cmake-toolchains(7)

Contents

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 `-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.

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. In cases where CMake does not have enough information to extract information from the compiler, the **CMakeForceCompiler** module can be used to bypass some of the 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 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})
```

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 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 using NVIDIA Nsight Tegra

A toolchain file to configure a Visual Studio generator 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 the **ANDROID_API_MIN**, **ANDROID_API** and **ANDROID_GUI** target properties to configure targets within the project.