

Dr. Dobb's

JOURNAL

SOFTWARE
TOOLS FOR THE
PROFESSIONAL
PROGRAMMER

Cross-Platform Builds

All it takes is five rules and 42 lines

JOHN GRAHAM-CUMMING

Programmers often find themselves producing cross-platform software, which requires writing and building code that works on Linux, Solaris, HP-UX, AIX, and Windows. And whether you're a professional build manager or simply a programmer charged with maintaining the Makefiles, you know that multiple platforms means multiple Make hassles. In this article, I present a cross-platform pattern for Makefiles that works across Windows and common UNIX platforms, minimizing platform-specific code. At the same time, the pattern completely eliminates recursive use of Make, another hassle that complicates a build manager's life. To simplify the discussion, I've stuck to compiling C code, but there's nothing in the technique that excludes languages such as C++, C#, or Java.

For the purposes of illustration, imag-

ine this layout of directories and source code:

```
src
src/library
src/executable
```

Further imagine a library (lib.a on UNIX, lib.lib on Windows) containing common code built from C files lib1.c and lib2.c:

```
src/library/lib1.c
src/library/lib2.c
```

And finally, imagine the executable (exec on UNIX, exec.exe on Windows) compiled from foo.c and bar.c and linked with the library.

```
src/executable/foo.c
src/executable/bar.c
```

Rule #1: Use GNU Make

The GNU Project's GNU Make (gmake) is powerful, has broad platform support, and has excellent documentation. It simplifies the task of building a cross-platform Make system if you follow the rest of the rules in this article. (Use GNU Make 3.80 or higher since I rely here on some recent features; see <http://www.gnu.org/software/make/>.)

The following is the top-level Makefile stored in src/:

```
MODULES=library executable
include $(addsuffix /Makefile,
$(MODULES))
```

Typing *make* in src/ builds all the object files, library, and executable. The *MODULES* variable is a list of the directories containing components that must

be built. *include \$(addsuffix /Makefile,\$(MODULES))* turns into *include library/Makefile executable/Makefile* and includes the Makefiles that describe each of the separate components.

Rule #2: No Spaces in Filenames, No Drive Letters

GNU Make (in common with other Make programs and scripting languages) treats a space as a list separator. (You can see that in the top-level Makefile where *MODULES* is a list of directory names and *\$(addsuffix)* iterates over that list.) Because of this, it's best to avoid spaces in filenames (for instance, GNU Make sees C:\Program Files as a list with two elements). Either use short filenames or avoid spaces altogether.

Similarly, Make provides special treatment for the colon (:), typically using it as the separator between a target and its prerequisites (foo: foo.o bar.o, and so on), which can lead to confusion if Make sees something like *c:\foo: foo.o bar.o*. ("Is this a rule that defines how to make *c* or *c:\foo*?"). In short, use relative paths.

A good source of GNU Make on Windows is CYGWIN. CYGWIN provides a UNIX-style pathname for drive letters. In a CYGWIN shell, */cygdrive/c* is *C:*. Use CYGWIN paths if you must have access to a specific drive.

The Makefile in the library/ subdirectory consists of five lines:

```
include top.mak
SRCS = lib1.c lib2.c
BINARY = lib
BINARY_EXT=$(X_LIBEXT)
DEPS=
include bottom.mak
```

John is chief scientist at Electric Cloud, which focuses on reducing software build times. He can be contacted at jgc@electric-cloud.com.

Evidently, all the real work is being done by top.mak and bottom.mak.

Rule #3: Decide What Platform the Make is Running On

The simplest way to determine your platform is the *uname* command (available on all flavors of UNIX and on Windows under CYGWIN). The *uname* options *m* and *s* give the processor type and the operating system. For example, on a Linux-based PC, *uname -ms* returns Linux i686. One of the things top.mak does is get the architecture into a variable called “*X_ARCH*”:

```
X_ARCH := $(shell uname -ms |
sed -e s"/ /_g")
```

The *sed* invocation turns any spaces in the output from *uname* into underscores. So, on CYGWIN, *X_ARCH* will be *CYGWIN_NT-5.0_i686*; on Solaris, *SunOS_sun4u*; and on Linux, *Linux_i686*. top.mak begins by determining the machine architecture, setting *X_ARCH*:

```
ifndef X_ALREADYLOADED
.PHONY: all
all:
X_ARCH := $(shell uname -ms |
sed -e s"/ /_g")
endif
```

Notice how the definition of *X_ARCH* uses the GNU Make *:=* operator. This causes the RHS to be evaluated at once and the result placed in *X_ARCH*. If the more familiar *=* operator were used, every reference to *X_ARCH* (and there could be hundreds because of the recursive nature of Make macros) would require a shell invocation to execute *uname* and *sed*, wasting valuable time in the process.

This code also defines a phony target (that is, a target that isn't a file) called *all*. Since this is the first target mentioned, if you just type *make*, it builds the *all* target. *all* is defined using the *::* syntax, which lets you later add other commands and prerequisites as each individual Makefile is parsed. You use this to define *all* for each of the modules.

ifndef X_ALREADYLOADED ensures that this block of code is only parsed once (later in top.mak, you set *X_ALREADYLOADED* to 1).

Rule #4: Separate Output on a Per-Platform Basis

GNU Make does a good job of tracking the dependencies between files, but cross-platform Make and network mounted directories mean that it would be easy to confuse GNU Make. If the object files being created aren't separated on a per-platform basis, an *.o* built on Solaris could

clash with one built on Linux, and so on. top.mak strictly separates the output by platform (and by module) by defining three variables used elsewhere in the system (throughout top.mak and bottom.mak, variables are prefixed by *X_* to stop them from clashing with variables you might want to use in the rest of your system):

- *X_OUTARCH*, the location where objects for this architecture should be placed (by default, this is a subdirectory of *src/*; on PC Linux it would be *src/Linux_i686*).
- *X_MODULE*, the name of the module being built. This is the name of a subdirectory of *src/* (library or executable).
- A specific variable for the module being built giving the location of its object files. For the library module, *library_OUTPUT* is *src/Linux_i686/library* and *executable_OUTPUT* is *src/Linux_i686/executable*.

top.mak starts by setting *X_OUTARCH*, which by default is a subdirectory of the current working directory (*src/*, for example), but can be overridden by setting *X_OUTTOP* on the command-line of GNU Make (or in the environment):

```
ifndef X_ALREADYLOADED
X_OUTTOP ?= .
X_OUTARCH :=
$(X_OUTTOP)/$(X_ARCH)
endif
```

top.mak then sets *X_MODULE*. To do this it uses a GNU Make 3.80 feature—the variable *MAKEFILE_LIST*, which is a (space-separated) list of the Makefiles included so far. By stripping out references to top.mak and bottom.mak, you set the *X_MAKEFILES* variable, the last element of which will be the Makefile, which included top.mak; for instance, the Makefile to one of the modules. The complex definition of *X_MODULE* first takes the last element of *X_MAKEFILES* using *\$(word)* (library/Makefile), gets just the directory portion using *\$(dir)* (library/), and strips the trailing / with *\$(patsubst)*.

```
X_MAKEFILES :=
$(filter-out %.mak,$(MAKEFILE_LIST))
X_MODULE :=
$(patsubst %/,%,$(dir $(word $(words
$(X_MAKEFILES)),$(X_MAKEFILES))))
```

The next step requires a little mind-bending—you define a variable with a computed name. The variable name is *\$(X_MODULE)_OUTPUT* whose name is determined by the value of *X_MODULE* at the time the variable is defined (that is, either library or executable).

```
$(X_MODULE)_OUTPUT :=
$(X_OUTARCH)/$(X_MODULE)
```

This variable contains the name of the directory into which the module's object files will be written. To ensure that the directory exists, you set a dummy variable to the output of a call to *mkdir* using GNU Make's *\$(shell)* command and you use the *:=* operator to perform the command at once. Notice how you use the value of *\$(X_MODULE)_OUTPUT*, which requires writing *\$(\$(X_MODULE)_OUTPUT)*:

```
X_IGNORE :=
$(shell mkdir -p $( $(X_MODULE)_OUTPUT ))
```

Lastly, top.mak defines a clean target that uses the fact that you know that the output for this architecture is in the directory named in *X_OUTARCH*. A single *rm -rf* cleans up all objects:

```
ifndef X_ALREADYLOADED
X_ALREADYLOADED = 1
.PHONY: clean
clean:
@rm -rf $(X_OUTARCH)
.SUFFIXES:
endif
```

You'll notice that the line *.SUFFIXES:* is included in the code, which brings me to the final rule.

Rule #5: Override Built-In Rules

GNU Make (and other Makes) include extensive built-in rules describing how to build objects from various source types (*.o* from *.c*, *.obj* from *.cpp*, and so on). It's best to override them for two reasons:

- Searching all the possible ways of making *X* from *Y* (because the rules can chain together) can be slow.
- You want per-platform control so that you can specify flags, system libraries, and file locations.

Writing *.SUFFIXES:* turns off GNU Make's built-in rules, which brings you to bottom.mak, where the rules are defined. Recall the Makefile in the library/directory:

```
include top.mak
SRCS = lib1.c lib2.c
BINARY = lib
BINARY_EXT=$(X_LIBEXT)
DEPS=
include bottom.mak
```

As GNU Make proceeds through parsing, it includes top.mak (and thus sets *X_MODULE*, *X_ARCH*, and *\$(X_MODULE)_OUTPUT*). Furthermore, the Makefile sets the following:

- *SRCS*, the list of source files used to

build this module.

- *BINARY*, the name of the binary (minus extension) that this module creates.
- *BINARY_EXT*, the extension for the binary. Since this is platform specific, you reference yet another variable *X_LIBEXT*, which gives the platform-specific extension for a library (.a or .lib). *X_LIBEXT* (*X_EXEEXT* for executables) and *X_OBJEXT* (for regular object files) are set by a platform-specific Makefile that *bottom.mak* includes. For example, *bottom.mak* includes *Linux_i686.mak* on PC Linux that specifies (among other things):

```
X_OBJEXT=.o
X_LIBEXT=.a
X_EXEEXT=
```

bottom.mak does the final work of setting up the dependencies for the module and the rules to build the objects, library, or executable. First, it takes the lists of source files and transforms them into a list of object files in the right output directory (module and architecture specific), then it does the same with the binary.

```
include $(X_ARCH).mak
$(X_MODULE)_OBJS =
$(addsuffix $(X_OBJEXT),$(addprefix
$(X_MODULE)_OUTPUT)/,
$(basename $(SRCS))) $(DEPS)
$(X_MODULE)_BINARY =
$(addprefix $(X_MODULE)_OUTPUT)/,
$(BINARY))$(BINARY_EXT)
include $(X_ARCH)-rules.mak
```

Once again, the variable names are computed so that the same macro definitions can be written once and create separate variables for each module: *\$(X_MODULE)_OBJS* (*library_OBJS* for the library module) is the list of object files for the module (*Linux_i686/library/lib1.o* *Linux_i686/library/lib2.o*), and *\$(X_MODULE)_BINARY* (*library_BINARY*) is the

name of the binary (*Linux_i686/library/lib.a*).

Then you specify a couple of rules. The first says that the *all* target builds the binary. The second says that, if users specify the name of the module, then it also builds the binary. This means that *all* will build every binary, although it's possible to use *make library* to build the library module.

```
all: $(X_MODULE)_BINARY
$(X_MODULE): $(X_MODULE)_BINARY
```

You'll notice that *bottom.mak* included a platform-specific Makefile with the directives *include \$(X_ARCH).mak* and *include \$(X_ARCH)-rules.mak*, which contain all the platform code you need. Here's *Linux_i686.mak*:

```
X_OBJEXT=.o
X_LIBEXT=.a
X_EXEEXT=
```

and here's *Linux_i686-rules.mak*:

```
$(X_MODULE)_OUTPUT)/%.o:
$(X_MODULE)/%.c
@$(COMPILE.c) -o '$@' '$<'
$(X_MODULE)_OUTPUT)/$(BINARY).a:
$(X_MODULE)_OBJS
@$(AR) r '$@' '$^'
@ranlib '$@'
$(X_MODULE)_OUTPUT)/
$(BINARY)$(X_EXEEXT):
$(X_MODULE)_OBJS
@$(LINK.c) $^ -o'$@'
```

These two files first specify the extensions that are valid for the platform and then define rules for building an *.o* from a *.c*, a *.a* from the list of objects for a module, and an executable from a list of objects. Notice how each rule uses *\$(X_MODULE)_OUTPUT* for the architecture- and module-specific output location for binaries. The rules are using GNU Make built-in variables *COMPILE.c* (how to compile a C file), *AR* (*ar*), and *LINK.c* (how to link).

If necessary, these can be overridden to have platform-specific flags and other options.

The only change needed to build the same code on another platform is the creation of the appropriate equivalents of *Linux_i686.mak* and *Linux_i686-rules.mak*. For example, under Windows the extensions for objects, libraries, and executables are *.obj*, *.lib*, and *.exe* and the rules can be redefined to use Microsoft's *cl* compiler (and done so without making any changes to the Makefile in the module directories or *top.mak* and *bottom.mak*).

If you're a Make guru, you're probably ready to point out that the executable needs to link against the library and needs to depend on the library itself. Here's the Makefile used in the executable/directory:

```
include top.mak
SRCS = foo.c bar.c
BINARY = exec
BINARY_EXT=$(X_EXEEXT)
DEPS = $(library_BINARY)
include bottom.mak
```

The interesting line is *DEPS=\$(library_BINARY)*, which specifies that this module depends on the binary file generated by the library module. Because the system has set up separate variables for each module, the executable module can refer directly to the binary or objects created by another module. All that matters is that the Makefile for library was included before the Makefile for executable. Put together, *top.mak*, *bottom.mak*, and the platform-specific Makefiles consist of 42 lines of Make syntax that gives a powerful, flexible cross-platform Make system that is fast and eliminates the problems inherent in recursive Makes.

DDJ

