NAME

cmake-toolchains − CMake Toolchains Reference

INTRODUCTION

CMake uses a toolchain of utilities to compile, link libraries and create archives, and other tasks to drive the build. The toolchain utilities available are determined by the languages enabled. In normal builds, CMake automatically determines the toolchain for host builds based on system introspection and defaults. In cross−compiling scenarios, a toolchain file may be specified with information about compiler and utility paths.

LANGUAGES

Languages are enabled by the **project()** command. Language−specific built−in variables, such as **CMAKE_CXX_COMPILER**, **CMAKE_CXX_COMPILER_ID** etc are set by invoking the **project()** command. If no project command is in the top−level CMakeLists file, one will be implicitly generated. By default the enabled languages are **C** and **CXX**:

```
project(C_Only C)
```

A special value of **NONE** can also be used with the **project()** command to enable no languages:

```
project(MyProject NONE)
```

The **enable_language()** command can be used to enable languages after the **project()** command:

```
enable_language(CXX)
```

When a language is enabled, CMake finds a compiler for that language, and determines some information, such as the vendor and version of the compiler, the target architecture and bitwidth, the location of corresponding utilities etc.

The **ENABLED_LANGUAGES** global property contains the languages which are currently enabled.

VARIABLES AND PROPERTIES

Several variables relate to the language components of a toolchain which are enabled. **CMAKE_<LANG>_COMPILER** is the full path to the compiler used for **<LANG>**. **CMAKE_<LANG>_COMPILER_ID** is the identifier used by CMake for the compiler and **CMAKE_<LANG>_COMPILER_VERSION** is the version of the compiler.

The **CMAKE_<LANG>_FLAGS** variables and the configuration−specific equivalents contain flags that will be added to the compile command when compiling a file of a particular language.

As the linker is invoked by the compiler driver, CMake needs a way to determine which compiler to use to invoke the linker. This is calculated by the **LANGUAGE** of source files in the target, and in the case of static libraries, the language of the dependent libraries. The choice CMake makes may be overridden with the **LINKER_LANGUAGE** target property.

TOOLCHAIN FEATURES

CMake provides the **try_compile()** command and wrapper macros such as **CheckCXXSourceCompiles**, **CheckCXXSymbolExists** and **CheckIncludeFile** to test capability and availability of various toolchain features. These APIs test the toolchain in some way and cache the result so that the test does not have to be performed again the next time CMake runs.

Some toolchain features have built−in handling in CMake, and do not require compile−tests. For example, **POSITION_INDEPENDENT_CODE** allows specifying that a target should be built as position−independent code, if the compiler supports that feature. The **<LANG>_VISIBILITY_PRESET** and **VISIBILITY_INLINES_HIDDEN** target properties add flags for hidden visibility, if supported by the compiler.

CROSS COMPILING

If **cmake(1)** is invoked with the command line parameter **−−toolchain path/to/file** or **−DCMAKE_TOOLCHAIN_FILE=path/to/file**, the file will be loaded early to set values for the compilers. The **CMAKE_CROSSCOMPILING** variable is set to true when CMake is cross−compiling.

Note that using the **CMAKE_SOURCE_DIR** or **CMAKE_BINARY_DIR** variables inside a toolchain file is typically undesirable. The toolchain file is used in contexts where these variables have different values when used in different places (e.g. as part of a call to **try_compile()**). In most cases, where there is a need to evaluate paths inside a toolchain file, the more appropriate variable to use would be **CMAKE_CURRENT_LIST_DIR**, since it always has an unambiguous, predictable value.

Cross Compiling for Linux

A typical cross−compiling toolchain for Linux has content such as:

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_SYSROOT /home/devel/rasp-pi-rootfs)
set(CMAKE_STAGING_PREFIX /home/devel/stage)

set(tools /home/devel/gcc-4.7-linaro-rpi-gnueabihf)
set(CMAKE_C_COMPILER ${tools}/bin/arm-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER ${tools}/bin/arm-linux-gnueabihf-g++)

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

The **CMAKE_SYSTEM_NAME** is the CMake−identifier of the target platform to build for.

The **CMAKE_SYSTEM_PROCESSOR** is the CMake−identifier of the target architecture to build for.

The **CMAKE_SYSROOT** is optional, and may be specified if a sysroot is available.

The **CMAKE_STAGING_PREFIX** is also optional. It may be used to specify a path on the host to install to. The **CMAKE_INSTALL_PREFIX** is always the runtime installation location, even when cross−compiling.

The **CMAKE_<LANG>_COMPILER** variables may be set to full paths, or to names of compilers to search for in standard locations. For toolchains that do not support linking binaries without custom flags or scripts one may set the **CMAKE_TRY_COMPILE_TARGET_TYPE** variable to **STATIC_LIBRARY** to tell CMake not to try to link executables during its checks.

CMake **find_*** commands will look in the sysroot, and the **CMAKE_FIND_ROOT_PATH** entries by default in all cases, as well as looking in the host system root prefix. Although this can be controlled on a case−by−case basis, when cross−compiling, it can be useful to exclude looking in either the host or the target for particular artifacts. Generally, includes, libraries and packages should be found in the target system prefixes, whereas executables which must be run as part of the build should be found only on the host and not on the target. This is the purpose of the **CMAKE_FIND_ROOT_PATH_MODE_*** variables.

Cross Compiling for the Cray Linux Environment

Cross compiling for compute nodes in the Cray Linux Environment can be done without needing a separate toolchain file. Specifying **−DCMAKE_SYSTEM_NAME=CrayLinuxEnvironment** on the CMake command line will ensure that the appropriate build settings and search paths are configured. The platform will pull its configuration from the current environment variables and will configure a project to use the

compiler wrappers from the Cray Programming Environment's **PrgEnv−*** modules if present and loaded.

The default configuration of the Cray Programming Environment is to only support static libraries. This can be overridden and shared libraries enabled by setting the **CRAYPE_LINK_TYPE** environment variable to **dynamic**.

Running CMake without specifying **CMAKE_SYSTEM_NAME** will run the configure step in host mode assuming a standard Linux environment. If not overridden, the **PrgEnv−*** compiler wrappers will end up getting used, which if targeting the either the login node or compute node, is likely not the desired behavior. The exception to this would be if you are building directly on a NID instead of cross−compiling from a login node. If trying to build software for a login node, you will need to either first unload the currently loaded **PrgEnv−*** module or explicitly tell CMake to use the system compilers in **/usr/bin** instead of the Cray wrappers. If instead targeting a compute node is desired, just specify the **CMAKE_SYSTEM_NAME** as mentioned above.

Cross Compiling using Clang

Some compilers such as Clang are inherently cross compilers. The **CMAKE_<LANG>_COMPILER_TARGET** can be set to pass a value to those supported compilers when compiling:

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(triple arm-linux-gnueabihf)

set(CMAKE_C_COMPILER clang)
set(CMAKE_C_COMPILER_TARGET ${triple})
set(CMAKE_CXX_COMPILER clang++)
set(CMAKE_CXX_COMPILER_TARGET ${triple})
```

Similarly, some compilers do not ship their own supplementary utilities such as linkers, but provide a way to specify the location of the external toolchain which will be used by the compiler driver. The **CMAKE_<LANG>_COMPILER_EXTERNAL_TOOLCHAIN** variable can be set in a toolchain file to pass the path to the compiler driver.

Cross Compiling for QNX

As the Clang compiler the QNX QCC compile is inherently a cross compiler. And the **CMAKE_<LANG>_COMPILER_TARGET** can be set to pass a value to those supported compilers when compiling:

```
set(CMAKE_SYSTEM_NAME QNX)

set(arch gcc_ntoarmv7le)

set(CMAKE_C_COMPILER qcc)
set(CMAKE_C_COMPILER_TARGET ${arch})
set(CMAKE_CXX_COMPILER QCC)
set(CMAKE_CXX_COMPILER_TARGET ${arch})

set(CMAKE_SYSROOT $ENV{QNX_TARGET})
```

Cross Compiling for Windows CE

Cross compiling for Windows CE requires the corresponding SDK being installed on your system. These SDKs are usually installed under **C:/Program Files (x86)/Windows CE Tools/SDKs**.

A toolchain file to configure a Visual Studio generator for Windows CE may look like this:

```
set(CMAKE_SYSTEM_NAME WindowsCE)

set(CMAKE_SYSTEM_VERSION 8.0)
set(CMAKE_SYSTEM_PROCESSOR arm)

set(CMAKE_GENERATOR_TOOLSET CE800) # Can be omitted for 8.0
set(CMAKE_GENERATOR_PLATFORM SDK_AM335X_SK_WEC2013_V310)
```

The **CMAKE_GENERATOR_PLATFORM** tells the generator which SDK to use. Further **CMAKE_SYSTEM_VERSION** tells the generator what version of Windows CE to use. Currently version 8.0 (Windows Embedded Compact 2013) is supported out of the box. Other versions may require one to set **CMAKE_GENERATOR_TOOLSET** to the correct value.

Cross Compiling for Windows 10 Universal Applications

A toolchain file to configure a Visual Studio generator for a Windows 10 Universal Application may look like this:

```
set(CMAKE_SYSTEM_NAME WindowsStore)
set(CMAKE_SYSTEM_VERSION 10.0)
```

A Windows 10 Universal Application targets both Windows Store and Windows Phone. Specify the **CMAKE_SYSTEM_VERSION** variable to be **10.0** to build with the latest available Windows 10 SDK. Specify a more specific version (e.g. **10.0.10240.0** for RTM) to build with the corresponding SDK.

Cross Compiling for Windows Phone

A toolchain file to configure a Visual Studio generator for Windows Phone may look like this:

```
set(CMAKE_SYSTEM_NAME WindowsPhone)
set(CMAKE_SYSTEM_VERSION 8.1)
```

Cross Compiling for Windows Store

A toolchain file to configure a Visual Studio generator for Windows Store may look like this:

```
set(CMAKE_SYSTEM_NAME WindowsStore)
set(CMAKE_SYSTEM_VERSION 8.1)
```

Cross Compiling for Android

A toolchain file may configure cross−compiling for Android by setting the **CMAKE_SYSTEM_NAME** variable to **Android**. Further configuration is specific to the Android development environment to be used.

For Visual Studio Generators, CMake expects *NVIDIA Nsight Tegra Visual Studio Edition* or the *Visual Studio tools for Android* to be installed. See those sections for further configuration details.

For Makefile Generators and the **Ninja** generator, CMake expects one of these environments:

- *NDK*
- *Standalone Toolchain*

CMake uses the following steps to select one of the environments:

- If the **CMAKE_ANDROID_NDK** variable is set, the NDK at the specified location will be used.
- Else, if the **CMAKE_ANDROID_STANDALONE_TOOLCHAIN** variable is set, the Standalone Toolchain at the specified location will be used.
- Else, if the **CMAKE_SYSROOT** variable is set to a directory of the form **<ndk>/platforms/android−<api>/arch−<arch>**, the **<ndk>** part will be used as the value of **CMAKE_ANDROID_NDK** and the NDK will be used.

- Else, if the **CMAKE_SYSROOT** variable is set to a directory of the form **<standalone−toolchain>/sysroot**, the **<standalone−toolchain>** part will be used as the value of **CMAKE_ANDROID_STANDALONE_TOOLCHAIN** and the Standalone Toolchain will be used.
- Else, if a cmake variable **ANDROID_NDK** is set it will be used as the value of **CMAKE_ANDROID_NDK**, and the NDK will be used.
- Else, if a cmake variable **ANDROID_STANDALONE_TOOLCHAIN** is set, it will be used as the value of **CMAKE_ANDROID_STANDALONE_TOOLCHAIN**, and the Standalone Toolchain will be used.
- Else, if an environment variable **ANDROID_NDK_ROOT** or **ANDROID_NDK** is set, it will be used as the value of **CMAKE_ANDROID_NDK**, and the NDK will be used.
- Else, if an environment variable **ANDROID_STANDALONE_TOOLCHAIN** is set then it will be used as the value of **CMAKE_ANDROID_STANDALONE_TOOLCHAIN**, and the Standalone Toolchain will be used.
- Else, an error diagnostic will be issued that neither the NDK or Standalone Toolchain can be found.

New in version 3.20: If an Android NDK is selected, its version number is reported in the **CMAKE_ANDROID_NDK_VERSION** variable.

Cross Compiling for Android with the NDK

A toolchain file may configure Makefile Generators, Ninja Generators, or Visual Studio Generators to target Android for cross−compiling.

Configure use of an Android NDK with the following variables:

CMAKE_SYSTEM_NAME
Set to **Android**. Must be specified to enable cross compiling for Android.

CMAKE_SYSTEM_VERSION
Set to the Android API level. If not specified, the value is determined as follows:

- If the **CMAKE_ANDROID_API** variable is set, its value is used as the API level.
- If the **CMAKE_SYSROOT** variable is set, the API level is detected from the NDK directory structure containing the sysroot.
- Otherwise, the latest API level available in the NDK is used.

CMAKE_ANDROID_ARCH_ABI
Set to the Android ABI (architecture). If not specified, this variable will default to the first supported ABI in the list of **armeabi**, **armeabi−v7a** and **arm64−v8a**. The**CMAKE_ANDROID_ARCH** variable will be computed from **CMAKE_ANDROID_ARCH_ABI** automatically. Also see the**CMAKE_ANDR OID_ARM_MODE** and **CMAKE_ANDROID_ARM_NEON** variables.

CMAKE_ANDROID_NDK
Set to the absolute path to the Android NDK root directory. If not specified, a default for this variable will be chosen as specified *above*.

CMAKE_ANDROID_NDK_DEPRECATED_HEADERS
Set to a true value to use the deprecated per−api−level headers instead of the unified headers. If not specified, the default will be false unless using a NDK that does not provide unified headers.

CMAKE_ANDROID_NDK_TOOLCHAIN_VERSION
On NDK r19 or above, this variable must be unset or set to **clang**. On NDK r18 or below, set this to the version of the NDK toolchain to be selected as the compiler. If not specified, the default will be the latest available GCC toolchain.

CMAKE_ANDROID_STL_TYPE
Set to specify which C++ standard library to use. If not specified, a default will be selected as described in the variable documentation.

The following variables will be computed and provided automatically:

CMAKE_<LANG>_ANDROID_TOOLCHAIN_PREFIX
The absolute path prefix to the binutils in the NDK toolchain.

CMAKE_<LANG>_ANDROID_TOOLCHAIN_SUFFIX
The host platform suffix of the binutils in the NDK toolchain.

For example, a toolchain file might contain:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_SYSTEM_VERSION 21) # API level
set(CMAKE_ANDROID_ARCH_ABI arm64-v8a)
set(CMAKE_ANDROID_NDK /path/to/android-ndk)
set(CMAKE_ANDROID_STL_TYPE gnustl_static)
```

Alternatively one may specify the values without a toolchain file:

```
$ cmake ../src \
  -DCMAKE_SYSTEM_NAME=Android \
  -DCMAKE_SYSTEM_VERSION=21 \
  -DCMAKE_ANDROID_ARCH_ABI=arm64-v8a \
  -DCMAKE_ANDROID_NDK=/path/to/android-ndk \
  -DCMAKE_ANDROID_STL_TYPE=gnustl_static
```

Cross Compiling for Android with a Standalone Toolchain
A toolchain file may configure Makefile Generators or the **Ninja** generator to target Android for cross-compiling using a standalone toolchain.

Configure use of an Android standalone toolchain with the following variables:

CMAKE_SYSTEM_NAME
Set to **Android**. Must be specified to enable cross compiling for Android.

CMAKE_ANDROID_STANDALONE_TOOLCHAIN
Set to the absolute path to the standalone toolchain root directory. A**${CMAKE_ANDROID_STANDALONE_TOOLCHAIN}/sysroot** directory must exist. If not specified, a default for this variable will be chosen as specified *above*.

CMAKE_ANDROID_ARM_MODE
When the standalone toolchain targets ARM, optionally set this to **ON** to target 32-bit ARM instead of 16-bit Thumb. See variable documentation for details.

CMAKE_ANDROID_ARM_NEON
When the standalone toolchain targets ARM v7, optionally set thisto **ON** to target ARM NEON devices. See variable documentation for details.

The following variables will be computed and provided automatically:

CMAKE_SYSTEM_VERSION
The Android API level detected from the standalone toolchain.

CMAKE_ANDROID_ARCH_ABI
The Android ABI detected from the standalone toolchain.

CMAKE_<LANG>_ANDROID_TOOLCHAIN_PREFIX
The absolute path prefix to the **binutils** in the standalone toolchain.

CMAKE_<LANG>_ANDROID_TOOLCHAIN_SUFFIX
The host platform suffix of the **binutils** in the standalone toolchain.

For example, a toolchain file might contain:

```
set(CMAKE_SYSTEM_NAME Android)
set(CMAKE_ANDROID_STANDALONE_TOOLCHAIN /path/to/android-toolchain)
```

Alternatively one may specify the values without a toolchain file:

```
$ cmake ../src \
  -DCMAKE_SYSTEM_NAME=Android \
  -DCMAKE_ANDROID_STANDALONE_TOOLCHAIN=/path/to/android-toolchain
```

Cross Compiling for Android with NVIDIA Nsight Tegra Visual Studio Edition
A toolchain file to configure one of the Visual Studio Generators to build using NVIDIA Nsight Tegra targeting Android may look like this:

```
set(CMAKE_SYSTEM_NAME Android)
```

The **CMAKE_GENERATOR_TOOLSET** may be set to select the Nsight Tegra "Toolchain Version" value.

See also target properties:

- **ANDROID_ANT_ADDITIONAL_OPTIONS**
- **ANDROID_API_MIN**
- **ANDROID_API**
- **ANDROID_ARCH**
- **ANDROID_ASSETS_DIRECTORIES**
- **ANDROID_GUI**
- **ANDROID_JAR_DEPENDENCIES**
- **ANDROID_JAR_DIRECTORIES**
- **ANDROID_JAVA_SOURCE_DIR**
- **ANDROID_NATIVE_LIB_DEPENDENCIES**
- **ANDROID_NATIVE_LIB_DIRECTORIES**
- **ANDROID_PROCESS_MAX**
- **ANDROID_PROGUARD_CONFIG_PATH**
- **ANDROID_PROGUARD**
- **ANDROID_SECURE_PROPS_PATH**
- **ANDROID_SKIP_ANT_STEP**
- **ANDROID_STL_TYPE**

Cross Compiling for iOS, tvOS, or watchOS
For cross-compiling to iOS, tvOS, or watchOS, the **Xcode** generator is recommended. The **Unix Makefiles** or **Ninja** generators can also be used, but they require the project to handle more areas like target CPU selection and code signing.

Any of the three systems can be targeted by setting the **CMAKE_SYSTEM_NAME** variable to a value from the table below. By default, the latest Device SDK is chosen. As for all Apple platforms, a different SDK (e.g. a simulator) can be selected by setting the **CMAKE_OSX_SYSROOT** variable, although this should rarely be necessary (see *Switching Between Device and Simulator* below). A list of available SDKs can be obtained by running **xcodebuild −showsdks**.

OS	CMAKE_SYSTEM_NAME	Device SDK (default)	Simulator SDK
iOS	iOS	iphoneos	iphonesimulator
tvOS	tvOS	appletvos	appletvsimulator
watchOS	watchOS	watchos	watchsimulator

For example, to create a CMake configuration for iOS, the following command is sufficient:

```
cmake .. -GXcode -DCMAKE_SYSTEM_NAME=iOS
```

Variable **CMAKE_OSX_ARCHITECTURES** can be used to set architectures for both device and simulator. Variable **CMAKE_OSX_DEPLOYMENT_TARGET** can be used to set an iOS/tvOS/watchOS deployment target.

Next configuration will install fat 5 architectures iOS library and add the **−miphoneos−version−min=9.3**/**−mios−simulator−version−min=9.3** flags to the compiler:

```
$ cmake -S. -B_builds -GXcode \
    -DCMAKE_SYSTEM_NAME=iOS \
    "-DCMAKE_OSX_ARCHITECTURES=armv7;armv7s;arm64;i386;x86_64" \
    -DCMAKE_OSX_DEPLOYMENT_TARGET=9.3 \
    -DCMAKE_INSTALL_PREFIX=`pwd`/_install \
    -DCMAKE_XCODE_ATTRIBUTE_ONLY_ACTIVE_ARCH=NO \
    -DCMAKE_IOS_INSTALL_COMBINED=YES
```

Example:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.14)
project(foo)
add_library(foo foo.cpp)
install(TARGETS foo DESTINATION lib)
```

Install:

```
$ cmake --build _builds --config Release --target install
```

Check library:

```
$ lipo -info _install/lib/libfoo.a
Architectures in the fat file: _install/lib/libfoo.a are: i386 armv7 armv7s x8

$ otool -l _install/lib/libfoo.a | grep -A2 LC_VERSION_MIN_IPHONEOS
      cmd LC_VERSION_MIN_IPHONEOS
  cmdsize 16
  version 9.3
```

Code Signing

Some build artifacts for the embedded Apple platforms require mandatory code signing. If the **Xcode** generator is being used and code signing is required or desired, the development team ID can be specified via the **CMAKE_XCODE_ATTRIBUTE_DEVELOPMENT_TEAM** CMake variable. This team ID will then be included in the generated Xcode project. By default, CMake avoids the need for code signing during the internal configuration phase (i.e compiler ID and feature detection).

Switching Between Device and Simulator

When configuring for any of the embedded platforms, one can target either real devices or the simulator. Both have their own separate SDK, but CMake only supports specifying a single SDK for the configuration phase. This means the developer must select one or the other at configuration time. When using the **Xcode** generator, this is less of a limitation because Xcode still allows you to build for either a device or a simulator, even though configuration was only performed for one of the two. From within the Xcode IDE, builds are performed for the selected "destination" platform. When building from the command line, the desired sdk can be specified directly by passing a **−sdk** option to the underlying build tool (**xcodebuild**). For example:

```
$ cmake --build ... -- -sdk iphonesimulator
```

Please note that checks made during configuration were performed against the configure−time SDK and might not hold true for other SDKs. Commands like **find_package()**, **find_library()**, etc. store and use details only for the configured SDK/platform, so they can be problematic if wanting to switch between device and simulator builds. You can follow the next rules to make device + simulator configuration work:

- Use explicit **−l** linker flag, e.g. **target_link_libraries(foo PUBLIC "−lz")**
- Use explicit **−framework** linker flag, e.g. **target_link_libraries(foo PUBLIC "−framework CoreFoundation")**
- Use **find_package()** only for libraries installed with **CMAKE_IOS_INSTALL_COMBINED** feature

COPYRIGHT