CMake Build System

From OSDev Wiki

CMake is a cross-platform, multi-environment build system which can remove some of the headache from building your operating system. The aim of this page is to help the reader set up his or her build environment to use CMake.

Difficulty level

Beginner

Contents

About CMake

CMake is makefile generator that allows you to provide a description of your build environment, therefore affording the creation of toolchain-specific build files. It is an open-source project that is largely maintained by KitWare, Inc.

Who uses CMake?

Because CMake was developed by KitWare, it is closely associated with their software. However, momentum for CMake has steadily been increasing, and some fairly high-profile projects have switched over to it, including:

- The K Desktop Environment
- Second Life
- Apache QPid
- ReactOS

Getting Started

Design Considerations

There are a variety of options at your disposal when you design your CMake project. It is possible to have CMake cater to some of your personal build preferences, and as a result, you should probably ask yourself a few questions:

- Will my build be performed in-source or out-of-source?
- How granular is the project? Will there be a bunch of small components I plan to link together?
- What prerequisites am I considering for my build?
- Will I be generating any code from templates?

The first question is largely one of preference. An in-source build means that the build output will be placed in the same directory as the project. This might be the root directory of the project or (assuming a heirarchical organization) subdirectories associated with the unit being compiled. However, the generated files mingling with sources can be an annoyance, especially when using version-control systems (CMake does not have a `cmake clean` command). Some people prefer to send the build output to a separate folder, perhaps named "build", or even multiple directories for debug and release builds. This sort of design is called an out-of-source build, and can make building for multiple targets easier and more flexible for all but the simplest projects.

Project directory structure is also worth strong consideration. In a quick academic exercise, a flat single-directory layout may be desirable. However, if you plan to go beyond this stage, it may be a good idea to consider leveraging the filesystem to your advantage. Consider the following layout, for instance:

```
src
|
+----+ kernel
|    |
|    +----+ i386
|    |    |
|    |    +---- pc
|    |
|    +----+ arm
|         |
|         +---- rasberrypi
|         +---- beagle
|
+----+ libc
     |
     +---- ...
```

The above might be the natural way of organizing your source code and we can leverage a directory structure like this to our advantage. Consider the following advantages for this directory tree:

- Sub-projects are neatly isolated from one another. Kernel and libc are definitely related, but we might prefer to think of them as separate projects.
- ISA-dependent code can be isolated from common code or other ISA-dependent code.
- Platform-dependent code (which may add further restrictions to ISA-dependent code) is separated from other platforms sharing the same ISA.

The chances are that if you're building an operating system, you won't have many library dependencies. Most of your dependencies will be related to the toolchain required to build the code. For instance, you will probably want a C compiler, and almost certainly an assembler to go with it. You may write your own tools to simplify the process of creating your operating system, and in that case, you may want to ensure support for other programming environments like Python or Perl. Fortunately, this part can be fairly forgiving: adding dependencies such as these is not terribly difficult, and if you change your mind, the change is easy to implement.

A Simple CMakeLists.txt

One of the nice things about CMake is that it affords you several programming concepts that you are already familiar with. As one would expect, CMake lets you perform build configuration through variables, which are typically defined in user-provided files named CMakeLists.txt. The resulting values of these variables, after cmake processing, can conveniently be found in a file called CMakeCache.txt. CMakeCache.txt is generated when you first run CMake on a CMakeLists.txt file and can be tweaked to provide various special build options for your script. However, the preferred method is to use the `SET()` macro (and other macros) in a CMakeLists.txt file, which defines a variable name with its first argument and the variable's value using the rest of the argument(s).

Of course, variables aren't really enough, so CMake provides two kinds of procedures: macros and functions. Functions and macros are very subtly different: functions create their own environment so the variables defined in a function are limited to the scope of the callee and below by default. Macros, on the other hand, will place any variables defined within the macro in the parent's environment. The bulk of CMake is implemented using functions and macros; this permits code reuse that is far superior to the targets provided by make. Note that some of these builtin commands will automatically SET variables on our behalf. This is particularly useful when we want to deal with dependency resolution.

A useable CMakeLists.txt can have as little as two lines as code. One of the most important families of macros include `ADD_EXECUTABLE()` or `ADD_LIBRARY()`. Each of these macros take several parameters: the target name as the first parameter and the source file(s) on which the target depends on as the remaining parameter(s).

The `CMAKE_MINIMUM_REQUIRED()` function, which takes the version of CMake necessary to parse the CMakeList.txt file is also required, because that is how CMake can tell whether or not it meets the requirements to parse your script.

However, such a script has a key disadvantage: every time we add a new source file, we need to edit CMakeLists.txt to add it. Fortunately, CMake comes with a built in file manipulation interface. In this case, we could use the GLOB operator for the `FILE()` command, which lets us specify a file globbing pattern to collect all of our source files or file paths into one variable. The addition of this extra line of code can make CMake much more useful.

If you have dependencies you need to resolve, CMake can handle that too. The `FIND_PACKAGE()` command can handle a variety of different types of dependencies, including libraries (like Boost). `FIND_PACKAGE()` will also attempt to locate a script by the name of `Find<1st param>.cmake` and handle it appropriately. In instances where `FIND_PACKAGE()` is unable to resolve the package, it sets a special variable called `<1st param>_NOTFOUND`, which you might use to detect optional dependencies. However, if a dependency is absolutely required, then you can simply supply `REQUIRED` as the second parameter. Failing to find the package will cause CMake to bail out.

You can emit messages back to the console using the `MESSAGE()` command. Note that this is only run at CMake time. You can use this command for debugging your CMake project. Alternately, if you supply `STATUS` as the first parameter to this command, it will print out a specialized status message for you.

```
# So CMake can tell whether or not it can process this file
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.0)

# Require Perl (for whatever reason)
FIND_PACKAGE(PERL REQUIRED)
```

```
MESSAGE(STATUS "Hi!")

# Grab all of the C files in src; store in C_SRCS
FILE(GLOB C_SRCS src/*.c)

# Note how we obtain this variable's value!
ADD_EXECUTABLE(foo ${C_SRCS})
```

This is enough for a small project to generate an executable. Creating the associated Makefile and starting the build is simple:

```
$ cd project/
$ mkdir build
$ cd build/
$ cmake ../
$ make
```

Note that if you chance CMakeLists.txt, you will need to run CMake again. In cases like these, CMake may need to update CMakeCache.txt, for instance. Particularly in the case where you use file globbing find your source files, it is imperative that you do this when you add or delete source files; otherwise, CMake will not pick up the changes, and havoc will break loose.

Sub-Project Isolation

The previous section discussed a sort of "Hello World" implementation of CMakeLists.txt. Unfortunately, operating system development is rarely in the same class as "Hello World", and the chances are that you have advanced beyond intro computer science. If you're reading this, you more than likely have an idea of how you want to structure your project, and that probably means breaking it into manageable pieces. If you followed the advice from Design Considerations, then it is very likely that you have thought about this a great deal. The real question is how to put this into practice.

A common approach involves generating custom build scripts for each sub-project you create. Thus, we have a separate CMakeLists.txt for each sub-project we create, and we link them together in the CMakeLists.txt in the project root. If we adhere to our filesystem layout from above, for instance, we'd have 3 CMakeLists.txt files: one in /, one in /src/kernel, and one in /src/libc. We can link them together with the following:

```
ADD_SUBDIRECTORY(src/kernel)
ADD_SUBDIRECTORY(src/libc)
```

Using the `ADD_SUBDIRECTORY()` command is analagous to recursively calling make, and in many cases, this is precisely what happens. Other generators may interpret this command differently: for instance, the MSVC generator might decide to turn these directories into multiple projects within a solution. In either case, the child CMakeLists.txt inherits the environment of the parent so variables are propagated. You can use this to your advantage by doing dependency resolution and setting up critical shared variables in the root CMakeLists.txt file.

Alternately, you can leverage the `INCLUDE()` command to directly insert CMake code into your CMakeLists.txt file at its point of invocation, which can be useful for important small snippets of code into your project. Note that there are some subtle differences between `INCLUDE()` and `ADD_SUBDIRECTORY()`:

- You can use `INCLUDE()` to include any file as CMake code. `ADD_SUBDIRECTORY()` expects a CMakeLists.txt file in the directory you point it at.
- `INCLUDE()` operates from the current working directory. `ADD_SUBDIRECTORY()` will change the current working directory to the supplied path before evaluating.

Applying CMake to your Operating System

Building Assembly Code

Unless you intend to use somebody else's kernel and write your operating system completely from portable code, it is very likely that you will need an assembler. For the kinds of projects that CMake was designed for, this rarely comes up; as an operating system designer, such support is probably critical to your project. Fortunately, some work has been done to address this issue and in most cases you can get away with not only detecting an assembler, but even specifying the syntax it uses. By using the `ENABLE_LANGUAGE()` command it is possible to turn on support for assembly:

```
# We want an AT&T Syntax Assembler
ENABLE_LANGUAGE(ASM-ATT)

ADD_EXECUTABLE(foo bar.s baz.s)
```

Setting Appropriate Build Options

Depending on your project the stock compiler options may be insufficient for your needs. You may need to supply switches to your toolchain that affect linking or object assembly. CMake provides a number of ways of doing this:

- For C programs, you can use `CMAKE_C_FLAGS` in the same way you would use `$CFLAGS` in the context of make. `ADD_DEFINITIONS()` can also be used, but it is probably inadvisable to do so since a C flag variable exists by default.
- For other languages, (including assembly) use `CMAKE_<lang>_COMPILE_OBJECT`. For instance, if ASM-ATT is enabled, one would modify `CMAKE_ASM-ATT_COMPILE_OJBECT`.
- Link-time options can be set using `SET_TARGET_FLAGS(target PROPERTIES LINK_FLAGS "flags")`.

Build Profile Detection

Remember how we made the claim that the directory structure could be leveraged in order to help make our lives easier? You might have gotten some idea from the directory structure of how this might be accomplished, at least on a conceptual level. Consider the example provided by src/kernel. Here, we have a well-defined hierarchy which allows us to narrow down on varying scopes of the actual kernel implementation. We can split our code into three directories, such that:

- `/src/kernel/` contains the code for the kernel. (platform-indepdendent code might go into this

directory).

- `/src/kernel/isa/` contains code specifically for the instruction set architecture isa (e.g., i386).
- `/src/kernel/isa/platform/` contains code for the platform which is implemented with isa.

Let's consider the ARM branch of our code. While the i386 may see limited usage outside of IBM-compatible PCs, the ARM conversely is found in a number of environments. Many of those environments have their own quirks, such as how the memory bus is physically mapped. Will the kernel binary for a BeagleBone run on a Raspberry Pi? It is possible, but unlikely. Even if it did, what if these two platforms are fundamentally incompatible at some level? We need to take that into consideration. What we need is a macro that does all of the dirty work for us.

We can take two approaches to this problem: we can either make the assumption platforms (and ISAs for that matter) share some things in common, or make the assumption that they don't. In the former approach, we might assume that the following is true of platforms:

- Each platform provides a memory layout via a linker script.
- Each platform provides a CMake file describing its own build flags.
- Each platform provides a number of C or Assembly sources.

And likewise, we might say that each ISA provides:

- A CMake file describing its own build flags.
- A number of C or Assembly sources.

So far, so good. We can approach this by writing a function in order to handle loading the right profile. When you write a function, you delimit the function body between the `FUNCTION()` and `ENDFUNCTION()` commands. The first parameter to `FUNCTION()` shall be the name of the function you are defining, and those remaining are formal parameters to the function.

However, there is a slight problem: we need a way to report our findings to the caller. Remember that variables defined in functions go out of scope as soon as the function ends. We could use a macro here at the risk of polluting the namespace but it would be much better if we could export variables to the parent scope. Fortunately, the `SET()` command accepts `PARENT_SCOPE` as an optional parameter. When this is used, it means that the variable being set should become part of the parent's environment. Let's write a small build profile function now:

```
FUNCTION(LOAD_PROFILE ISA PLATFORM)
  # Obtain sources for the ISA
  FILE(GLOB ISA_SRCS "${ISA}/*.c" "${ISA}/*.s")
  FILE(GLOB PLATFORM_SRCS "${ISA}/${PLATFORM}/*.c" "${ISA}/${PLA

  # Load flags associated with ISA and Profile
  INCLUDE("${ISA}/flags.cmake")
  INCLUDE("${ISA}/${PLATFORM}/flags.cmake")

  # Now export our output variables
  SET(ISA_SRCS ${ISA_SRCS} PARENT_SCOPE)
  SET(PLATFORM_SRCS ${PLATFORM_SRCS} PARENT_SCOPE)
  SET(PLATFORM_LAYOUT "${ISA}/${PLATFORM}/layout.ld" PARENT_SCOP

  # And specific flags
```

```
  SET(ISA_C_FLAGS ${ISA_C_FLAGS} PARENT_SCOPE)
  SET(ISA_ASM_FLAGS ${ISA_ASM_FLAGS} PARENT_SCOPE)
  # ...
ENDFUNCTION(LOAD_PROFILE)
```

Now, all we have to do is call `LOAD_PROFILE()` with the provided parameters, and we should be able to set up our build environment in a sane manner:

```
FILE(GLOB GENERIC_SRCS "*.c")

# We could also use CMakeCache variables here!
LOAD_PROFILE("arm" "raspberrypi")


# Now set up our environment
ADD_EXECUTABLE(kernel ${PLATFORM_SRCS} ${ISA_SRCS} ${GENERIC_SRC

SET(CMAKE_ASM-ATT_COMPILE_OBJECT
  "<CMAKE_ASM-ATT_COMPILER> ${ISA_ASM_FLAGS} ${PLATFORM_ASM_FLAG
SET(CMAKE_C_FLAGS "${ISA_C_FLAGS} ${PLATFORM_C_FLAGS}")
SET_TARGET_PROPERTIES(kernel PROPERTIES LINK_FLAGS
  "-T ${PLATFORM_LAYOUT} -N ${ISA_LINKER_FLAGS} ${PLATFORM_LINKE
```

Here, we make a reasonable attempt to control the build order, but the truth is, we don't really know exactly what that order should be; it might be dependent on the platform. For instance, for i386/pc, we might want a multiboot header, which must come within the first 8K of the kernel image. In that case, we must somehow control the ordering. We could have a place `FIRST_SRCS()` variable present in the platform flags, then use the following loop to extract it from the list:

```
FOREACH(I ${FIRST_SRCS})
  # Assume path is relative to src/kernel
  LIST(APPEND TMP_FIRST_SRCS "${CMAKE_CURRENT_LIST_DIR}/${I}"
ENDFOREACH(I)

# Now remove any trace of these files from the other lists
LIST(REMOVE_ITEM ISA_SRCS ${TMP_FIRST_SRCS})
LIST(REMOVE_ITEM PLATFORM_SRCS ${TMP_FIRST_SRCS})

# During exports:
SET(FIRST_SRCS ${TMP_FIRST_SRCS})
```

Now, all we have to do is put `${FIRST_SRCS}` at the head of the list, and we can control the order in which our code is linked.

See Also

Articles

- Makefile - One potential target for CMake. The tried and true method of build management.

External Links

- CMake Offical Page (http://www.cmake.org) - Contains useful links, including how to download and documentation.
- CMake Useful Variables (http://www.cmake.org/Wiki/CMake_Useful_Variables) - Gives names and descriptions of common and useful variables used in CMake.
- CMake Official Documentation (http://www.cmake.org/cmake/help/v3.2) - Official documentation all about CMake.
- CMake Examples (http://www.cmake.org/Wiki/CMake/Examples) - Examples showing how to perform many common CMake procedures.

Categories: Level 1 Tutorials | Tools
