Chapter 8

Cross Compiling with CMake

Cross compiling a piece of software means that the software is built on one system but intended to run on a different system. The system which is used to build the software will be called the build host, the system for which the software is built will be called the target system or target platform. The target system usually runs a different operating system (or none at all) and/or runs on different hardware. A typical use case is software development for embedded devices like network switches, mobile phones or engine control units. In these cases the target platform doesn't have or is not able to run the required software development environment.

Starting with CMake 2.6.0 cross compiling is fully supported by CMake, ranging from cross compiling from Linux to Windows, cross compiling for supercomputers through to cross compiling for small embedded devices without an operating system (OS).

Cross compiling has several consequences for CMake;

- CMake cannot automatically detect the target platform
- CMake cannot find libraries and headers in the default system directories
- executables built during cross compiling cannot be executed

Cross compiling support doesn't mean that all CMake-based projects can be magically cross compiled out-of-the-box (some are), but that CMake separates between information about the build platform and target platform and gives the user mechanisms to solve cross compiling issues without additional requirements such as running virtual machines, etc.

To support cross compiling for a specific software project, CMake must to be told about the target platform via a so called toolchain file. The CMakeLists.txt may have to be adjusted so they are aware that the build platform may have different properties to the target platform, and it has to deal with the cases where a compiled executable tries to execute on the build host.

8.1 Toolchain Files

In order to use CMake for cross compiling, a CMake file that describes the target platform has to be created, called the "toolchain file". This file tells CMake everything it needs to know about the target platform. Here is an example that uses the MinGW cross compiler for Windows under Linux, the contents will be explained line by line afterwards:

```
# the name of the target operating system
set (CMAKE_SYSTEM_NAME Windows)

# which compilers to use for C and C++
set (CMAKE_C_COMPILER   i586-mingw32msvc-gcc )
set (CMAKE_CXX_COMPILER i586-mingw32msvc-g++ )

# where is the target environment located
set (CMAKE_FIND_ROOT_PATH  /usr/i586-mingw32msvc
     /home/alex/mingw-install )

# adjust the default behavior of the FIND_XXX() commands:
# search programs in the host environment
set (CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)

# search headers and libraries in the target environment
set (CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set (CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

Assuming that this file is saved with the name TC-mingw.cmake in your home directory, you instruct CMake to use this file by setting the `CMAKE_TOOLCHAIN_FILE` variable:

```
~/src$ cd build
~/src/build$ cmake -DCMAKE_TOOLCHAIN_FILE=~/TC-mingw.cmake ..
...
```

`CMAKE_TOOLCHAIN_FILE` has to be specified only on the initial CMake run, after that the results are reused from the CMake cache. You don't need to write a separate toolchain file for every piece of software you want to build. The toolchain files are per target platform, i.e. if you are building several software packages for the same target platform, you only have to write one toolchain file that can be used for all packages. What do the settings in the toolchain file mean? We will examine them one by one. Since CMake cannot guess the target operating system or hardware, you have to set the following CMake variables:

CMAKE_SYSTEM_NAME

This variable is mandatory; it sets the name of the target system, i.e. to the same value as `CMAKE_SYSTEM_NAME` would have if CMake were run on the target system. Typical examples are "Linux" and "Windows". It is used for constructing the file names of the platform files like `Linux.cmake` or `Windows-gcc.cmake`. If your target is an embedded system without an OS then set `CMAKE_SYSTEM_NAME` to "Generic". Presetting `CMAKE_SYSTEM_NAME` this way instead of being detected automatically causes CMake to consider the build a cross compiling build and the CMake variable `CMAKE_CROSSCOMPILING` will be set to `TRUE`. `CMAKE_CROSSCOMPILING` is the variable which should be tested in CMake files to determine whether the current build is a cross compiled build or not.

CMAKE_SYSTEM_VERSION

This variable is optional, it sets the version of your target system. CMake does not currently use `CMAKE_SYSTEM_VERSION`.

CMAKE_SYSTEM_PROCESSOR

This variable is optional, it sets the processor or hardware name of the target system. It is used in CMake for one purpose, load the

```
${CMAKE_SYSTEM_NAME}-COMPILER_ID-${CMAKE_SYSTEM_PROCESSOR}.cmake
```

file. This file can be used to modify settings like compiler flags for the target. You should only have to set this variable if you are using a cross compiler where each target needs special build settings. The value can be chosen freely, so it could be e.g. i386, or IntelPXA255, or MyControlBoardRev42.

In CMake code the `CMAKE_SYSTEM_XXX` variables always describe the target platform. The same is true for the short `WIN32`, `UNIX`, `APPLE` variables. These variables can be used to test the properties of the target. If it is necessary to test the build host system, there is a corresponding set of variables: `CMAKE_HOST_SYSTEM`, `CMAKE_HOST_SYSTEM_NAME`, `CMAKE_HOST_SYSTEM_VERSION`, `CMAKE_HOST_SYSTEM_PROCESSOR` and also the short forms `CMAKE_HOST_WIN32`, `CMAKE_HOST_UNIX` and `CMAKE_HOST_APPLE`.

Since CMake cannot guess the target system, it cannot guess which compiler it should use. Setting the following variables defines what compilers to use for the target system.

CMAKE_C_COMPILER

This specifies the C compiler executable as either a full path or just the filename. If it is specified with full path, then this path will be preferred when searching for the C++ compiler and the other tools (binutils, linker, etc.). If the compiler is a GNU cross compiler with a prefixed name (e.g. "arm-elf-gcc") CMake will detect this and automatically find the corresponding C++ compiler (i.e. "arm-elf-c++"). The compiler can also be set via the CC environment variable. Setting `CMAKE_C_COMPILER` directly in a toolchain file has the advantage that the information about the target system is completely contained in this file, and it does not depend on environment variables.

CMAKE_CXX_COMPILER

This specifies the C++ compiler executable as either a full path or just the filename. It is handled the same way as `CMAKE_C_COMPILER`. If the toolchain is a GNU toolchain, it should suffice to set only `CMAKE_C_COMPILER`, CMake should find the corresponding C++ compiler automatically. As for `CMAKE_C_COMPILER`, also for C++ the compiler can be set via the CXX environment variable.

Once the system and the compiler are determined by CMake, it will load the corresponding files in the order described in section 11.2, The Enable Language Process.

Finding External Libraries, Programs and Other Files

Most non-trivial projects make use of external libraries or tools. CMake offers the `find_program`, `find_library`, `find_file`, `find_path`, and `find_package` commands for this purpose. They search the file system in common places for these files and return the results. `find_package` is a bit different in that it does not actually search itself, but executes Find<*>.cmake modules, which in turn usually call the `find_program`, `find_library`, `find_file` and `find_path` commands.

When cross compiling these commands become more complicated. For example, when cross compiling to Windows on a Linux system, getting /usr/lib/libjpeg.so as the result of the command `find_package(JPEG)` would be useless, since this would be the JPEG library for the host system and not the target system. In some cases you want to find files that are meant for the target platform, in other cases you will want to find files for the build host. The following variables are designed to give you the flexibility to change how the typical find commands in CMake work, so that you can find both build host and target files as necessary.

The toolchain will come with its own set of libraries and headers for the target platform, which are usually installed under a common prefix. It is also a good idea to set up a directory where all the software that is built for the target platform will be installed, so that the software packages don't get mixed up with the libraries that come with the toolchain.

The `find_program` command is usually used to find a program which will be executed during the build, so this should still search in the host file system, not in the environment of the target platform. `find_library` is normally used to find a library which is then used for linking purposes, so this command should only search in the target environment. For `find_path` and `find_file` it is not so obvious, in many cases they are used to search for headers, so by default they should only search in the target environment. The following CMake variables can be set to adjust the behavior of the find commands for cross compiling.

CMAKE_FIND_ROOT_PATH

This is a list of the directories that contain the target environment. Each of the directories listed here will be prepended to each of the search directories of every find command. Assuming your target environment is installed under /opt/eldk/ppc_74xx and your installation for that target platform goes to ~/install-eldk-ppc74xx, set `CMAKE_FIND_ROOT_PATH` to these two directories. Then `find_library` (JPEG_LIB jpeg) will search in /opt/eldk/ppc_74xx/lib, /opt/eldk/ppc_74xx/usr/lib, ~/install-eldk-ppc74xx/lib, ~/install-eldk-ppc74xx/usr/lib, and should result in /opt/eldk/ppc_74xx/usr/lib/libjpeg.so.

By default `CMAKE_FIND_ROOT_PATH` is empty. If set, first the directories prefixed with the path given in `CMAKE_FIND_ROOT_PATH` will be searched, and after that the unprefixed versions of the same directories will be searched.

By setting this variable you are basically adding a new set of search prefixes to all of the find commands in CMake, but for some find commands you may not want to search the target or host directories. You can control how each find command invocation works by passing in one of the three following options `NO_CMAKE_FIND_ROOT_PATH`, `ONLY_CMAKE_FIND_ROOT_PATH` or `CMAKE_FIND_ROOT_PATH_BOTH` when you call it. You can also control how the find commands work using the following three variables.

CMAKE_FIND_ROOT_PATH_MODE_PROGRAM

This sets the default behavior for the `find_program` command. It can be set to `NEVER`, `ONLY` or `BOTH`. The default setting is `BOTH`. When set to `NEVER`, `CMAKE_FIND_ROOT_PATH` will not be used for `find_program` calls except where it is enabled explicitly. If set to `ONLY`, only the search directories with the prefixes coming from `CMAKE_FIND_ROOT_PATH` will be used by `find_program`. The default is `BOTH`, which means that first the prefixed directories, and then the unprefixed directories, will be searched.

In most cases `find_program` is used to search for an executable which will then be executed, e.g. using `execute_process` or `add_custom_command`. So in most cases an executable from the build host is required, so setting `CMAKE_FIND_ROOT_PATH_MODE_PROGRAM` to `NEVER` is normally preferred.

CMAKE_FIND_ROOT_PATH_MODE_LIBRARY

This is the same as above, but for the `find_library` command. In most cases this is used to find a library which will then be used for linking, so a library for the target is required. So in the common case it should be set to `ONLY`.

CMAKE_FIND_ROOT_PATH_MODE_INCLUDE

This is the same as above and used for both `find_path` and `find_file`. In most cases this is used for finding include directories, so the target environment should be searched. In the common case it should be set to `ONLY`. If you also need to find files in the file system of the build host, e.g. some data files that will be processed during the build. You may need to adjust the behavior for those `find_path` or `find_file` calls using the `NO_CMAKE_FIND_ROOT_PATH`, `ONLY_CMAKE_FIND_ROOT_PATH` and `CMAKE_FIND_ROOT_PATH_BOTH` options.

With a toolchain file set up as described, CMake now knows how to handle the target platform and the cross compiler. We should now able to build software for the target platform. For complex projects there are more issues that must to be taken care of.

8.2 System Inspection

Most portable software projects have a set of system inspection tests for determining the properties of the (target) system. The simplest way to check for a system feature with CMake is by testing variables. For this purpose CMake provides the variables `UNIX`, `WIN32` and `APPLE`. When cross compiling, these variables apply to the target platform, for testing the build host platform there are corresponding variables `CMAKE_HOST_UNIX`, `CMAKE_HOST_WINDOWS` and `CMAKE_HOST_APPLE`.

If this granularity is too coarse, the variables `CMAKE_SYSTEM_NAME`, `CMAKE_SYSTEM`, `CMAKE_SYSTEM_VERSION` and `CMAKE_SYSTEM_PROCESSOR` can be tested, along with their counterparts `CMAKE_HOST_SYSTEM_NAME`, `CMAKE_HOST_SYSTEM`, `CMAKE_HOST_SYSTEM_VERSION` and `CMAKE_HOST_SYSTEM_PROCESSOR`, which contain the same information, but for the build host and not for the target system.

```
if (CMAKE_SYSTEM MATCHES Windows)
   message (STATUS "Target system is Windows")
endif ()

if (CMAKE_HOST_SYSTEM MATCHES Linux)
   message (STATUS "Build host runs Linux")
endif ()
```

Using Compile Checks

In CMake there are macros such as CHECK_INCLUDE_FILES and CHECK_C_SOURCE_RUNS that are used to test the properties of the platform. Most of these macros internally use either the try_compile or the try_run commands. The try_compile command works as expected when cross compiling, it tries to compile the piece of code with the cross compiling toolchain, which will give the expected result.

All tests using try_run will not work since the created executables cannot normally run on the build host. In some cases this might be possible, e.g. using virtual machines, emulation layers like Wine or interfaces to the actual target, CMake does not depend on such mechanisms. Depending on emulators during the build process would introduce a new set of potential problems, they may have a different view on the file system, use other line endings, require special hardware or software, etc.

If try_run is invoked when cross compiling, it will first try to compile the software, which will work the same way as when not cross compiling. If this succeeds, it will check the variable CMAKE_CROSSCOMPILING to determine whether the resulting executable can be executed or not. If it cannot, it will create two cache variables, which then have to be set by the user or via the CMake cache. Assume the command looks like this:

```
try_run (SHARED_LIBRARY_PATH_TYPE
         SHARED_LIBRARY_PATH_INFO_COMPILED
         ${PROJECT_BINARY_DIR}/CMakeTmp
         ${PROJECT_SOURCE_DIR}/CMake/SharedLibraryPathInfo.cxx
         OUTPUT_VARIABLE OUTPUT
         ARGS "LDPATH"
         )
```

In this example the source file SharedLibraryPathInfo.cxx will be compiled and if that succeeds, the resulting executable should be executed. The variable SHARED_LIBRARY_PATH_INFO_COMPILED will be set to the result of the build, i.e. TRUE or FALSE. CMake will create a cache variable SHARED_LIBRARY_PATH_TYPE and preset it to PLEASE_FILL_OUT-FAILED_TO_RUN. This variable must be set to what the exit code of the executable would have been if it had been executed on the target. Additionally, CMake will create a cache variable SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT and preset it to PLEASE_FILL_OUT-NOTFOUND. This variable should be set to the output that the executable prints to stdout and stderr if it were executed on the target. This variable is only created if the try_run command was used with the RUN_OUTPUT_VARIABLE or the OUTPUT_VARIABLE argument. You have to fill in the appropriate values for these variables. To help you with this CMake tries its best to give you useful information. To accomplish this CMake creates a file ${CMAKE_BINARY_DIR}/TryRunResults.cmake,that you can see an example of here:

```
# SHARED_LIBRARY_PATH_TYPE
#    indicates whether the executable would have been able to run
#    on its target platform. If so, set SHARED_LIBRARY_PATH_TYPE
#    to the exit code (in many cases 0 for success), otherwise
#    enter "FAILED_TO_RUN".
# SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT
#    contains the text the executable would have printed on
#    stdout and stderr. If the executable would not have been
#    able to run, set SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT
#    empty. Otherwise check if the output is evaluated by the
#    calling CMake code. If so, check what the source file would
#    have printed when called with the given arguments.
# The SHARED_LIBRARY_PATH_INFO_COMPILED variable holds the build
# result for this TRY_RUN().
#
# Source file: ~/src/SharedLibraryPathInfo.cxx
# Executable : ~/build/cmTryCompileExec-SHARED_LIBRARY_PATH_TYPE
# Run arguments:  LDPATH
#     Called from: [1]   ~/src/CMakeLists.cmake

set (SHARED_LIBRARY_PATH_TYPE
     "0"
     CACHE STRING "Result from TRY_RUN" FORCE)

set (SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT
     ""
     CACHE STRING "Output from TRY_RUN" FORCE)
```

You can find all of the variables that CMake could not determine, from which CMake file they were called, the source file, the arguments for the executable and the path to the executable. CMake will also copy the executables to the build directory, they have the names `cmTryCompileExec-<name of the variable>`, e.g. in this case `cmTryCompileExec-SHARED_LIBRARY_PATH_TYPE`. You can then try to run this executable manually on the actual target platform and check the results.

Once you have these results, they have to be put into the CMake cache. This can be done by using ccmake/cmake-gui/"make edit_cache" and editing the variables directly in the cache. It is not possible to reuse these changes in another build directory or if CMakeCache.txt is removed.

The recommended approach is to use the `TryRunResults.cmake` file created by CMake. You should copy it to a safe location (i.e. where it will not be removed if the build directory is deleted), and give it a useful name, e.g. `TryRunResults-MyProject-eldk-ppc.cmake`. The contents of this file have to be edited so that the set commands set the required variables to the appropriate values for the target system. This file can then be used to preload the CMake cache by using the -C option of cmake:

```
src/build/ $ cmake -C ~/TryRunResults-MyProject-eldk-ppc.cmake .
```

You do not have to use the other CMake options again, they are now in the cache. This way you can use `MyProjectTryRunResults-eldk-ppc.cmake` in multiple build trees, and it could be distributed with your project so that it is easier for other users to cross compile it.

8.3 Running Executables Built in the Project

In some cases it is necessary that during a build an executable is invoked that was built earlier in the same build, this is usually the case for code generators and similar tools. This does not work when cross compiling, as the executables are built for the target platform and cannot run on the build host (without the use of virtual machines, compatibility layers, emulators, etc.). With CMake these programs are created using `add_executable`, and executed with `add_custom_command` or `add_custom_target`. The following three options can be used to support these executables with CMake. The old version of the CMake code could look something like this:

```
add_executable (mygen gen.c)
get_target_property (mygenLocation mygen LOCATION)
add_custom_command (
  OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/generated.h"
  COMMAND ${mygenLocation}
       -o "${CMAKE_CURRENT_BINARY_DIR}/generated.h" )
```

Now we will show how this file can be modified so that it works when cross compiling. The basic idea is that the executable is built only when doing a native build for the build host and then exported as an executable target to a CMake script file. This file is then included when cross compiling, and the executable target for the executable mygen will be loaded. An imported target with the same name as the original target will be created. Since CMake 2.6 `add_custom_command` recognizes target names as executables, so for the command in `add_custom_command` simply the target name can be used, it is not necessary to use the `LOCATION` property to obtain the path of the executable:

```
if (CMAKE_CROSSCOMPILING)
   find_package (MyGen)
endif ()

if (NOT CMAKE_CROSSCOMPILING)
```

```
   add_executable (mygen gen.c)
   export (TARGETS mygen FILE
           "${CMAKE_BINARY_DIR}/MyGenConfig.cmake")
endif ()

add_custom_command (
  OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/generated.h"
  COMMAND mygen -o "${CMAKE_CURRENT_BINARY_DIR}/generated.h" )
```

With the CMakeLists.txt modified like this the project can be cross compiled. First, a native build has to be done in order to create the necessary mygen executable. After that the cross compiling build can begin. The build directory of the native build has to be given to the cross compiling build as the location of the MyGen package, so that `find_package(MyGen)` can find it:

```
mkdir build-native; cd build-native
cmake ..
make
cd ..
mkdir build-cross; cd build-cross
cmake -DCMAKE_TOOLCHAIN_FILE=MyToolchain.cmake \
      -DMyGen_DIR=~/src/build-native/ ..
make
```

This code works, but CMake versions prior to 2.6 will not be able to process it, as they do not know the export command and they do not recognize the target name mygen in `add_custom_command`. A compatible version that works with CMake 2.4 looks like this:

```
if (CMAKE_CROSSCOMPILING)
   find_package (MyGen)
endif (CMAKE_CROSSCOMPILING)

if (NOT CMAKE_CROSSCOMPILING)
    add_executable (mygen gen.c)
    if (COMMAND EXPORT)
      export (TARGETS mygen FILE
              "${CMAKE_BINARY_DIR}/MyGenConfig.cmake")
   endif (COMMAND EXPORT)
endif (NOT CMAKE_CROSSCOMPILING)

get_target_property (mygenLocation mygen LOCATION)
```

```
add_custom_command (
  OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/generated.h"
  COMMAND ${mygenLocation}
     -o "${CMAKE_CURRENT_BINARY_DIR}/generated.h" )
```

In this case the target is only exported if the `export` command exists and the location of the executable is retrieved using the `LOCATION` target property.

```
mkdir build-native; cd build-native
cmake ..
make
cd ..
mkdir build-cross; cd build-cross
cmake -DCMAKE_TOOLCHAIN_FILE=MyToolchain.cmake \
      -DMyGen_DIR=~/src/build-native/ ..
make
```

The “old” CMake code could also be using the `utility_source` command:

```
subdirs (mygen)
utility_source (MYGEN_LOCATION mygen mygen gen.c)
add_custom_command (
   OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/generated.h"
   COMMAND ${MYGEN_LOCATION}
      -o "${CMAKE_CURRENT_BINARY_DIR}/generated.h" )
```

In this case the CMake script doesn't have to be changed, but the invocation of CMake is more complicated, since each executable location has to be specified manually:

```
mkdir build-native; cd build-native
cmake ..
make
cd ..
mkdir build-cross; cd build-cross
cmake -DCMAKE_TOOLCHAIN_FILE=MyToolchain.cmake
      -DMYGEN_LOCATION=~/src/build-native/bin/mygen ..
make
```

8.4 Cross Compiling Hello World

Now let's actually start with the cross compiling. The first step is to install a cross compiling toolchain. If this is already installed, you can skip the next paragraph.

There are many different approaches and projects that deal with cross compiling for Linux, ranging from free software projects working on Linux based PDAs to commercial embedded Linux vendors. Most of these projects come with their own way to build and use the respective toolchain. Any of these toolchains can be used with CMake, the only requirement is that it works in the normal file system and does not expect a "sandboxed" environment, like for example the Scratchbox project.

An easy to use toolchain with a relatively complete target environment is the Embedded Linux Development Toolkit (http://www.denx.de/wiki/DULG/ELDK). It supports ARM, PowerPC and MIPS as target platforms. ELDK can be downloaded from ftp://ftp.sunet.se/pub/Linux/distributions/eldk/. The easiest way is to download the ISOs, mount them and then install them:

```
mkdir mount-iso/
sudo mount -tiso9660 mips-2007-01-21.iso mount-iso/ -o loop
cd mount-iso/
./install -d /home/alex/eldk-mips/
...
Preparing...
########################################### [100%]
   1:appWeb-mips_4KCle
########################################### [100%]
Done
ls /opt/eldk-mips/
bin  eldk_init  etc  mips_4KC  mips_4KCle  usr  var  version
```

ELDK (and other toolchains) can be installed anywhere, either in the home directory or system wide if there are more users working with them. In this example the toolchain will now be located in /home/alex/eldk-mips/usr/bin/ and the target environment is in /home/alex/eldk-mips/mips_4KC/.

Now that a cross compiling toolchain is installed, CMake has to be set up to use it. As already described, this is done by creating a toolchain file for CMake. In this example the toolchain file looks like this:

```
# the name of the target operating system
set (CMAKE_SYSTEM_NAME Linux)
```

```
# which C and C++ compiler to use
set (CMAKE_C_COMPILER /home/alex/eldk-mips/usr/bin/mips_4KC-gcc)
set (CMAKE_CXX_COMPILER
     /home/alex/eldk-mips/usr/bin/mips_4KC-g++)

# location of the target environment
set (CMAKE_FIND_ROOT_PATH /home/alex/eldk-mips/mips_4KC
                          /home/alex/eldk-mips-extra-install )

# adjust the default behavior of the FIND_XXX() commands:
# search for headers and libraries in the target environment,
# search for programs in the host environment
set (CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set (CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set (CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

The toolchain files can be located anywhere, but it is a good idea to put them in a central place so that they can be reused in multiple projects. We will save this file as `~/Toolchains/Toolchain-eldk-mips4K.cmake`. The variables mentioned above are set here: `CMAKE_SYSTEM_NAME`, the C/C++ compilers, `CMAKE_FIND_ROOT_PATH` to specify where libraries and headers for the target environment are located. The find modes are also set up so that libraries and headers are searched for in the target environment only, whereas programs are searched for in the host environment only. Now we will cross compile the hello world project from Chapter 2:

```
project (Hello)
add_executable (Hello Hello.c)
```

Run CMake, this time telling it to use the toolchain file from above:

```
mkdir Hello-eldk-mips
cd Hello-eldk-mips
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-eldk-
mips4K.cmake ..
make VERBOSE=1
```

This should give you an executable that can run on the target platform. Thanks to the `VERBOSE=1` option you should see that the cross compiler is used. Now we will make the example a bit more sophisticated by adding system inspection and install rules. We will build and install a shared library named Tools, and then build the Hello application which links to the Tools library.

```
include (CheckIncludeFiles)
check_include_files (stdio.h HAVE_STDIO_H)

set (VERSION_MAJOR 2)
set (VERSION_MINOR 6)
set (VERSION_PATCH 0)

configure_file (config.h.in ${CMAKE_BINARY_DIR}/config.h)

add_library (Tools  SHARED  tools.cxx)
set_target_properties (Tools PROPERTIES
    VERSION ${VERSION_MAJOR}.${VERSION_MINOR}.${VERSION_PATCH}
    SOVERSION ${VERSION_MAJOR})

install (FILES tools.h DESTINATION include)
install (TARGETS Tools DESTINATION lib)
```

There is no difference to a normal CMakeLists.txt, no special prerequisites are required for cross compiling. The CMakeLists.txt checks that the header `stdio.h` is available and sets the version number for the Tools library. These are configured into `config.h`, which is then used in `tools.cxx`. The version number is also used to set the version number of the Tools library. The library and headers are installed to `${CMAKE_INSTALL_PREFIX}/lib` and `${CMAKE_INSTALL_PREFIX}/include` respectively. Running CMake gives this result:

```
mkdir build-eldk-mips
cd build-eldk-mips
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-eldk-
mips4K.cmake -DCMAKE_INSTALL_PREFIX=~/eldk-mips-extra-install ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /home/alex/eldk-
mips/usr/bin/mips_4KC-gcc
-- Check for working C compiler: /home/alex/eldk-
mips/usr/bin/mips_4KC-gcc -- works
-- Check size of void*
-- Check size of void* - done
-- Check for working CXX compiler: /home/alex/eldk-
mips/usr/bin/mips_4KC-g++
-- Check for working CXX compiler: /home/alex/eldk-
mips/usr/bin/mips_4KC-g++ -- works
-- Looking for include files HAVE_STDIO_H
-- Looking for include files HAVE_STDIO_H - found
-- Configuring done
```

```
-- Generating done
-- Build files have been written to:
/home/alex/src/tests/Tools/build-mips
make install
Scanning dependencies of target Tools
[100%] Building CXX object CMakeFiles/Tools.dir/tools.o
Linking CXX shared library libTools.so
[100%] Built target Tools
Install the project...
-- Install configuration: ""
-- Installing /home/alex/eldk-mips-extra-install/include/tools.h
-- Installing /home/alex/eldk-mips-extra-install/lib/libTools.so
```

As can be seen in the output above, CMake detected the correct compiler, found the `stdio.h` header for the target platform and successfully generated the Makefiles. The make command was invoked, which then successfully built and installed the library in the specified installation directory. Now we can build an executable that uses the Tools library and does some system inspection:

```
project (HelloTools)

find_package (ZLIB REQUIRED)

find_library (TOOLS_LIBRARY Tools)
find_path (TOOLS_INCLUDE_DIR tools.h)

if (NOT TOOLS_LIBRARY OR NOT TOOLS_INCLUDE_DIR)
  message (FATAL_ERROR "Tools library not found")
endif (NOT TOOLS_LIBRARY OR NOT TOOLS_INCLUDE_DIR)

set (CMAKE_INCLUDE_CURRENT_DIR TRUE)
set (CMAKE_INCLUDE_DIRECTORIES_PROJECT_BEFORE TRUE)
include_directories ("${TOOLS_INCLUDE_DIR}"
                     "${ZLIB_INCLUDE_DIR}")

add_executable (HelloTools main.cpp)
target_link_libraries (HelloTools ${TOOLS_LIBRARY}
                       ${ZLIB_LIBRARIES})
set_target_properties (HelloTools PROPERTIES
                       INSTALL_RPATH_USE_LINK_PATH TRUE)

install (TARGETS HelloTools DESTINATION bin)
```

Building works in the same way as with the library, the toolchain file has to be used, and then it should just work:

```
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-eldk-
mips4K.cmake -DCMAKE_INSTALL_PREFIX=~/eldk-mips-extra-install ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /home/alex/denx-
mips/usr/bin/mips_4KC-gcc
-- Check for working C compiler: /home/alex/denx-
mips/usr/bin/mips_4KC-gcc -- works
-- Check size of void*
-- Check size of void* - done
-- Check for working CXX compiler: /home/alex/denx-
mips/usr/bin/mips_4KC-g++
-- Check for working CXX compiler: /home/alex/denx-
mips/usr/bin/mips_4KC-g++ -- works
-- Found ZLIB: /home/alex/denx-mips/mips_4KC/usr/lib/libz.so
-- Found Tools library: /home/alex/denx-mips-extra-
install/lib/libTools.so
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/alex/src/tests/HelloTools/build-eldk-mips
make
[100%] Building CXX object CMakeFiles/HelloTools.dir/main.o
Linking CXX executable HelloTools
[100%] Built target HelloTools
```

Obviously CMake found the correct zlib and also `libTools.so`, that had been installed in the previous step.

8.5 Cross Compiling for a Microcontroller

CMake can be used for more than cross compiling to targets with operating systems, it is also possible to use it in development for deeply embedded devices with small microcontrollers and no operating system at all. As an example we will use the Small Devices C Compiler (http://sdcc.sourceforge.net), which runs under Windows, Linux and Mac OS X, that supports 8 and 16 Bit microcontrollers. For driving the build we will use MS NMake under Windows. As before, the first step is to write a toolchain file so that CMake knows about the target platform. For sdcc it should look something like this:

```
set (CMAKE_SYSTEM_NAME Generic)
set (CMAKE_C_COMPILER "c:/Program Files/SDCC/bin/sdcc.exe")
```

The system name for targets that do not have an operating system, "Generic", should be used as the `CMAKE_SYSTEM_NAME`. The CMake platform file for "Generic" doesn't set up any specific features. All that it assumes is that the target platform does not support shared libraries, and so all properties will depend on the compiler and `CMAKE_SYSTEM_PROCESSOR`. The toolchain file above does not set the FIND-related variables. As long as none of the find commands is used in the CMake commands, this is fine. In many projects for small microcontrollers this will be the case. The CMakeLists.txt should look like the following:

```
project (Blink C)

add_library (blink blink.c)

add_executable (hello main.c)
target_link_libraries (hello blink)
```

There are no major differences to other CMakeLists.txt files. One important point is that the language "C" is enabled explicitly using the `PROJECT` command. If this is not done, CMake will also try to enable support for C++, which will fail as sdcc only has support for C. Running CMake and building the project should work as usual:

```
cmake -G"NMake Makefiles"
  -DCMAKE_TOOLCHAIN_FILE=c:/Toolchains/Toolchain-sdcc.cmake ..
-- The C compiler identification is SDCC
-- Check for working C compiler: c:/program
files/sdcc/bin/sdcc.exe
-- Check for working C compiler: c:/program
files/sdcc/bin/sdcc.exe -- works
-- Check size of void*
-- Check size of void* - done
-- Configuring done
-- Generating done
-- Build files have been written to: C:/src/tests/blink/build

nmake
Microsoft (R) Program Maintenance Utility Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

Scanning dependencies of target blink
[ 50%] Building C object CMakeFiles/blink.dir/blink.rel
```

```
Linking C static library blink.lib
[ 50%] Built target blink
Scanning dependencies of target hello
[100%] Building C object CMakeFiles/hello.dir/main.rel
Linking C executable hello.ihx
[100%] Built target hello
```

This was a simple example using NMake with sdcc with the default settings of sdcc. Of course more sophisticated project layouts are possible. For this kind of project it is also a good idea to setup an install directory where reusable libraries can be installed, so it is easier to use them in multiple projects. It is normally necessary to choose the correct target platform for sdcc, not everybody uses i8051, which is the default for sdcc. The recommended way to do this is via setting `CMAKE_SYSTEM_PROCESSOR`.

This will cause CMake to search for and load the platform file `Platform/Generic-SDCC-C-${CMAKE_SYSTEM_PROCESSOR}.cmake`. As this happens right before loading `Platform/Generic-SDCC-C.cmake`, it can be used to setup the compiler and linker flags for the specific target hardware and project. Therefore, a slightly more complex toolchain file is required:

```
get_filename_component (_ownDir
                        "${CMAKE_CURRENT_LIST_FILE}" PATH)
set (CMAKE_MODULE_PATH "${_ownDir}/Modules"
${CMAKE_MODULE_PATH})

set (CMAKE_SYSTEM_NAME Generic)
set (CMAKE_C_COMPILER "c:/Program Files/SDCC/bin/sdcc.exe")
set (CMAKE_SYSTEM_PROCESSOR "Test_DS80C400_Rev_1")

# here is the target environment located
set (CMAKE_FIND_ROOT_PATH  "c:/Program Files/SDCC"
                           "c:/ds80c400-install" )

# adjust the default behavior of the FIND_XXX() commands:
# search for headers and libraries in the target environment
# search for programs in the host environment
set (CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set (CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set (CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

This toolchain file contains a few new settings, it is also about the most complicated toolchain file you should ever need. `CMAKE_SYSTEM_PROCESSOR` is set to Test_DS80C400_Rev_1, which is just an identifier for the specific target hardware. This has the effect that CMake will try to load Platform/Generic-SDCC-C-Test_DS80C400_Rev_1.cmake. As this file does not exist in the CMake system module directory, the CMake variable `CMAKE_MODULE_PATH` has to be adjusted so that this file can be found. If this toolchain file is saved to `c:/Toolchains/sdcc-ds400.cmake`, the hardware specific file should be saved in `c:/Toolchains/Modules/Platform/`. An example of this is shown below:

```
set (CMAKE_C_FLAGS_INIT "-mds390 --use-accelerator")
set (CMAKE_EXE_LINKER_FLAGS_INIT "")
```

This will select the DS80C390 as the target platform and add the --use-accelerator argument to the default compile flags. In this example the "NMake Makefiles" generator was used. In the same way e.g. the "MinGW Makefiles" generator could be used if GNU make from MinGW, or another Windows version of GNU make, are available. At least version 3.78 is required, or the "Unix Makefiles" generator under UNIX. Also any Makefile-based IDE-project generators could be used, e.g. the Eclipse, CodeBlocks, or the KDevelop3 generator.

8.6 Cross Compiling an Existing Project

Existing CMake based projects may need some work so that they can be cross compiled, other projects may work without any modifications. One such project is FLTK, the Fast Lightweight Toolkit. We will compile FLTK on a Linux machine using the MinGW cross compiler for Windows.

The first step is to install the MinGW cross compiler. For some Linux distributions there are ready-to-use binary packages, for Debian the package name is mingw32. Once this is installed you need to setup a toolchain file for this as described above. It should look something like this:

```
# the name of the target operating system
set (CMAKE_SYSTEM_NAME Windows)

# which compiler to use
set (CMAKE_C_COMPILER i586-mingw32msvc-gcc)
set (CMAKE_CXX_COMPILER i586-mingw32msvc-g++)

# where are the target libraries and headers installed ?
set(CMAKE_FIND_ROOT_PATH /usr/i586-mingw32msvc
                         /home/alex/mingw-install )

# find_program() should by default NEVER search the target tree
# adjust the default behavior of the FIND_XX() commands:
# search for headers and libraries in the target environment
```

```
# search for programs in the host environment
set (CMAKE_FIND_ROOT_MODE_PROGRAM NEVER)
set (CMAKE_FIND_ROOT_MODE_LIBRARY ONLY)
set (CMAKE_FIND_ROOT_MODE_INCLUDE ONLY)
```

Once this is working, run CMake with the appropriate options on FLTK:

```
mkdir build-mingw
cd build-mingw
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-
mingw32.cmake -DCMAKE_INSTALL_PREFIX=~/mingw-install ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/i586-mingw32msvc-gcc
-- Check for working C compiler: /usr/bin/i586-mingw32msvc-gcc -
- works
...
```

In FLTK the `utility_source` command is used to build the executable fluid, whose location is put into the CMake variable `FLUID_COMMAND`. If you intend to run this executable, you need to preload the cache with the full path to a version of that program that can be run on the build host.

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/alex/src/fltk-1.1.x-
r5940/build-mingw
```

Below you can see a warning from CMake about the use of the `utility_source` command. To find out more CMake offers the --debug-output argument:

```
rm -rf *
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-
mingw32.cmake -DCMAKE_INSTALL_PREFIX=~/mingw-install .. --debug-
output
...
UTILITY_SOURCE is used in cross compiling mode for
FLUID_COMMAND. If your intention is to run this executable, you
need to preload the cache with the full path to a version of
that program, which runs on this build machine.
Called from: [1] /home/alex/src/fltk-1.1.x-r5940/CMakeLists.txt
```

This tells us that utility_source has been called from /home/alex/src/fltk-1.1.x-r5940/CMakeLists.txt, then CMake processed some more directories, finally it created Makefiles in each subdirectory. Examining the top level CMakeLists.txt shows the following:

```
# Set the fluid executable path
utility_source (FLUID_COMMAND fluid fluid fluid.cxx)
set (FLUID_COMMAND "${FLUID_COMMAND}" CACHE INTERNAL "" FORCE)
```

Apparently FLUID_COMMAND is used to hold the path for the executable fluid, which is built by the project. Fluid is used during the build to generate code, so the cross compiled executable will not work, instead a native fluid has to be used. In the following example the variable FLUID_COMMAND is set to the location of a fluid executable for the build host, which is then used in the cross compiling build to generate code that will be compiled for the target system:

```
cmake . -DFLUID_COMMAND=/home/alex/src/download/fltk-1.1.x-
r5940/build-native/bin/fluid
...
-- Configuring done
-- Generating done
make
Scanning dependencies of target fltk_zlib
[  0%] Building C object
zlib/CMakeFiles/fltk_zlib.dir/adler32.obj
[  0%] Building C object
zlib/CMakeFiles/fltk_zlib.dir/compress.obj
...
Scanning dependencies of target valuators
[100%] Building CXX object
test/CMakeFiles/valuators.dir/valuators.obj
Linking CXX executable ../bin/valuators.exe
[100%] Built target valuators
```

That's it, the executables are now in mingw-bin/, and can be run via wine or by copying them to a Windows system.

8.7 Cross Compiling a Complex Project - VTK

Building a complex project is a multi-step process. Complex in this case means that the project uses tests that run executables, and that it builds executables which are used later in the build to generate code (or something similar). One such project is VTK, the Visualization

Toolkit. It uses several `try_run` tests and creates several code generators. When running CMake on the project, every `try_run` command will produce an error message and at the end there will be a `TryRunResults.cmake` file in the build directory. You need to go through all of the entries of this file and fill in the appropriate values. If you are uncertain about the correct result, you can also try to execute the test binaries on the real target platform, they are saved in the binary directory.

VTK contains several code generators, one of which is ProcessShader. These code generators are added using `add_executable` and `get_target_property(LOCATION)` is used to get the locations of the resulting binaries, which are then used in `add_custom_command` or `add_custom_target` commands. Since the cross compiled executables cannot be executed during the build, the `add_executable` calls are surrounded by `if (NOT CMAKE_CROSSCOMPILING)` commands and the executable targets are imported into the project using the `add_executable` command with the `IMPORTED` option. These import statements are in the file `VTKCompileToolsConfig.cmake`, which does not have to be created manually, but it is created by a native build of VTK.

So in order to cross compile VTK you need

- install a toolchain and create a toolchain file for CMake
- build VTK natively for the build host
- run CMake for the target platform
- complete `TryRunResults.cmake`
- use the `VTKCompileToolsConfig.cmake` file from the native build
- finally build

So first, build a native VTK for the build host using the standard procedure.

```
cvs -d :pserver:anonymous@public.kitware.com:/cvsroot/VTK co VTK
cd VTK
mkdir build-native; cd build-native
ccmake ..
make
```

Ensure that all required options are enabled using ccmake, e.g. if you need Python wrapping for the target platform you must enable Python wrapping in `build-native/`. Once this build has finished, there will be a `VTKCompileToolsConfig.cmake` file in `build-native/`. If this succeeded, we can continue to cross compiling the project, in this example for an IBM BlueGene supercomputer.

```
cd VTK
mkdir build-bgl-gcc
cd build-bgl-gcc
cmake -DCMAKE_TOOLCHAIN_FILE=~/Toolchains/Toolchain-BlueGeneL-
gcc.cmake -DVTKCompileTools_DIR=~/VTK/build-native/ ..
```

This will finish with an error message for each `try_run` and a `TryRunResults.cmake` file, that you have to complete as described above. You should save the file to a safe location, otherwise it will be overwritten on the next CMake run.

```
cp TryRunResults.cmake ../TryRunResults-VTK-BlueGeneL-gcc.cmake
ccmake -C ../TryRunResults-VTK-BlueGeneL-gcc.cmake .
...
make
```

On the second run of ccmake all the other arguments can be skipped as they are now in the cache. It is possible to point CMake to the build directory that contains a CMakeCache.txt, so CMake will figure out that this is the build directory.

8.8 Some Tips and Tricks

Dealing with try_run tests

In order to make cross compiling your project easier, try to avoid `try_run` tests and use other methods to test something instead. For examples of how this can be done consider the tests for endianess in `CMake/Modules/TestBigEndian.cmake`, and the test for the compiler id using the source file `CMake/Modules/CMakeCCompilerId.c`. In both `try_compile` is used to compile the source file into an executable, where the desired information is encoded into a text string. Using the `COPY_FILE` option of `try_compile` this executable is copied to a temporary location and then all strings are extracted from this file using `file (STRINGS)`. The test result is obtained using regular expressions to get the information from the string.

If you cannot avoid `try_run` tests, try to use just the exit code from the run, not the output of the process. That way it will not be necessary to set both the exit code and the stdout and stderr variables for the `try_run` test when cross compiling. This allows the `OUTPUT_VARIABLE` or the `RUN_OUTPUT_VARIABLE` options for `try_run` to be omitted.

If you have done that, created and completed a correct `TryRunResults.cmake` file for the target platform, you might consider adding this file to the sources of the project, so that it can be reused by others. These files are per-target per-toolchain.

Target platform and toolchain issues

If your toolchain is not able to build a simple program without special arguments, like e.g. a linker script file or a memory layout file, the tests CMake does initially will fail. To make it work anyway, there is a CMake module, CMakeForceCompiler, that offers the following macros:

```
CMAKE_FORCE_SYSTEM (name version processor),
CMAKE_FORCE_C_COMPILER (compiler compiler_id sizeof_void_p)
CMAKE_FORCE_CXX_COMPILER (compiler compiler_id).
```

These macros can be used in a toolchain file so that the required variables will be preset and the CMake tests avoided.

RPATH handling under UNIX

For native builds CMake builds executables and libraries by default with RPATH. In the build tree the RPATH is set so that the executables can be run from the build tree, i.e. the RPATH points into the build tree. When installing the project, CMake links the executables again, this time with the RPATH for the install tree, which is empty by default.

When cross compiling you probably want to set up RPATH handling differently, as the executable cannot run on the build host it makes more sense to build it with the install RPATH right from the start. There are several CMake variables and target properties for adjusting RPATH handling.

```
set (CMAKE_BUILD_WITH_INSTALL_RPATH TRUE)
set (CMAKE_INSTALL_RPATH "<whatever you need>")
```

With these two settings the targets will be built with the install RPATH instead of the build RPATH, this avoids the need to link them again when installing. If you don't need RPATH support in your project, you don't need to set `CMAKE_INSTALL_RPATH`, it is empty by default.

Setting `CMAKE_INSTALL_RPATH_USE_LINK_PATH` to `TRUE` is useful for native builds, since it automatically collects the RPATH from all libraries against which a targets links. For cross compiling it should be left at the default setting, which is `FALSE`, because on the target the automatically generated RPATH will be wrong in most cases, it will probably have a different file system layout to the build host.