```

android-cmake will search for your NDK install in the following order:

1. Value of `ANDROID_NDK` CMake variable;
2. Value of `ANDROID_NDK` environment variable;
3. Search under paths from `ANDROID_NDK_SEARCH_PATHS` CMake variable;
4. Search platform specific locations (home folder, Windows "Program Files", etc).

So if you have installed the NDK as `~/android-ndk-r10d` then *android-cmake* will locate it automatically.

Getting started

To build a cmake-based C/C++ project for Android you need:

- Android NDK (>= r5) http://developer.android.com/tools/sdk/ndk/index.html
- CMake (>= v2.6.3, >= v2.8.9 recommended) http://www.cmake.org/download

The *android-cmake* is also capable to build with NDK from AOSP or Linaro Android source tree, but you may be required to manually specify path to `libm` binary to link with.

Difference from traditional CMake

Folowing the *ndk-build* the *android-cmake* supports **only two build targets**:

- `-DCMAKE_BUILD_TYPE=Release`
- `-DCMAKE_BUILD_TYPE=Debug`

So don't even try other targets that can be found in CMake documentation and don't forget to explicitly specify `Release` or `Debug` because CMake builds without a build configuration by default.

Difference from *ndk-build*

- Latest GCC available in NDK is used as the default compiler;
- `Release` builds with `-O3` instead of `-Os`;
- `Release` builds without debug info (without `-g`) (because *ndk-build* always creates a stripped version but cmake delays this for `install/strip` target);
- `-fsigned-char` is added to compiler flags to make `char` signed by default as it is on x86/x86_64;
- GCC's stack protector is not used neither in `Debug` nor `Release` configurations;
- No builds for multiple platforms (e.g. building for both arm and x86 require to run cmake twice with different parameters);
- No file level Neon via `.neon` suffix;

The following features of *ndk-build* are not supported by the *android-cmake* yet:

- `armeabi-v7a-hard` ABI
- `libc++_static` / `libc++_shared` STL runtime

Basic options

Similarly to the NDK build system *android-cmake* allows to select between several compiler toolchains and target platforms. Most of the options can be set either as cmake arguments: `-D<NAME>=<VALUE>` or as environment variables:

- **ANDROID_NDK** - path to the Android NDK. If not set then *android-cmake* will search for the most recent version of supported NDK in commonly used locations;
- **ANDROID_ABI** - specifies the target Application Binary Interface (ABI). This option nearly matches to the APP_ABI variable used by ndk-build tool from Android NDK. If not specified then set to `armeabi-v7a`. Possible target names are:
  - `armeabi` - ARMv5TE based CPU with software floating point operations;
  - **`armeabi-v7a`** - ARMv7 based devices with hardware FPU instructions (VFPv3_D16);
  - `armeabi-v7a with NEON` - same as armeabi-v7a, but sets NEON as floating-point unit;
  - `armeabi-v7a with VFPV3` - same as armeabi-v7a, but sets VFPv3_D32 as floating-point unit;
  - `armeabi-v6 with VFP` - tuned for ARMv6 processors having VFP;
  - `x86` - IA-32 instruction set
  - `mips` - MIPS32 instruction set
  - `arm64-v8a` - ARMv8 AArch64 instruction set - only for NDK r10 and newer
  - `x86_64` - Intel64 instruction set (r1) - only for NDK r10 and newer
  - `mips64` - MIPS64 instruction set (r6) - only for NDK r10 and newer
- **ANDROID_NATIVE_API_LEVEL** - level of android API to build for. Can be set either to full name (example: `android-8`) or a numeric value (example: `17`). The default API level depends on the target ABI:
  - `android-8` for ARM;
  - `android-9` for x86 and MIPS;
  - `android-21` for 64-bit ABIs.

    Building for `android-L` is possible only when it is explicitly selected.

- **ANDROID_TOOLCHAIN_NAME** - the name of compiler toolchain to be used. This option allows to select between different GCC and Clang versions. The list of possible values depends on the NDK version and will be printed by toolchain file if an invalid value is set. By default *android-cmake* selects the most recent version of GCC which can build for specified `ANDROID_ABI`.

  Example values are:

  - `aarch64-linux-android-4.9`
  - `aarch64-linux-android-clang3.5`
  - `arm-linux-androideabi-4.8`
  - `arm-linux-androideabi-4.9`
  - `arm-linux-androideabi-clang3.5`
  - `mips64el-linux-android-4.9`
  - `mipsel-linux-android-4.8`
  - `x86-4.9`
  - `x86_64-4.9`
  - etc.
- **ANDROID_STL** - the name of C++ runtime to use. The default is `gnustl_static`.

  - `none` - do not configure the runtime.
  - `system` - use the default minimal system C++ runtime library.
    - Implies `-fno-rtti -fno-exceptions`.
  - `system_re` - use the default minimal system C++ runtime library.
    - Implies `-frtti -fexceptions`.
  - `gabi++_static` - use the GAbi++ runtime as a static library.
    - Implies `-frtti -fno-exceptions`.
    - Available for NDK r7 and newer.
  - `gabi++_shared` - use the GAbi++ runtime as a shared library.
    - Implies `-frtti -fno-exceptions`.
    - Available for NDK r7 and newer.
  - `stlport_static` - use the STLport runtime as a static library.
    - Implies `-fno-rtti -fno-exceptions` for NDK before r7.
    - Implies `-frtti -fno-exceptions` for NDK r7 and newer.
  - `stlport_shared` - use the STLport runtime as a shared library.
    - Implies `-fno-rtti -fno-exceptions` for NDK before r7.
    - Implies `-frtti -fno-exceptions` for NDK r7 and newer.
  - **`gnustl_static`** - use the GNU STL as a static library.
    - Implies `-frtti -fexceptions`.
  - `gnustl_shared` - use the GNU STL as a shared library.
    - Implies `-frtti -fno-exceptions`.
    - Available for NDK r7b and newer.
    - Silently degrades to `gnustl_static` if not available.
- **NDK_CCACHE** - path to `ccache` executable. If not set then initialized from `NDK_CCACHE` environment variable.

Advanced *android-cmake* options

Normally *android-cmake* users are not supposed to touch these variables but they might be useful to workaround some build issues:

- **ANDROID_FORCE_ARM_BUILD** = `OFF` - generate 32-bit ARM instructions instead of Thumb. Applicable only for arm ABIs and is forced to be `ON` for `armeabi-v6 with VFP`;
- **ANDROID_NO_UNDEFINED** = `ON` - show all undefined symbols as linker errors;
- **ANDROID_SO_UNDEFINED** = `OFF` - allow undefined symbols in shared libraries;
  - actually it is turned `ON` by default for NDK older than `r7`
- **ANDROID_STL_FORCE_FEATURES** = `ON` - automatically configure rtti and exceptions support based on C++ runtime;
- **ANDROID_NDK_LAYOUT** = `RELEASE` - inner layout of Android NDK, should be detected automatically. Possible values are:
  - `RELEASE` - public releases from Google;
  - `LINARO` - NDK from Linaro project;
  - `ANDROID` - NDK from AOSP.
- **ANDROID_FUNCTION_LEVEL_LINKING** = `ON` - enables saparate putting each function and data items into separate sections and enable garbage collection of unused input sections at link time (`-fdata-sections -ffunction-sections -Wl,--gc-sections`);
- **ANDROID_GOLD_LINKER** = `ON` - use gold linker with GCC 4.6 for NDK r8b and newer (only for ARM and x86);
- **ANDROID_NOEXECSTACK** = `ON` - enables or disables stack execution protection code (`-Wl,-z,noexecstack`);
- **ANDROID_RELRO** = `ON` - Enables RELRO - a memory corruption mitigation technique (`-Wl,-z,relro -Wl,-z,now`);
- **ANDROID_LIBM_PATH** - path to `libm.so` (set to something like `$(TOP)/out/target/product/<product_name>/obj/lib/libm.so`) to workaround unresolved `sincos`.

Fine-tuning `CMakeLists.txt` for *android-cmake*

Recognizing Android build

android-cmake defines `ANDROID` CMake variable which can be used to add Android-specific stuff:

```
if (ANDROID)
    message(STATUS "Hello from Android build!")
endif()
```

The recommended way to identify ARM/MIPS/x86 architecture is examining `CMAKE_SYSTEM_PROCESSOR` which is set to the appropriate value:

- `armv5te` - for `armeabi` ABI
- `armv6` - for `armeabi-v6 with VFP` ABI
- `armv7-a` - for `armeabi-v7a`, `armeabi-v7a with VFPV3` and `armeabi-v7a with NEON` ABIs
- `aarch64` - for `arm64-v8a` ABI
- `i686` - for `x86` ABI
- `x86_64` - for `x86_64` ABI

- `mips` - for `mips` ABI
- `mips64` - for `mips64` ABI

Other variables that are set by *android-cmake* and can be used for the fine-grained build configuration are:

- `NEON` - set if target ABI supports Neon;
- `ANDROID_NATIVE_API_LEVEL` - native Android API level we are building for (note: Java part of Andoid application can be built for another API level)
- `ANDROID_NDK_RELEASE` - version of the Android NDK
- `ANDROID_NDK_HOST_SYSTEM_NAME` - "windows", "linux-x86" or "darwin-x86" depending on the host platform
- `ANDROID_RTTI` - set if rtti is enabled by the runtime
- `ANDROID_EXCEPTIONS` - set if exceptions are enabled by the runtime

Finding packages

When crosscompiling CMake `find_*` commands are normally expected to find libraries and packages belonging to the same build target. So *android-cmake* configures CMake to search in Android-specific paths only and ignore your host system locations. So

```
find_package(ZLIB)
```

will surely find libz.so within the Android NDK.

However sometimes you need to locate a host package even when cross-compiling. For example you can be searching for your documentation generator. The *android-cmake* recommends you to use `find_host_package` and `find_host_program` macro defined in the `android.toolchain.cmake`:

```
find_host_package(Doxygen)
find_host_program(PDFLATEX pdflatex)
```

However this will break regular builds so instead of wrapping package search into platform-specific logic you can copy the following snippet into your project (put it after your top-level `project()` command):

```
# Search packages for host system instead of packages for target system
# in case of cross compilation these macro should be defined by toolchain file
if(NOT COMMAND find_host_package)
  macro(find_host_package)
    find_package(${ARGN})
  endmacro()
endif()
if(NOT COMMAND find_host_program)
  macro(find_host_program)
    find_program(${ARGN})
  endmacro()
endif()
```

Compiler flags recycling

Make sure to do the following in your scripts:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${my_cxx_flags}")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${my_cxx_flags}")
```

The flags will be prepopulated with critical flags, so don't loose them. Also be aware that *android-cmake* also sets configuration-specific compiler and linker flags.

Troubleshooting

Building on Windows

First of all `cygwin` builds are **NOT supported** and will not be supported by *android-cmake*. To build natively on Windows you need a port of make but I recommend http://martine.github.io/ninja/ instead.

To build with Ninja you need:

- Ensure you are using CMake newer than 2.8.9;
- Download the latest Ninja from https://github.com/martine/ninja/releases;
- Put the `ninja.exe` into your PATH (or add path to `ninja.exe` to your PATH environment variable);
- Pass `-GNinja` to `cmake` alongside with other arguments (or choose Ninja generator in `cmake-gui`).
- Enjoy the fast native multithreaded build :)

But if you still want to stick to old make then:

- Get a Windows port of GNU Make:
  - Android NDK r7 (and newer) already has `make.exe` on board;
  - `mingw-make` should work as fine;
  - Download some other port. For example, this one: http://gnuwin32.sourceforge.net/packages/make.htm.
- Add path to your `make.exe` to system PATH or always use full path;
- Pass `-G"MinGW Makefiles"` and `-DCMAKE_MAKE_PROGRAM="<full/path/to/>make.exe"`
  - It must be `MinGW Makefiles` and not `Unix Makefiles` even if your `make.exe` is not a MinGW's make.
- Run `make.exe` or `cmake --build .` for single-threaded build.

Projects with assembler files

The *android-cmake* should correctly handle projects with assembler sources (`*.s` or `*.S`). But if you still facing problems with assembler then try to upgrade your CMake to version newer than 2.8.5

Copying

android-cmake is distributed under the terms of BSD 3-Clause License