PART II

# Advanced and Specialized Topics

In Part II, we take a problem-oriented view of make. It is often not obvious how to apply make to real-world problems such as multidirectory builds, new programming languages, portability and performance issues, or debugging. Each of these problems is discussed, along with a chapter covering several complex examples.

CHAPTER 6

# Managing Large Projects

What do you call a large project? For our purposes, it is one that requires a team of developers, may run on multiple architectures, and may have several field releases that require maintenance. Of course, not all of these are required to call a project large. A million lines of prerelease C++ on a single platform is still large. But software rarely stays prerelease forever. And if it is successful, someone will eventually ask for it on another platform. So most large software systems wind up looking very similar after awhile.

Large software projects are usually simplified by dividing them into major components, often collected into distinct programs, libraries, or both. These components are often stored under their own directories and managed by their own *makefile*s. One way to build an entire system of components employs a top-level *makefile* that invokes the *makefile* for each component in the proper order. This approach is called *recursive make* because the top-level *makefile* invokes `make` recursively on each component's *makefile*. Recursive `make` is a common technique for handling componentwise builds. An alternative suggested by Peter Miller in 1998 avoids many issues with recursive `make` by using a single *makefile* that includes information from each component directory.*

Once a project gets beyond building its components, it eventually finds that there are larger organizational issues in managing builds. These include handling development on multiple versions of a project, supporting several platforms, providing efficient access to source and binaries, and performing automated builds. We will discuss these problems in the second half of this chapter.

* Miller, P.A., *Recursive Make Considered Harmful*, AUUGN Journal of AUUG Inc., 19(1), pp. 14–25 (1998). Also available from *http://aegis.sourceforge.net/auug97.pdf*.

## Recursive make

The motivation behind recursive make is simple: make works very well within a single directory (or small set of directories) but becomes more complex when the number of directories grows. So, we can use make to build a large project by writing a simple, self-contained *makefile* for each directory, then executing them all individually. We could use a scripting tool to perform this execution, but it is more effective to use make itself since there are also dependencies involved at the higher level.

For example, suppose I have an mp3 player application. It can logically be divided into several components: the user interface, codecs, and database management. These might be represented by three libraries: *libui.a*, *libcodec.a*, and *libdb.a*. The application itself consists of glue holding these pieces together. A straightforward mapping of these components onto a file structure might look like Figure 6-1.

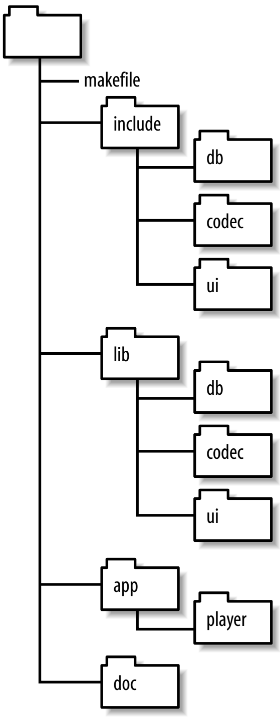

*Figure 6-1. File layout for an MP3 player*

A more traditional layout would place the application's main function and glue in the top directory rather than in the subdirectory *app/player*. I prefer to put application code in its own directory to create a cleaner layout at the top level and allow for growth of the system with additional modules. For instance, if we choose to add a separate cataloging application later it can neatly fit under *app/catalog*.

If each of the directories *lib/db*, *lib/codec*, *lib/ui*, and *app/player* contains a *makefile*, then it is the job of the top-level *makefile* to invoke them.

```
lib_codec := lib/codec
lib_db    := lib/db
lib_ui    := lib/ui
libraries := $(lib_ui) $(lib_db) $(lib_codec)
player    := app/player

.PHONY: all $(player) $(libraries)
all: $(player)

$(player) $(libraries):
        $(MAKE) --directory=$@

$(player): $(libraries)
$(lib_ui): $(lib_db) $(lib_codec)
```

The top-level *makefile* invokes `make` on each subdirectory through a rule that lists the subdirectories as targets and whose action is to invoke `make`:

```
$(player) $(libraries):
        $(MAKE) --directory=$@
```

The variable `MAKE` should always be used to invoke `make` within a *makefile*. The `MAKE` variable is recognized by `make` and is set to the actual path of `make` so recursive invocations all use the same executable. Also, lines containing the variable `MAKE` are handled specially when the command-line options `--touch` (`-t`), `--just-print` (`-n`), and `--question` (`-q`) are used. We'll discuss this in detail in the section "Command-Line Options" later in this chapter.

The target directories are marked with `.PHONY` so the rule fires even though the target may be up to date. The `--directory` (`-C`) option is used to cause `make` to change to the target directory before reading a *makefile*.

This rule, although a bit subtle, overcomes several problems associated with a more straightforward command script:

```
all:
        for d in $(player) $(libraries); \
        do                               \
          $(MAKE) --directory=$$d;       \
        done
```

This command script fails to properly transmit errors to the parent `make`. It also does not allow `make` to execute any subdirectory builds in parallel. We'll discuss this feature of `make` in Chapter 10.

As `make` is planning the execution of the dependency graph, the prerequisites of a target are independent of one another. In addition, separate targets with no dependency relationships to one another are also independent. For example, the libraries have no *inherent* relationship to the app/player target or to each other. This means make is free to execute the *app/player makefile* before building any of the libraries. Clearly, this would cause the build to fail since linking the application requires the libraries. To solve this problem, we provide additional dependency information.

```
$(player): $(libraries)
$(lib_ui): $(lib_db) $(lib_codec)
```

Here we state that the *makefile*s in the library subdirectories must be executed before the *makefile* in the player directory. Similarly, the *lib/ui* code requires the *lib/db* and *lib/codec* libraries to be compiled. This ensures that any generated code (such as yacc/lex files) have been generated before the *ui* code is compiled.

There is a further subtle ordering issue when updating prerequisites. As with all dependencies, the order of updating is determined by the analysis of the dependency graph, but when the prerequisites of a target are listed on a single line, GNU make happens to update them from left to right. For example:

```
all: a b c
all: d e f
```

If there are no other dependency relationships to be considered, the six prerequisites can be updated in any order (e.g., “d b a c e f”), but GNU make uses left to right within a single target line, yielding the update order: “a b c d e f” *or* “d e f a b c.” Although this ordering is an accident of the implementation, the order of execution appears correct. It is easy to forget that the correct order is a happy accident and fail to provide full dependency information. Eventually, the dependency analysis will yield a different order and cause problems. So, if a set of targets must be updated in a specific order, enforce the proper order with appropriate prerequisites.

When the top-level *makefile* is run, we see:

```
$ make
make --directory=lib/db
make[1]: Entering directory `/test/book/out/ch06-simple/lib/db'
Update db library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/db'
make --directory=lib/codec
make[1]: Entering directory `/test/book/out/ch06-simple/lib/codec'
Update codec library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/codec'
make --directory=lib/ui
make[1]: Entering directory `/test/book/out/ch06-simple/lib/ui'
Update ui library...
make[1]: Leaving directory `/test/book/out/ch06-simple/lib/ui'
make --directory=app/player
make[1]: Entering directory `/test/book/out/ch06-simple/app/player'
Update player application...
make[1]: Leaving directory `/test/book/out/ch06-simple/app/player'
```

When make detects that it is invoking another make recursively, it enables the --print-directory (-w) option, which causes make to print the Entering directory and Leaving directory messages. This option is also enabled when the --directory (-C) option is used. The value of the make variable MAKELEVEL is printed in square brackets in each line as well. In this simple example, each component *makefile* prints a simple message about updating the component.

## Command-Line Options

Recursive make is a simple idea that quickly becomes complicated. The perfect recursive make implementation would behave as if the many *makefile*s in the system are a single *makefile*. Achieving this level of coordination is virtually impossible, so compromises must be made. The subtle issues become more clear when we look at how command-line options must be handled.

Suppose we have added comments to a header file in our mp3 player. Rather than recompiling all the source that depends on the modified header, we realize we can instead perform a make --touch to bring the timestamps of the files up to date. By executing the make --touch with the top-level *makefile,* we would like make to touch all the appropriate files managed by sub-makes. Let's see how this works.

Usually, when --touch is provided on the command line, the normal processing of rules is suspended. Instead, the dependency graph is traversed and the selected targets and those prerequisites that are not marked .PHONY are brought up to date by executing touch on the target. Since our subdirectories are marked .PHONY, they would normally be ignored (touching them like normal files would be pointless). But we don't want those targets ignored, we want their command script executed. To do the right thing, make automatically labels any line containing MAKE with the + modifier, meaning make runs the sub-make regardless of the --touch option.

When make runs the sub-make it must also arrange for the --touch flag to be passed to the sub-process. It does this through the MAKEFLAGS variable. When make starts, it automatically appends most command-line options to MAKEFLAGS. The only exceptions are the options --directory (-C), --file (-f), --old-file (-o), and --new-file (-W). The MAKEFLAGS variable is then exported to the environment and read by the sub-make as it starts.

With this special support, sub-makes behave mostly the way you want. The recursive execution of $(MAKE) and the special handling of MAKEFLAGS that is applied to --touch (-t) is also applied to the options --just-print (-n) and --question (-q).

## Passing Variables

As we have already mentioned, variables are passed to sub-makes through the environment and controlled using the export and unexport directives. Variables passed through the environment are taken as default values, but are overridden by any assignment to the variable. Use the `--environment-overrides` (`-e`) option to allow environment variables to override the local assignment. You can explicitly override the environment for a specific assignment (even when the `--environment-overrides` option is used) with the `override` directive:

```
override TMPDIR = ~/tmp
```

Variables defined on the command line are automatically exported to the environment if they use legal shell syntax. A variable is considered legal if it uses only letters, numbers, and underscores. Variable assignments from the command line are stored in the `MAKEFLAGS` variable along with command-line options.

## Error Handling

What happens when a recursive `make` gets an error? Nothing very unusual, actually. The `make` receiving the error status terminates its processing with an exit status of 2. The parent `make` then exits, propagating the error status up the recursive `make` process tree. If the `--keep-going` (`-k`) option is used on the top-level `make`, it is passed to sub-`makes` as usual. The sub-`make` does what it normally does, skips the current target and proceeds to the next goal that does not use the erroneous target as a prerequisite.

For example, if our mp3 player program encountered a compilation error in the `lib/db` component, the `lib/db make` would exit, returning a status of 2 to the top-level *makefile*. If we used the `--keep-going` (`-k`) option, the top-level *makefile* would proceed to the next unrelated target, `lib/codec`. When it had completed that target, regardless of its exit status, the `make` would exit with a status of 2 since there are no further targets that can be processed due to the failure of `lib/db`.

The `--question` (`-q`) option behaves very similarly. This option causes `make` to return an exit status of 1 if some target is not up to date, 0 otherwise. When applied to a tree of *makefile*s, `make` begins recursively executing *makefile*s until it can determine if the project is up to date. As soon as an out-of-date file is found, `make` terminates the currently active `make` and unwinds the recursion.

## Building Other Targets

The basic build target is essential for any build system, but we also need the other support targets we've come to depend upon, such as `clean`, `install`, `print`, etc. Because these are `.PHONY` targets, the technique described earlier doesn't work very well.

For instance, there are several broken approaches, such as:

```
clean: $(player) $(libraries)
        $(MAKE) --directory=$@ clean
```

or:

```
$(player) $(libraries):
        $(MAKE) --directory=$@ clean
```

The first is broken because the prerequisites would trigger a build of the default target in the `$(player)` and `$(libraries)` *makefile*s, not a build of the `clean` target. The second is illegal because these targets already exist with a different command script.

One approach that works relies on a shell `for` loop:

```
clean:
        for d in $(player) $(libraries); \
        do                              \
          $(MAKE) --directory=$$f clean; \
        done
```

A `for` loop is not very satisfying for all the reasons described earlier, but it (and the preceding illegal example) points us to this solution:

```
$(player) $(libraries):
        $(MAKE) --directory=$@ $(TARGET)
```

By adding the variable `$(TARGET)` to the recursive `make` line and setting the `TARGET` variable on the `make` command line, we can add arbitrary goals to the sub-`make`:

```
$ make TARGET=clean
```

Unfortunately, this does not invoke the `$(TARGET)` on the top-level *makefile*. Often this is not necessary because the top-level *makefile* has nothing to do, but, if necessary, we can add another invocation of `make` protected by an `if`:

```
$(player) $(libraries):
        $(MAKE) --directory=$@ $(TARGET)
        $(if $(TARGET), $(MAKE) $(TARGET))
```

Now we can invoke the `clean` target (or any other target) by simply setting `TARGET` on the command line.

## Cross-Makefile Dependencies

The special support in `make` for command-line options and communication through environment variables suggests that recursive `make` has been tuned to work well. So what are the serious complications alluded to earlier?

Separate *makefile*s linked by recursive `$(MAKE)` commands record only the most superficial top-level links. Unfortunately, there are often subtle dependencies buried in some directories.

For example, suppose a *db* module includes a `yacc`-based parser for importing and exporting music data. If the *ui* module, *ui.c*, includes the generated `yacc` header, we have a dependency between these two modules. If the dependencies are properly modeled, `make` should know to recompile our *ui* module whenever the grammar header is updated. This is not difficult to arrange using the automatic dependency generation technique described earlier. But what if the `yacc` file itself is modified? In this case, when the *ui makefile* is run, a correct *makefile* would recognize that `yacc` must first be run to generate the parser and header before compiling *ui.c*. In our recursive make decomposition, this does not occur, because the rule and dependencies for running yacc are in the *db makefile*, not the *ui makefile*.

In this case, the best we can do is to ensure that the *db makefile* is always executed before executing the *ui makefile*. This higher-level dependency must be encoded by hand. We were astute enough in the first version of our *makefile* to recognize this, but, in general, this is a very difficult maintenance problem. As code is written and modified, the top-level *makefile* will fail to properly record the intermodule dependencies.

To continue the example, if the yacc grammar in *db* is updated and the *ui makefile* is run before the *db makefile* (by executing it directly instead of through the top-level *makefile*), the *ui makefile* does not know there is an unsatisfied dependency in the *db makefile* and that yacc must be run to update the header file. Instead, the *ui makefile* compiles its program with the old yacc header. If new symbols have been defined and are now being referenced, then a compilation error is reported. Thus, the recursive make approach is inherently more fragile than a single *makefile*.

The problem worsens when code generators are used more extensively. Suppose that the use of an RPC stub generator is added to *ui* and the headers are referenced in *db*. Now we have mutual reference to contend with. To resolve this, it may be required to visit *db* to generate the yacc header, then visit *ui* to generate the RPC stubs, then visit *db* to compile the files, and finally visit *ui* to complete the compilation process. The number of passes required to create and compile the source for a project is dependent on the structure of the code and the tools used to create it. This kind of mutual reference is common in complex systems.

The standard solution in real-world *makefile*s is usually a hack. To ensure that all files are up to date, every *makefile* is executed when a command is given to the top-level *makefile*. Notice that this is precisely what our mp3 player *makefile* does. When the top-level *makefile* is run, each of the four sub-*makefile*s is unconditionally run. In complex cases, *makefile*s are run repeatedly to ensure that all code is first generated then compiled. Often this iterative execution is a complete waste of time, but occasionally it is required.

## Avoiding Duplicate Code

The directory layout of our application includes three libraries. The *makefile*s for these libraries are very similar. This makes sense because the three libraries serve different purposes in the final application but are all built with similar commands. This kind of decomposition is typical of large projects and leads to many similar *makefile*s and lots of (*makefile*) code duplication.

Code duplication is bad, even *makefile* code duplication. It increases the maintenance costs of the software and leads to more bugs. It also makes it more difficult to understand algorithms and identify minor variations in them. So we would like to avoid code duplication in our *makefile*s as much as possible. This is most easily accomplished by moving the common pieces of a *makefile* into a common include file.

For example, the *codec makefile* contains:

```
lib_codec    := libcodec.a
sources      := codec.c
objects      := $(subst .c,.o,$(sources))
dependencies := $(subst .c,.d,$(sources))

include_dirs := .. ../../include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

all: $(lib_codec)

$(lib_codec): $(objects)
        $(AR) $(ARFLAGS) $@ $^

.PHONY: clean
clean:
        $(RM) $(lib_codec) $(objects) $(dependencies)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< |      \
        sed 's,\($*\.o\) *:,\1 $@: ,' > $@.tmp
        mv $@.tmp $@
```

Almost all of this code is duplicated in the *db* and *ui makefile*s. The only lines that change for each library are the name of the library itself and the source files the library contains. When duplicate code is moved into *common.mk*, we can pare this *makefile* down to:

```
library := libcodec.a
sources := codec.c

include ../../common.mk
```

See what we have moved into the single, shared include file:

```
MV           := mv -f
RM           := rm -f
SED          := sed

objects      := $(subst .c,.o,$(sources))
dependencies := $(subst .c,.d,$(sources))
include_dirs := .. ../../include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))

vpath %.h $(include_dirs)
```

```
.PHONY: library
library: $(library)

$(library): $(objects)
	$(AR) $(ARFLAGS) $@ $^

.PHONY: clean
clean:
	$(RM) $(objects) $(program) $(library) $(dependencies) $(extra_clean)

ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

%.c %.h: %.y
	$(YACC.y) --defines $<
	$(MV) y.tab.c $*.c
	$(MV) y.tab.h $*.h

%.d: %.c
	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< |      \
	$(SED) 's,\($*\.o\) *:,\1 $@: ,' > $@.tmp
	$(MV) $@.tmp $@
```

The variable `include_dirs`, which was different for each *makefile*, is now identical in all *makefile*s because we reworked the path source files use for included headers to make all libraries use the same include path.

The *common.mk* file even includes the default goal for the library include files. The original *makefile*s used the default target `all`. That would cause problems with nonlibrary *makefile*s that need to specify a different set of prerequisites for their default goal. So the shared code version uses a default target of `library`.

Notice that because this common file contains targets it must be included after the default target for nonlibrary *makefile*s. Also notice that the `clean` command script references the variables `program`, `library`, and `extra_clean`. For library *makefile*s, the `program` variable is empty; for program *makefile*s, the `library` variable is empty. The `extra_clean` variable was added specifically for the *db makefile*. This *makefile* uses the variable to denote code generated by `yacc`. The *makefile* is:

```
library     := libdb.a
sources     := scanner.c playlist.c
extra_clean := $(sources) playlist.h

.SECONDARY: playlist.c playlist.h scanner.c

include ../../common.mk
```

Using these techniques, code duplication can be kept to a minimum. As more *makefile* code is moved into the common *makefile*, it evolves into a generic *makefile* for the entire project. `make` variables and user-defined functions are used as customization points, allowing the generic *makefile* to be modified for each directory.

## Nonrecursive make

Multidirectory projects can also be managed without recursive makes. The difference here is that the source manipulated by the *makefile* lives in more than one directory. To accommodate this, references to files in subdirectories must include the path to the file—either absolute or relative.

Often, the *makefile* managing a large project has many targets, one for each module in the project. For our mp3 player example, we would need targets for each of the libraries and each of the applications. It can also be useful to add phony targets for collections of modules such as the collection of all libraries. The default goal would typically build all of these targets. Often the default goal builds documentation and runs a testing procedure as well.

The most straightforward use of nonrecursive make includes targets, object file references, and dependencies in a single *makefile*. This is often unsatisfying to developers familiar with recursive make because information about the files in a directory is centralized in a single file while the source files themselves are distributed in the filesystem. To address this issue, the Miller paper on nonrecursive make suggests using one make include file for each directory containing file lists and module-specific rules. The top-level *makefile* includes these sub-*makefile*s.

Example 6-1 shows a *makefile* for our mp3 player that includes a module-level *makefile* from each subdirectory. Example 6-2 shows one of the module-level include files.

*Example 6-1. A nonrecursive makefile*

```
# Collect information from each module in these four variables.
# Initialize them here as simple variables.
programs     :=
sources      :=
libraries    :=
extra_clean  :=

objects      = $(subst .c,.o,$(sources))
dependencies = $(subst .c,.d,$(sources))

include_dirs := lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MV  := mv -f
RM  := rm -f
SED := sed

all:

include lib/codec/module.mk
include lib/db/module.mk
```

*Example 6-1. A nonrecursive makefile (continued)*

```
include lib/ui/module.mk
include app/player/module.mk

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
	$(RM) $(objects) $(programs) $(libraries) \
	      $(dependencies) $(extra_clean)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.c %.h: %.y
	$(YACC.y) --defines $<
	$(MV) y.tab.c $*.c
	$(MV) y.tab.h $*.h

%.d: %.c
	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
	$(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,' > $@.tmp
	$(MV) $@.tmp $@
```

*Example 6-2. The lib/codec include file for a nonrecursive makefile*

```
local_dir  := lib/codec
local_lib  := $(local_dir)/libcodec.a
local_src  := $(addprefix $(local_dir)/,codec.c)
local_objs := $(subst .c,.o,$(local_src))

libraries  += $(local_lib)
sources    += $(local_src)

$(local_lib): $(local_objs)
	$(AR) $(ARFLAGS) $@ $^
```

Thus, all the information specific to a module is contained in an include file in the module directory itself. The top-level *makefile* contains only a list of modules and `include` directives. Let's examine the *makefile* and *module.mk* in detail.

Each *module.mk* include file appends the local library name to the variable `libraries` and the local sources to `sources`. The `local_` variables are used to hold constant values or to avoid duplicating a computed value. Note that each include file reuses these same `local_` variable names. Therefore, it uses simple variables (those assigned with `:=`) rather than recursive ones so that builds combining multiple *makefile*s hold no risk of infecting the variables in each *makefile*. The library name and source file lists use a relative path as discussed earlier. Finally, the include file defines a rule for updating the local library. There is no problem with using the local_ variables in this rule because the target and prerequisite parts of a rule are immediately evaluated.

In the top-level *makefile*, the first four lines define the variables that accumulate each module's specific file information. These variables must be simple variables because each module will append to them using the same local variable name:

```
local_src  := $(addprefix $(local_dir)/,codec.c)
…
sources    += $(local_src)
```

If a recursive variable were used for sources, for instance, the final value would simply be the last value of local_src repeated over and over. An explicit assignment is required to initialize these simple variables, even though they are assigned null values, since variables are recursive by default.

The next section computes the object file list, objects, and dependency file list from the sources variable. These variables are recursive because at this point in the *makefile* the sources variable is empty. It will not be populated until later when the include files are read. In this *makefile*, it is perfectly reasonable to move the definition of these variables after the includes and change their type to simple variables, but keeping the basic file lists (e.g., sources, libraries, objects) together simplifies understanding the *makefile* and is generally good practice. Also, in other *makefile* situations, mutual references between variables require the use of recursive variables.

Next, we handle C language include files by setting CPPFLAGS. This allows the compiler to find the headers. We append to the CPPFLAGS variable because we don't know if the variable is really empty; command-line options, environment variables, or other make constructs may have set it. The vpath directive allows make to find the headers stored in other directories. The include_dirs variable is used to avoid duplicating the include directory list.

Variables for mv, rm, and sed are defined to avoid hard coding programs into the *makefile*. Notice the case of variables. We are following the conventions suggested in the make manual. Variables that are internal to the *makefile* are lowercased; variables that might be set from the command line are uppercased.

In the next section of the *makefile,* things get more interesting. We would like to begin the explicit rules with the default target, all. Unfortunately, the prerequisite for all is the variable programs. This variable is evaluated immediately, but is set by reading the module include files. So, we must read the include files before the all target is defined. Unfortunately again, the include modules contain targets, the first of which will be considered the default goal. To work through this dilemma, we can specify the all target with no prerequisites, source the include files, then add the prerequisites to all later.

The remainder of the *makefile* is already familiar from previous examples, but how make applies implicit rules is worth noting. Our source files now reside in subdirectories. When make tries to apply the standard %.o: %.c rule, the prerequisite will be a file with a relative path, say *lib/ui/ui.c*. make will automatically propagate that relative path to the target file and attempt to update *lib/ui/ui.o*. Thus, make automagically does the Right Thing.

There is one final glitch. Although make is handling paths correctly, not all the tools used by the *makefile* are. In particular, when using gcc, the generated dependency file does not include the relative path to the target object file. That is, the output of gcc -M is:

```
ui.o: lib/ui/ui.c include/ui/ui.h lib/db/playlist.h
```

rather than what we expect:

```
lib/ui/ui.o: lib/ui/ui.c include/ui/ui.h lib/db/playlist.h
```

This disrupts the handling of header file prerequisites. To fix this problem we can alter the sed command to add relative path information:

```
$(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,'
```

Tweaking the *makefile* to handle the quirks of various tools is a normal part of using make. Portable *makefile*s are often very complex due to vagarities of the diverse set of tools they are forced to rely upon.

We now have a decent nonrecursive *makefile*, but there are maintenance problems. The *module.mk* include files are largely similar. A change to one will likely involve a change to all of them. For small projects like our mp3 player it is annoying. For large projects with several hundred include files it can be fatal. By using consistent variable names and regularizing the contents of the include files, we position ourselves nicely to cure these ills. Here is the *lib/codec* include file after refactoring:

```
local_src := $(wildcard $(subdirectory)/*.c)

$(eval $(call make-library, $(subdirectory)/libcodec.a, $(local_src)))
```

Instead of specifying source files by name, we assume we want to rebuild all *.c* files in the directory. The make-library function now performs the bulk of the tasks for an include file. This function is defined at the top of our project *makefile* as:

```
# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
    $(AR) $(ARFLAGS) $$@ $$^
endef
```

The function appends the library and sources to their respective variables, then defines the explicit rule to build the library. Notice how the automatic variables use two dollar signs to defer actual evaluation of the $@ and $^ until the rule is fired. The source-to-object function translates a list of source files to their corresponding object files:

```
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))
```

In our previous version of the *makefile*, we glossed over the fact that the actual parser and scanner source files are *playlist.y* and *scanner.l*. Instead, we listed the source files as the generated *.c* versions. This forced us to list them explicitly and to include an extra variable, extra_clean. We've fixed that issue here by allowing the sources variable to include *.y* and *.l* files directly and letting the source-to-object function do the work of translating them.

In addition to modifying source-to-object, we need another function to compute the yacc and lex output files so the clean target can perform proper clean up. The generated-source function simply accepts a list of sources and produces a list of intermediate files as output:

```
# $(call generated-source, source-file-list)
generated-source = $(subst .y,.c,$(filter %.y,$1)) \
                   $(subst .y,.h,$(filter %.y,$1)) \
                   $(subst .l,.c,$(filter %.l,$1))
```

Our other helper function, subdirectory, allows us to omit the variable local_dir.

```
subdirectory = $(patsubst %/makefile,%,                        \
                 $(word                                        \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))
```

As noted in the section "String Functions" in Chapter 4, we can retrieve the name of the current *makefile* from MAKEFILE_LIST. Using a simple patsubst, we can extract the relative path from the top-level *makefile*. This eliminates another variable and reduces the differences between include files.

Our final optimization (at least for this example), uses wildcard to acquire the source file list. This works well in most environments where the source tree is kept clean. However, I have worked on projects where this is not the case. Old code was kept in the source tree "just in case." This entailed real costs in terms of programmer time and anguish since old, dead code was maintained when it was found by global search and replace and new programmers (or old ones not familiar with a module) attempted to compile or debug code that was never used. If you are using a modern source code control system, such as CVS, keeping dead code in the source tree is unnecessary (since it resides in the repository) and using wildcard becomes feasible.

The include directives can also be optimzed:

```
modules := lib/codec lib/db lib/ui app/player
...
include $(addsuffix /module.mk,$(modules))
```

For larger projects, even this can be a maintenance problem as the list of modules grows to the hundreds or thousands. Under these circumstances, it might be preferable to define modules as a find command:

```
modules := $(subst /module.mk,,$(shell find . -name module.mk))
...
include $(addsuffix /module.mk,$(modules))
```

We strip the filename from the find output so the modules variable is more generally useful as the list of modules. If that isn't necessary, then, of course, we would omit the subst and addsuffix and simply save the output of find in modules. Example 6-3 shows the final *makefile*.

*Example 6-3. A nonrecursive makefile, version 2*

```
# $(call source-to-object, source-file-list)
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))

# $(subdirectory)
subdirectory = $(patsubst %/module.mk,%,                        \
                 $(word                                         \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))

# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
	$(AR) $(ARFLAGS) $$@ $$^
endef

# $(call generated-source, source-file-list)
generated-source = $(subst .y,.c,$(filter %.y,$1))      \
                   $(subst .y,.h,$(filter %.y,$1))      \
                   $(subst .l,.c,$(filter %.l,$1))

# Collect information from each module in these four variables.
# Initialize them here as simple variables.
modules      := lib/codec lib/db lib/ui app/player
programs     :=
libraries    :=
sources      :=

objects      =  $(call source-to-object,$(sources))
dependencies =  $(subst .o,.d,$(objects))

include_dirs := lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)
```

*Example 6-3. A nonrecursive makefile, version 2 (continued)*

```
MV  := mv -f
RM  := rm -f
SED := sed

all:

include $(addsuffix /module.mk,$(modules))

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
        $(RM) $(objects) $(programs) $(libraries) $(dependencies)       \
              $(call generated-source, $(sources))

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%.d: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
        $(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,' > $@.tmp
        $(MV) $@.tmp $@
```

Using one include file per module is quite workable and has some advantages, but I'm not convinced it is worth doing. My own experience with a large Java project indicates that a single top-level *makefile*, effectively inserting all the *module.mk* files directly into the *makefile*, provides a reasonable solution. This project included 997 separate modules, about two dozen libraries, and half a dozen applications. There were several *makefile*s for disjoint sets of code. These *makefile*s were roughly 2,500 lines long. A common include file containing global variables, user-defined functions, and pattern rules was another 2,500 lines.

Whether you choose a single *makefile* or break out module information into include files, the nonrecursive make solution is a viable approach to building large projects. It also solves many traditional problems found in the recursive make approach. The only drawback I'm aware of is the paradigm shift required for developers used to recursive make.

## Components of Large Systems

For the purposes of this discussion, there are two styles of development popular today: the free software model and the commercial development model.

In the free software model, each developer is largely on his own. A project has a *makefile* and a *README* and developers are expected to figure it out with only a small amount of help. The principals of the project want things to work well and want to receive contributions from a large community, but they are mostly interested in contributions from the skilled and well-motivated. This is not a criticism. In this point of view, software should be written well, and not necessarily to a schedule.

In the commercial development model, developers come in a wide variety of skill levels and all of them must be able to develop software to contribute to the bottom line. Any developer who can't figure out how to do their job is wasting money. If the system doesn't compile or run properly, the development team as a whole may be idle, the most expensive possible scenario. To handle these issues, the development process is managed by an engineering support team that coordinates the build process, configuration of software tools, coordination of new development and maintenance work, and the management of releases. In this environment, efficiency concerns dominate the process.

It is the commercial development model that tends to create elaborate build systems. The primary reason for this is pressure to reduce the cost of software development by increasing programmer efficiency. This, in turn, should lead to increased profit. It is this model that requires the most support from `make`. Nevertheless, the techniques we discuss here apply to the free software model as well when their requirements demand it.

This section contains a lot of high-level information with very few specifics and no examples. That's because so much depends on the language and operating environment used. In Chapters 8 and 9, I will provide specific examples of how to implement many of these features.

### Requirements

Of course requirements vary with every project and every work environment. Here we cover a wide range that are often considered important in many commercial development environments.

The most common feature desired by development teams is the separation of source code from binary code. That is, the object files generated from a compile should be placed in a separate binary tree. This, in turn, allows many other features to be added. Separate binary trees offer many advantages:

- It is easier to manage disk resources when the location of large binary trees can be specified.
- Many versions of a binary tree can be managed in parallel. For instance, a single source tree may have optimized, debug, and profiling binary versions available.
- Multiple platforms can be supported simultaneously. A properly implemented source tree can be used to compile binaries for many platforms in parallel.
- Developers can check out partial source trees and have the build system automatically "fill in" the missing files from a reference source and binary trees. This doesn't strictly require separating source and binary, but without the separation it is more likely that developer build systems would get confused about where binaries should be found.
- Source trees can be protected with read-only access. This provides added assurance that the builds reflect the source code in the repository.
- Some targets, such as `clean`, can be implemented trivially (and will execute dramatically faster) if a tree can be treated as a single unit rather than searching the tree for files to operate on.

Most of the above points are themselves important build features and may be project requirements.

Being able to maintain reference builds of a project is often an important system feature. The idea is that a clean check-out and build of the source is performed nightly, typically by a `cron` job. Since the resulting source and binary trees are unmodified with respect to the CVS source, I refer to these as reference source and binary trees. The resulting trees have many uses.

First, a reference source tree can be used by programmers and managers who need to look at the source. This may seem trivial, but when the number of files and releases grows it can be unwieldy or unreasonable to expect someone to check-out the source just to examine a single file. Also, while CVS repository browsing tools are common, they do not typically provide for easy searching of the entire source tree. For this, tags tables or even `find`/`grep` (or `grep -R`) are more appropriate.

Second, and most importantly, a reference binary tree indicates that the source builds cleanly. When developers begin each morning, they know if the system is broken or whole. If a batch-oriented testing framework is in place, the clean build can be used to run automated tests. Each day developers can examine the test report to determine the health of the system without wasting time running the tests themselves. The cost savings is compounded if a developer has only a modified version of the source because he avoids spending additional time performing a clean check-out and build. Finally, the reference build can be run by developers to test and compare the functionality of specific components.

The reference build can be used in other ways as well. For projects that consist of many libraries, the precompiled libraries from the nightly build can be used by programmers to link their own application with those libraries they are not modifying. This allows them to shorten their develoment cycle by omiting large portions of the source tree from their local compiles. Of course, easy access to the project source on a local file server is convenient if developers need to examine the code and do not have a complete checked out source tree.

With so many different uses, it becomes more important to verify the integrity of the reference source and binary trees. One simple and effective way to improve reliability is to make the source tree read-only. Thus, it is guaranteed that the reference source files accurately reflect the state of the repository at the time of check out. Doing this can require special care, because many different aspects of the build may attempt to causally write to the source tree. Especially when generating source code or writing temporary files. Making the source tree read-only also prevents casual users from accidentally corrupting the source tree, a most common occurrence.

Another common requirement of the project build system is the ability to easily handle different compilation, linking, and deployment configurations. The build system typically must be able to manage different versions of the project (which may be branches of the source repository).

Most large projects rely on significant third-party software, either in the form of linkable libraries or tools. If there are no other tools to manage configurations of the software (and often there are not), using the *makefile* and build system to manage this is often a reasonable choice.

Finally, when software is released to a customer, it is often repackaged from its development form. This can be as complex as constructing a *setup.exe* file for Windows or as simple as formatting an HTML file and bundling it with a jar. Sometimes this installer build operation is combined with the normal build process. I prefer to keep the build and the install generation as two separate stages because they seem to use radically different processes. In any case, it is likely that both of these operations will have an impact on the build system.

## Filesystem Layout

Once you choose to support fmultiple binary trees, the question of filesystem layout arises. In environments that require multiple binary trees, there are often *a lot* of binary trees. To keep all these trees straight requires some thought.

A common way to organize this data is to designate a large disk for a binary tree "farm." At (or near) the top level of this disk is one directory for each binary tree.

One reasonable layout for these trees is to include in each directory name the vendor, hardware platform, operating system, and build parameters of the binary tree:

```
$ ls
hp-386-windows-optimized
hp-386-windows-debug
sgi-irix-optimzed
sgi-irix-debug
sun-solaris8-profiled
sun-solaris8-debug
```

When builds from many different times must be kept, it is usually best to include a date stamp (and even a timestamp) in the directory name. The format yymmdd or yymmddhhmm sorts well:

```
$ ls
hp-386-windows-optimized-040123
hp-386-windows-debug-040123
sgi-irix-optimzed-040127
sgi-irix-debug-040127
sun-solaris8-profiled-040127
sun-solaris8-debug-040127
```

Of course, the order of these filename components is up your site. The top-level directory of these trees is a good place to hold the *makefile* and testing logs.

This layout is appropriate for storing many parallel developer builds. If a development team makes "releases," possibly for internal customers, you can consider adding an additional release farm, structured as a set of products, each of which may have a version number and timestamp as shown in Figure 6-2.

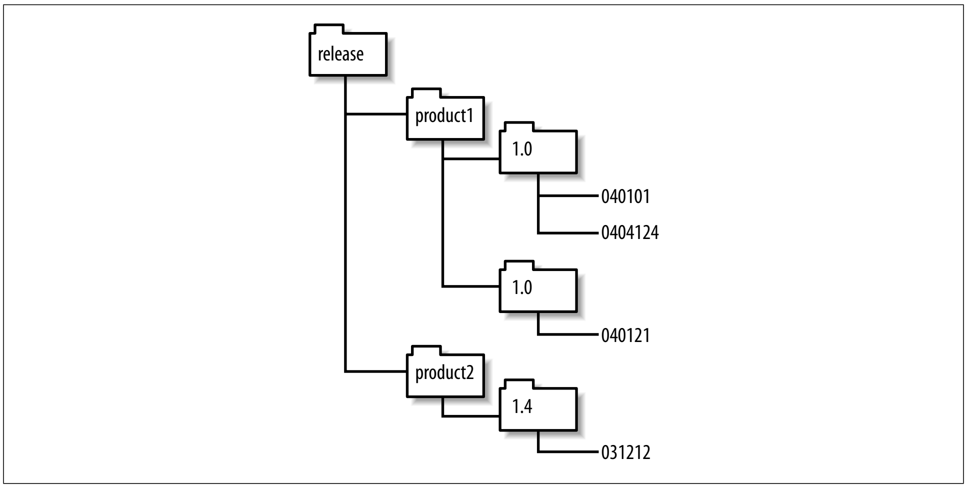

*Figure 6-2. Example of a release tree layout*

Here products might be libraries that are the output of a development team for use by other developers. Of course, they may also be products in the traditional sense.

Whatever your file layout or environment, many of the same criteria govern the implementation. It must be easy to identify each tree. Cleanup should be fast and obvious. It is useful to make it easy to move trees around and archive trees. In addition, the filesystem layout should closely match the process structure of the organization. This makes it easy for nonprogrammers such as managers, quality assurance, and technical publications to navigate the tree farm.

## Automating Builds and Testing

It is typically important to be able to automate the build process as much as possible. This allows reference tree builds to be performed at night, saving developer time during the day. It also allows developers themselves to run builds on their own machines unattended.

For software that is "in production," there are often many outstanding requests for builds of different versions of different products. For the person in charge of satisfying these requests, the ability to fire off several builds and "walk away" is often critical to maintaining sanity and satisfying requests.

Automated testing presents its own issues. Many nongraphical applications can use simple scripting to manage the testing process. The GNU tool `dejaGnu` can also be used to test nongraphical utilities that require interaction. Of course, testing frameworks like JUnit (*http://www.junit.org*) also provide support for nongraphical unit testing.

Testing of graphical applications presents special problems. For X11-based systems, I have successfully performed unattended, cron-based testing using the virtual frame buffer, `Xvfb`. On Windows, I have not found a satisfactory solution to unattended testing. All approaches rely on leaving the testing account logged in and the screen unlocked.

CHAPTER 7

# Portable Makefiles

What do we mean by a portable *makefile*? As an extreme example, we want a *makefile* that runs without change on any system that GNU make runs on. But this is virtually impossible due to the enormous variety in operating systems. A more reasonable interpretation is a *makefile* that is easy to change for each new platform it is run on. An important added constraint is that the port to the new system does not break support for the previous platforms.

We can achieve this level of portability for *makefile*s using the same techniques as traditional programming: encapsulation and abstraction. By using variables and user-defined functions we can encapsulate applications and algorithms. By defining variables for command-line arguments and parameters, we can abstract out elements that vary from platform to platform from elements that are constant.

You then have to determine what tools each platform offers to get your job done, and what to use from each platform. The extreme in portability is to use only those tools and features that exist on all platforms of interest. This is typically called the "least common denominator" approach and obviously can leave you with very primitive functionality to work with.

Another version of the least common denominator approach is to choose a powerful set of tools and make sure to bring it with you to every platform, thus guaranteeing that the commands you invoke in the *makefile* work exactly the same everywhere. This can be hard to pull off, both administratively and in terms of getting your organization to cooperate with your fiddling with their systems. But it can be successful, and I'll show one example of that later with the Cygwin package for Windows. As you'll see, standardizing on tools does not solve every problem; there are always operating system differences to deal with.

Finally, you can accept differences between systems and work around them by careful choices of macros and functions. I'll show this approach in this chapter, too.

So, by judicious use of variables and user-defined functions, and by minimizing the use of exotic features and relying on standard tools, we can increase the portability of our *makefile*s. As noted previously, there is no such thing as perfect portability, so it is our job to balance effort versus portability. But before we explore specific techniques, let's review some of the issues of portable *makefile*s.

## Portability Issues

Portability problems can be difficult to characterize since they span the entire spectrum from a total paradigm shift (such as traditional Mac OS versus System V Unix) to almost trivial bug fixes (such as a fix to a bug in the error exit status of a program). Nevertheless, here are some common portability problems that every *makefile* must deal with sooner or later:

*Program names*
: It is quite common for various platforms to use different names for the same or similar programs. The most common is the name of the C or C++ compiler (e.g., `cc`, `xlc`). It is also common for GNU versions of programs to be installed on a non-GNU system with the *g* prefix (e.g., `gmake`, `gawk`).

*Paths*
: The location of programs and files often varies between platforms. For instance, on Solaris systems the X directories are stored under */usr/X* while on many other systems the path is */usr/X11R6*. In addition, the distinction between */bin*, */usr/bin*, */sbin*, and */usr/sbin* is often rather fuzzy as you move from one system to another.

*Options*
: The command-line options to programs vary, particularly when an alternate implementation is used. Furthermore, if a platform is missing a utility or comes with a broken version, you may need to replace the utility with another that uses different command-line options.

*Shell features*
: By default, `make` executes command scripts with */bin/sh*, but `sh` implementations vary widely in their features. In particular, pre-POSIX shells are missing many features and will not accept the same syntax as a modern shell.

: The Open Group has a very useful white paper on the differences between the System V shell and the POSIX shell. It can be found at *http://www.unix-systems.org/whitepapers/shdiffs.html*. For those who want more details, the specification of the POSIX shell's command language can be found at *http://www.opengroup.org/onlinepubs/007904975/utilities/xcu_chap02.html*.

*Program behavior*
: Portable *makefile*s must contend with programs that simply behave differently. This is very common as different vendors fix or insert bugs and add features. There are also upgrades to utilities that may or may not have made it into a vendor's release. For instance, in 1987 the `awk` program underwent a major revision.

Nearly 20 years later, some systems still do not install this upgraded version as the standard awk.

*Operating system*
: Finally, there are the portability problems associated with a completely different operating system such as Windows versus Unix or Linux versus VMS.

## Cygwin

Although there is a native Win32 port of make, this is a small part of the Windows portability problem, because the shell this native port uses is *cmd.exe* (or *command.exe*). This, along with the absence of most of the Unix tools, makes cross-platform portability a daunting task. Fortunately, the Cygwin project (*http://www.cygwin.com*) has built a Linux-compatible library for Windows to which many programs[*] have been ported. For Windows developers who want Linux compatibility or access to GNU tools, I don't believe there is a better tool to be found.

I have used Cygwin for over 10 years on a variety of projects from a combined C++/Lisp CAD application to a pure Java workflow management system. The Cygwin tool set includes compilers and interpreters for many programming languages. However, Cygwin can be used profitably even when the applications themselves are implemented using non-Cygwin compilers and interpreters. The Cygwin tool set can be used solely as an aid to coordinating the development and build process. In other words, it is not necessary to write a "Cygwin" application or use Cygwin language tools to reap the benefits of the Cygwin environment.

Nevertheless, Linux is not Windows (thank goodness!) and there are issues involved when applying Cygwin tools to native Windows applications. Almost all of these issues revolve around the line endings used in files and the form of paths passed between Cygwin and Windows.

### Line Termination

Windows filesystems use a two-character sequence carriage return followed by line feed (or CRLF) to terminate each line of a text file. POSIX systems use a single character, a line feed (LF or *newline*). Occasionally this difference can cause the unwary some confusion as programs report syntax errors or seek to the wrong location in a data file. However, the Cygwin library does a very good job of working through these issues. When Cygwin is installed (or alternatively when the mount command is used), you can choose whether Cygwin should translate files with CRLF endings. If a DOS file format is selected, Cygwin will translate CRLF to LF when reading and the reverse when writing text files so that Unix-based programs can properly handle

[*] My Cygwin */bin* directory currently contains 1343 executables.

DOS text files. If you plan to use native language tools such as Visual C++ or Sun's Java SDK, choose the DOS file format. If you are going to use Cygwin compilers, choose Unix. (Your choice can be changed at any time.)

In addition, Cygwin comes with tools to translate files explicitly. The utilities `dos2unix` and `unix2dos` transform the line endings of a file, if necessary.

## Filesystem

Cygwin provides a POSIX view of the Windows filesystem. The root directory of a POSIX filesystem is /, which maps to the directory in which Cygwin is installed. Windows drives are accessible through the pseudo-directory */cygdrive/`letter`*. So, if Cygwin is installed in *C:\usr\cygwin* (my preferred location), the directory mappings shown in Table 7-1 would hold.

*Table 7-1. Default Cygwin directory mapping*

| Native Windows path | Cygwin path | Alternate Cygwin path |
|---|---|---|
| *c:\usr\cygwin* | */* | */cygdrive/c/usr/cygwin* |
| *c:\Program Files* | */cygdrive/c/Program Files* | |
| *c:\usr\cygwin\bin* | */bin* | */cygdrive/c/usr/cygwin/bin* |

This can be a little confusing at first, but doesn't pose any problems to tools. Cygwin also includes a `mount` command that allows users to access files and directories more conveniently. One option to `mount`, `--change-cygdrive-prefix`, allows you to change the prefix. I find that changing the prefix to simply / is particularly useful because drive letters can be accessed more naturally:

```
$ mount --change-cygdrive-prefix /
$ ls /c
AUTOEXEC.BAT            Home         Program Files             hp
BOOT.INI                I386         RECYCLER                  ntldr
CD                      IO.SYS       System Volume Information pagefile.sys
CONFIG.SYS              MSDOS.SYS    Temp                      tmp
C_DILLA                 NTDETECT.COM WINDOWS                   usr
Documents and Settings  PERSIST      WUTemp                    work
```

Once this change is made, our previous directory mapping would change to those shown in Table 7-2.

*Table 7-2. Modified Cygwin directory mapping*

| Native Windows path | Cygwin path | Alternate Cygwin path |
|---|---|---|
| *c:\usr\cygwin* | */* | */c/usr/cygwin* |
| *c:\Program Files* | */c/Program Files* | |
| *c:\usr\cygwin\bin* | */bin* | */c/usr/cygwin/bin* |

If you need to pass a filename to a Windows program, such as the Visual C++ compiler, you can usually just pass the relative path to the file using POSIX-style forward slashes. The Win32 API does not distinguish between forward and backward slashes. Unfortunately, some utilities that perform their own command-line argument parsing treat all forward slashes as command options. One such utility is the DOS `print` command; another is the `net` command.

If absolute paths are used, the drive letter syntax is always a problem. Although Windows programs are usually happy with forward slashes, they are completely unable to fathom the */c* syntax. The drive letter must always be tranformed back into *c:*. To accomplish this and the forward/backslash conversion, Cygwin provides the `cygpath` utility to translate between POSIX paths and Windows paths.

```
$ cygpath --windows /c/work/src/lib/foo.c
c:\work\src\lib\foo.c
$ cygpath --mixed /c/work/src/lib/foo.c
c:/work/src/lib/foo.c
$ cygpath --mixed --path "/c/work/src:/c/work/include"
c:/work/src;c:/work/include
```

The `--windows` option translates the POSIX path given on the command line into a Windows path (or vice versa with the proper argument). I prefer to use the `--mixed` option that produces a Windows path, but with forward slashes instead of backslashes (when the Windows utility accepts it). This plays much better with the Cygwin shell because the backslash is the escape character. The `cygpath` utility has many options, some of which provide portable access to important Windows paths:

```
$ cygpath --desktop
/c/Documents and Settings/Owner/Desktop
$ cygpath --homeroot
/c/Documents and Settings
$ cygpath --smprograms
/c/Documents and Settings/Owner/Start Menu/Programs
$ cygpath --sysdir
/c/WINDOWS/SYSTEM32
$ cygpath --windir
/c/WINDOWS
```

If you're using `cygpath` in a mixed Windows/Unix environment, you'll want to wrap these calls in a portable function:

```
ifdef COMSPEC
  cygpath-mixed         = $(shell cygpath -m "$1")
  cygpath-unix          = $(shell cygpath -u "$1")
  drive-letter-to-slash = /$(subst :,,$1)
else
  cygpath-mixed         = $1
  cygpath-unix          = $1
  drive-letter-to-slash = $1
endif
```

If all you need to do is map the *c:* drive letter syntax to the POSIX form, the `drive-letter-to-slash` function is faster than running the `cygpath` program.

Finally, Cygwin cannot hide all the quirks of Windows. Filenames that are invalid in Windows are also invalid in Cygwin. Thus, names such as *aux.h*, *com1*, and *prn* cannot be used in a POSIX path, even with an extension.

### Program Conflicts

Several Windows programs have the same names as Unix programs. Of course, the Windows programs do not accept the same command-line arguments or behave in compatible ways with the Unix programs. If you accidentally invoke the Windows versions, the usual result is serious confusion. The most troublesome ones seem to be `find`, `sort`, `ftp`, and `telnet`. For maximum portability, you should be sure to provide full paths to these programs when porting between Unix, Windows, and Cygwin.

If your commitment to Cygwin is strong and you do not need to build using native Windows support tools, you can safely place the Cygwin */bin* directory at the front of your Windows path. This will guarantee access to Cygwin tools over Windows versions.

If your *makefile* is working with Java tools, be aware that Cygwin includes the GNU `jar` program that is incompatible with the standard Sun *jar* file format. Therefore, the Java jdk *bin* directory should be placed before the Cygwin */bin* directory in your `Path` variable to avoid using Cygwin's `jar` program.

## Managing Programs and Files

The most common way to manage programs is to use a variable for program names or paths that are likely to change. The variables can be defined in a simple block, as we have seen:

```
MV ?= mv -f
RM ?= rm -f
```

or in a conditional block:

```
ifdef COMSPEC
  MV ?= move
  RM ?= del
else
  MV ?= mv -f
  RM ?= rm -f
endif
```

If a simple block is used, the values can be changed by resetting them on the command line, by editing the *makefile*, or (in this case because we used conditional assignment, `?=`) by setting an environment variable. As mentioned previously, one way to test for a Windows platform is to check for the COMSPEC variable, which is used by all Windows operating systems. Sometimes only a path needs to change:

```
ifdef COMSPEC
  OUTPUT_ROOT := d:
  GCC_HOME    := c:/gnu/usr/bin
else
  OUTPUT_ROOT := $(HOME)
  GCC_HOME    := /usr/bin
endif

OUTPUT_DIR := $(OUTPUT_ROOT)/work/binaries
CC := $(GCC_HOME)/gcc
```

This style results in a *makefile* in which most programs are invoked via make variables. Until you get used to it, this can make the *makefile* a little harder to read. However, variables are often more convenient to use in the *makefile* anyway, because they can be considerably shorter than the literal program name, particularly when full paths are used.

The same technique can be used to manage different command options. For instance, the built-in compilation rules include a variable, TARGET_ARCH, that can be used to set platform-specific flags:

```
ifeq "$(MACHINE)" "hpux-hppa"
  TARGET_ARCH := -mdisable-fpregs
endif
```

When defining your own program variables, you may need to use a similar approach:

```
MV := mv $(MV_FLAGS)

ifeq "$(MACHINE)" "solaris-sparc"
  MV_FLAGS := -f
endif
```

If you are porting to many platforms, chaining the ifdef sections can become ugly and difficult to maintain. Instead of using ifdef, place each set of platform-specific variables in its own file whose name contains a platform indicator. For instance, if you designate a platform by its uname parameters, you can select the appropriate make include file like this:

```
MACHINE := $(shell uname -smo | sed 's/ /-/g')
include $(MACHINE)-defines.mk
```

Filenames with spaces present a particularly irritating problem for make. The assumption that whitespace separates tokens during parsing is fundamental to make. Many built-in functions such as word, filter, wildcard, and others assume their arguments are space-separated words. Nevertheless, here are some tricks that may help in small ways. The first trick, noted in the section "Supporting Multiple Binary Trees" in Chapter 8, is how to replace spaces with another character using `subst`:

```
space = $(empty) $(empty)

# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,$1)
```

The `space-to-question` function replaces all spaces with the globbing wildcard question mark. Now, we can implement `wildcard` and `file-exists` functions that can handle spaces:

```
# $(call wildcard-spaces,file-name)
wildcard-spaces = $(wildcard $(call space-to-question,$1))

# $(call file-exists
file-exists = $(strip                                   \
                $(if $1,,$(warning $1 has no value))    \
                $(call wildcard-spaces,$1))
```

The `wildcard-spaces` function uses `space-to-question` to allow the *makefile* to perform a wildcard operation on a pattern including spaces. We can use our `wildcard-spaces` function to implement `file-exists`. Of course, the use of the question mark may also cause `wildcard-spaces` to return files that do not correctly match the original wildcard pattern (e.g., "my document.doc" and "my-document.doc"), but this is the best we can do.

The `space-to-question` function can also be used to transform filenames with spaces in targets and prerequisites, since those allow globbing patterns to be used.

```
space := $(empty) $(empty)

# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,$1)

# $(call question-to-space,file-name)
question-to-space = $(subst ?,$(space),$1)

$(call space-to-question,foo bar): $(call space-to-question,bar baz)
        touch "$(call question-to-space,$@)"
```

Assuming the file "*bar baz*" exists, the first time this *makefile* is executed the prerequisite is found because the globbing pattern is evaluated. But the target globbing pattern fails because the target does not yet exist, so `$@` has the value `foo?bar`. The command script then uses `question-to-space` to transform `$@` back to the file with spaces that we really want. The next time the *makefile* is run, the target is found because the globbing pattern finds the target with spaces. A bit ugly, but I have found these tricks useful in real *makefile*s.

## Source Tree Layout

Another aspect of portability is the ability to allow developers freedom to manage their development environment as they deem necessary. There will be problems if the build system requires the developers to always place their source, binaries, libraries, and support tools under the same directory or on the same Windows disk drive, for instance. Eventually, some developer low on disk space will be faced with the problem of having to partition these various files.

Instead, it makes sense to implement the *makefile* using variables to reference these collections of files and set reasonable defaults. In addition, each support library and tool can be referenced through a variable to allow developers to customize file locations as they find necessary. For the most likely customization variables, use the conditional assignment operator to allow developers a simple way of overriding the *makefile* with environment variables.

In addition, the ability to easily support multiple copies of the source and binary tree is a boon to developers. Even if they don't have to support different platforms or compilation options, developers often find themselves working with several copies of the source, either for debugging purposes or because they work on several projects in parallel. Two ways to support this have already been discussed: use a "top-level" environment variable to identify the root of the source and binary trees, or use the directory of the *makefile* and a fixed relative path to find the binary tree. Either of these allows developers the flexibility of supporting more than one tree.

## Working with Nonportable Tools

As noted previously, one alternative to writing *makefile*s to the least common denominator is to adopt some standard tools. Of course, the goal is to make sure the standard tools are at least as portable as the application you are building. The obvious choice for portable tools are programs from the GNU project, but portable tools come from a wide variety of sources. Perl and Python are two other tools that come to mind.

In the absence of portable tools, encapsulating nonportable tools in make functions can sometimes do just as well. For instance, to support a variety of compilers for Enterprise JavaBeans (each of which has a slightly different invocation syntax), we can write a basic function to compile an EJB jar and parameterize it to allow one to plug in different compilers.

```
EJB_TMP_JAR = $(TMPDIR)/temp.jar

# $(call compile-generic-bean, bean-type, jar-name,
#                              bean-files-wildcard, manifest-name-opt )
define compile-generic-bean
  $(RM) $(dir $(META_INF))
  $(MKDIR) $(META_INF)
  $(if $(filter %.xml %.xmi, $3),           \
    cp $(filter %.xml %.xmi, $3) $(META_INF))
  $(call compile-$1-bean-hook,$2)
  cd $(OUTPUT_DIR) &&                       \
  $(JAR) -cf0 $(EJB_TMP_JAR)                \
         $(call jar-file-arg,$(META_INF))    \
         $(call bean-classes,$3)
  $(call $1-compile-command,$2)
  $(call create-manifest,$(if $4,$4,$2),,)
endef
```

The first argument to this general EJB compilation function is the type of bean compiler we are using, such as Weblogic, Websphere, etc. The remaining arguments are the jar name, the files forming the content of the jar (including configuration files), and an optional manifest file. The template function first creates a clean temporary area by deleting any old temporary directory and recreating it. Next, the function copies in the *xml* or *xmi* files present in the prerequisites into the `$(META_INF)` directory. At this point, we may need to perform custom operations to clean up the *META-INF* files or prepare the *.class* files. To support these operations, we include a hook function, `compile-$1-bean-hook`, that the user can define, if necessary. For instance, if the Websphere compiler required an extra control file, say an *xsl* file, we would write this hook:

```
# $(call compile-websphere-bean-hook, file-list)
define compile-websphere-bean-hook
  cp $(filter %.xsl, $1) $(META_INF)
endef
```

By simply defining this function, we make sure the `call` in `compile-generic-bean` will be expanded appropriately. If we do not choose to write a hook function, the call in `compile-generic-bean` expands to nothing.

Next, our generic function creates the jar. The helper function `jar-file-arg` decomposes a normal file path into a `-C` option and a relative path:

```
# $(call jar-file-arg, file-name)
define jar-file-arg
  -C "$(patsubst %/,%,$(dir $1))" $(notdir $1)
endef
```

The helper function `bean-classes` extracts the appropriate class files from a source file list (the jar file only needs the interface and home classes):

```
# $(call bean-classes, bean-files-list)
define bean-classes
  $(subst $(SOURCE_DIR)/,,                  \
    $(filter %Interface.class %Home.class, \
      $(subst .java,.class,$1)))
endef
```

Then the generic function invokes the compiler of choice with `$(call $1-compile-command,$2)`:

```
define weblogic-compile-command
  cd $(TMPDIR) && \
  $(JVM) weblogic.ejbc -compiler $(EJB_JAVAC) $(EJB_TMP_JAR) $1
endef
```

Finally, our generic function adds the manifest.

Having defined `compile-generic-bean`, we wrap it in a compiler-specific function for each environment we want to support.

```
# $(call compile-weblogic-bean, jar-name,
#                               bean-files-wildcard, manifest-name-opt )
define compile-weblogic-bean
  $(call compile-generic-bean,weblogic,$1,$2,$3)
endef
```

### A Standard Shell

It is worth reiterating here that one of the irksome incompatibilities one finds in moving from system to system is the capabilities of */bin/sh*, the default shell used by `make`. If you find yourself tweaking the command scripts in your *makefile*, you should consider standardizing your shell. Of course, this is not reasonable for the typical open source project where the *makefile* is executed in uncontrolled environments. However, in a controlled setting, with a fixed set of specially configured machines, this is quite reasonable.

In addition to avoiding shell incompatibilities, many shells provide features that can avoid the use of numerous small utilities. For example, the `bash` shell includes enhanced shell variable expansion, such as `%%` and `##`, that can help avoid the use of shell utilities, such as `sed` and `expr`.

## Automake

The focus of this chapter has been on using GNU `make` and supporting tools effectively to achieve a portable build system. There are times, however, when even these modest requirements are beyond reach. If you cannot use the enhanced features of GNU `make` and are forced to rely on a least-common-denominator set of features, you should consider using the `automake` tool, *http://www.gnu.org/software/automake/automake.html*.

The `automake` tool accepts a stylized *makefile* as input and generates a portable old-style *makefile* as output. `automake` is built around a set of `m4` macros that allow a very terse notation in the input file (called *makefile.am*). Typically, `automake` is used in conjunction with `autoconf`, a portability support package for C/C++ programs, but `autoconf` is not required.

While automake is a good solution for build systems that require maxium portability, the *makefile*s it generates cannot use any of the advanced features of GNU make with the exception of appending assignment, +=, for which it has special support. Furthermore, the input to automake bears little resemblance to normal *makefile* input. Thus, using automake (without autoconf) isn't terribly different from using the least-common-denominator approach.

# CHAPTER 8
# C and C++

The issues and techniques shown in Chapter 6 are enhanced and applied in this chapter to C and C++ projects. We'll continue with the mp3 player example building on our nonrecursive *makefile*.

## Separating Source and Binary

If we want to support a single source tree with multiple platforms and multiple builds per platform, separating the source and binary trees is necessary, so how do we do it? The make program was originally written to work well for files in a single directory. Although it has changed dramatically since then, it hasn't forgotten its roots. make works with multiple directories best when the files it is updating live in the current directory (or its subdirectories).

### The Easy Way

The easiest way to get make to place binaries in a separate directory from sources is to start the make program from the binary directory. The output files are accessed using relative paths, as shown in the previous chapter, while the input files must be found either through explicit paths or through searching through vpath. In either case, we'll need to refer to the source directory in several places, so we start with a variable to hold it:

```
SOURCE_DIR := ../mp3_player
```

Building on our previous *makefile*, the source-to-object function is unchanged, but the subdirectory function now needs to take into account the relative path to the source.

```
# $(call source-to-object, source-file-list)
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))
```

```
# $(subdirectory)
subdirectory = $(patsubst $(SOURCE_DIR)/%/module.mk,%,  \
                 $(word                               \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))
```

In our new *makefile*, the files listed in the MAKEFILE_LIST will include the relative path to the source. So to extract the relative path to the module's directory, we must strip off the prefix as well as the *module.mk* suffix.

Next, to help make find the sources, we use the vpath feature:

```
vpath %.y $(SOURCE_DIR)
vpath %.l $(SOURCE_DIR)
vpath %.c $(SOURCE_DIR)
```

This allows us to use simple relative paths for our source files as well as our output files. When make needs a source file, it will search SOURCE_DIR if it cannot find the file in the current directory of the output tree. Next, we must update the include_dirs variable:

```
include_dirs := lib $(SOURCE_DIR)/lib $(SOURCE_DIR)/include
```

In addition to the source directories, this variable now includes the *lib* directory from the binary tree because the generated yacc and lex header files will be placed there.

The make include directive must be updated to access the *module.mk* files from their source directories since make does not use the vpath to find include files:

```
include $(patsubst %,$(SOURCE_DIR)/%/module.mk,$(modules))
```

Finally, we create the output directories themselves:

```
create-output-directories :=                          \
        $(shell for f in $(modules);                  \
                do                                    \
                  $(TEST) -d $$f || $(MKDIR) $$f;     \
                done)
```

This assignment creates a dummy variable whose value is never used, but because of the simple variable assignment we are guaranteed that the directories will be created before make performs any other work. We must create the directories "by hand" because yacc, lex, and the dependency file generation will not create the output directories themselves.

Another way to ensure these directories are created is to add the directories as prerequisites to the dependency files (the *.d* files). This is a bad idea because the directory is not really a prerequisite. The yacc, lex, or dependency files do not depend on the *contents* of the directory, nor should they be regenerated just because the directory timestamp is updated. In fact, this would be a source of great inefficiency if the project were remade when a file was added or removed from an output directory.

The modifications to the *module.mk* file are even simpler:

```
local_src := $(addprefix $(subdirectory)/,playlist.y scanner.l)

$(eval $(call make-library, $(subdirectory)/libdb.a, $(local_src)))

.SECONDARY: $(call generated-source, $(local_src))

$(subdirectory)/scanner.d: $(subdirectory)/playlist.d
```

This version omits the wildcard to find the source. It is a straightforward matter to restore this feature and is left as an exercise for the reader. There is one glitch that appears to be a bug in the original *makefile*. When this example was run, I discovered that the *scanner.d* dependency file was being generated before *playlist.h*, which it depends upon. This dependency was missing from the original *makefile*, but it worked anyway purely by accident. Getting *all* the dependencies right is a difficult task, even in small projects.

Assuming the source is in the subdirectory *mp3_player*, here is how we build our project with the new *makefile*:

```
$ mkdir mp3_player_out
$ cd mp3_player_out
$ make --file=../mp3_player/makefile
```

The *makefile* is correct and works well, but it is rather annoying to be forced to change directories to the output directory and then be forced to add the `--file` (`-f`) option. This can be cured with a simple shell script:

```
#! /bin/bash
if [[ ! -d $OUTPUT_DIR ]]
then
  if ! mkdir -p $OUTPUT_DIR
  then
    echo "Cannot create output directory" > /dev/stderr
    exit 1
  fi
fi

cd $OUTPUT_DIR
make --file=$SOURCE_DIR/makefile "$@"
```

This script assumes the source and output directories are stored in the environment variables `SOURCE_DIR` and `OUTPUT_DIR`, respectively. This is a standard practice that allows developers to switch trees easily but still avoid typing paths too frequently.

One last caution. There is nothing in `make` or our *makefile* to prevent a developer from executing the *makefile* from the source tree, even though it should be executed from the binary tree. This is a common mistake and some command scripts might behave badly. For instance, the `clean` target:

```
.PHONY: clean
clean:
        $(RM) -r *
```

would delete the user's entire source tree! Oops. It seems prudent to add a check for this eventuality in the *makefile* at the highest level. Here is a reasonable check:

```
    $(if $(filter $(notdir $(SOURCE_DIR)),$(notdir $(CURDIR))),\
      $(error Please run the makefile from the binary tree.))
```

This code tests if the name of the current working directory (`$(notdir $(CURDIR))`) is the same as the source directory (`$(notdir $(SOURCE_DIR))`). If so, print the error and exit. Since the `if` and `error` functions expand to nothing, we can place these two lines immediately after the definition of `SOURCE_DIR`.

## The Hard Way

Some developers find having to `cd` into the binary tree so annoying that they will go to great lengths to avoid it, or maybe the *makefile* maintainer is working in an environment where shell script wrappers or aliases are unsuitable. In any case, the *makefile* can be modified to allow running `make` from the source tree and placing binary files in a separate output tree by prefixing all the output filenames with a path. At this point I usually go with absolute paths since this provides more flexibility, although it does exacerbate problems with command-line length limits. The input files continue to use simple relative paths from the *makefile* directory.

Example 8-1 shows the *makefile* modified to allow executing `make` from the source tree and writing binary files to a binary tree.

*Example 8-1. A makefile separating source and binary that can be executed from the source tree*

```
SOURCE_DIR := /test/book/examples/ch07-separate-binaries-1
BINARY_DIR := /test/book/out/mp3_player_out

# $(call source-dir-to-binary-dir, directory-list)
source-dir-to-binary-dir = $(addprefix $(BINARY_DIR)/, $1)

# $(call source-to-object, source-file-list)
source-to-object = $(call source-dir-to-binary-dir,     \
                     $(subst .c,.o,$(filter %.c,$1))    \
                     $(subst .y,.o,$(filter %.y,$1))    \
                     $(subst .l,.o,$(filter %.l,$1)))

# $(subdirectory)
subdirectory = $(patsubst %/module.mk,%,                \
                 $(word                                 \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))

# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $(BINARY_DIR)/$1
  sources   += $2

  $(BINARY_DIR)/$1: $(call source-dir-to-binary-dir,    \
                      $(subst .c,.o,$(filter %.c,$2))   \
```

*Example 8-1. A makefile separating source and binary that can be executed from the source tree (continued)*

```
                      $(subst .y,.o,$(filter %.y,$2))   \
                      $(subst .l,.o,$(filter %.l,$2)))
        $(AR) $(ARFLAGS) $$@ $$^
endef

# $(call generated-source, source-file-list)
generated-source = $(call source-dir-to-binary-dir,     \
                     $(subst .y,.c,$(filter %.y,$1))    \
                     $(subst .y,.h,$(filter %.y,$1))    \
                     $(subst .l,.c,$(filter %.l,$1)))   \
                   $(filter %.c,$1)

# $(compile-rules)
define compile-rules
  $(foreach f, $(local_src),\
    $(call one-compile-rule,$(call source-to-object,$f),$f))
endef

# $(call one-compile-rule, binary-file, source-files)
define one-compile-rule
  $1: $(call generated-source,$2)
        $(COMPILE.c) -o $$@ $$<

  $(subst .o,.d,$1): $(call generated-source,$2)
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $$< | \
        $(SED) 's,\($$(notdir $$*)\.o\) *:,$$(dir $$@)\1 $$@: ,' > $$@.tmp
        $(MV) $$@.tmp $$@

endef

modules      := lib/codec lib/db lib/ui app/player
programs     :=
libraries    :=
sources      :=

objects      =  $(call source-to-object,$(sources))
dependencies =  $(subst .o,.d,$(objects))

include_dirs := $(BINARY_DIR)/lib lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MKDIR := mkdir -p
MV    := mv -f
RM    := rm -f
SED   := sed
TEST  := test

create-output-directories :=                                       \
        $(shell for f in $(call source-dir-to-binary-dir,$(modules));   \
                do                                                     \
```

*Example 8-1. A makefile separating source and binary that can be executed from the source tree (continued)*

```
                $(TEST) -d $$f || $(MKDIR) $$f;                       \
              done)

all:

include $(addsuffix /module.mk,$(modules))

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
        $(RM) -r $(BINARY_DIR)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif
```

In this version the `source-to-object` function is modified to prepend the path to the binary tree. This prefixing operation is performed several times, so write it as a function:

```
SOURCE_DIR := /test/book/examples/ch07-separate-binaries-1
BINARY_DIR := /test/book/out/mp3_player_out

# $(call source-dir-to-binary-dir, directory-list)
source-dir-to-binary-dir = $(addprefix $(BINARY_DIR)/, $1)

# $(call source-to-object, source-file-list)
source-to-object = $(call source-dir-to-binary-dir,     \
                     $(subst .c,.o,$(filter %.c,$1))    \
                     $(subst .y,.o,$(filter %.y,$1))    \
                     $(subst .l,.o,$(filter %.l,$1)))
```

The `make-library` function is similarly altered to prefix the output file with `BINARY_DIR`. The `subdirectory` function is restored to its previous version since the include path is again a simple relative path. One small snag; a bug in `make` 3.80 prevents calling `source-to-object` within the new version of `make-library`. This bug has been fixed in 3.81. We can work around the bug by hand expanding the `source-to-object` function.

Now we get to the truly ugly part. When the output file is not directly accessible from a path relative to the *makefile*, the implicit rules no longer fire. For instance, the basic compile rule `%.o: %.c` works well when the two files live in the same directory, or even if the C file is in a subdirectory, say *lib/codec/codec.c*. When the source file lives in a remote directory, we can instruct `make` to search for the source with the `vpath` feature. But when the object file lives in a remote directory, `make` has no way of determining where the object file resides and the target/prerequisite chain is broken.

The only way to inform make of the location of the output file is to provide an explicit rule linking the source and object files:

```
$(BINARY_DIR)/lib/codec/codec.o: lib/codec/codec.c
```

This must be done for every single object file.

Worse, this target/prerequisite pair is not matched against the implicit rule, `%.o: %.c`. That means we must also provide the command script, duplicating whatever is in the implicit database and possibly repeating this script many times. The problem also applies to the automatic dependency generation rule we've been using. Adding two explicit rules for every object file in a *makefile* is a maintenance nightmare, if done by hand. However, we can minimize the code duplication and maintenance by writing a function to generate these rules:

```
# $(call one-compile-rule, binary-file, source-files)
define one-compile-rule
  $1: $(call generated-source,$2)
        $(COMPILE.c) $$@ $$<

  $(subst .o,.d,$1): $(call generated-source,$2)
        $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $$< | \
        $(SED) 's,\($$(notdir $$*)\.o\) *:,$$(dir $$@)\1 $$@: ,' > $$@.tmp
        $(MV) $$@.tmp $$@

endef
```

The first two lines of the function are the explicit rule for the object-to-source dependency. The prerequisites for the rule must be computed using the generated-source function we wrote in Chapter 6 because some of the source files are yacc and lex files that will cause compilation failures when they appear in the command script (expanded with `$^`, for instance). The automatic variables are quoted so they are expanded later when the command script is executed rather than when the user-defined function is evaluated by eval. The generated-source function has been modified to return C files unaltered as well as the generated source for yacc and lex:

```
# $(call generated-source, source-file-list)
generated-source = $(call source-dir-to-binary-dir,     \
                     $(subst .y,.c,$(filter %.y,$1))    \
                     $(subst .y,.h,$(filter %.y,$1))    \
                     $(subst .l,.c,$(filter %.l,$1)))   \
                   $(filter %.c,$1)
```

With this change, the function now produces this output:

```
Argument               Result
lib/db/playlist.y      /c/mp3_player_out/lib/db/playlist.c
                       /c/mp3_player_out/lib/db/playlist.h
lib/db/scanner.l       /c/mp3_player_out/lib/db/scanner.c
app/player/play_mp3.c  app/player/play_mp3.c
```

The explicit rule for dependency generation is similar. Again, note the extra quoting (double dollar signs) required by the dependency script.

Our new function must now be expanded for each source file in a module:

```
# $(compile-rules)
define compile-rules
  $(foreach f, $(local_src),\
    $(call one-compile-rule,$(call source-to-object,$f),$f))
endef
```

This function relies on the global variable `local_src` used by the *module.mk* files. A more general approach would pass this file list as an argument, but in this project it seems unnecessary. These functions are easily added to our *module.mk* files:

```
local_src := $(subdirectory)/codec.c

$(eval $(call make-library,$(subdirectory)/libcodec.a,$(local_src)))

$(eval $(compile-rules))
```

We must use `eval` because the `compile-rules` function expands to more than one line of `make` code.

There is one last complication. If the standard C compilation pattern rule fails to match with binary output paths, the implicit rule for `lex` and our pattern rule for `yacc` will also fail. We can update these by hand easily. Since they are no longer applicable to other `lex` or `yacc` files, we can move them into *lib/db/module.mk*:

```
local_dir := $(BINARY_DIR)/$(subdirectory)
local_src := $(addprefix $(subdirectory)/,playlist.y scanner.l)

$(eval $(call make-library,$(subdirectory)/libdb.a,$(local_src)))

$(eval $(compile-rules))

.SECONDARY: $(call generated-source, $(local_src))

$(local_dir)/scanner.d: $(local_dir)/playlist.d

$(local_dir)/%.c $(local_dir)/%.h: $(subdirectory)/%.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $(dir $@)$*.c
        $(MV) y.tab.h $(dir $@)$*.h

$(local_dir)/scanner.c: $(subdirectory)/scanner.l
        @$(RM) $@
        $(LEX.l) $< > $@
```

The `lex` rule has been implemented as a normal explicit rule, but the `yacc` rule is a pattern rule. Why? Because the `yacc` rule is used to build two targets, a C file and a header file. If we used a normal explicit rule, `make` would execute the command script twice, once for the C file to be created and once for the header. But `make` assumes that a pattern rule with multiple targets updates both targets with a single execution.

If possible, instead of the *makefile*s shown in this section, I would use the simpler approach of compiling from the binary tree. As you can see, complications arise immediately (and seem to get worse and worse) when trying to compile from the source tree.

## Read-Only Source

Once the source and binary trees are separate, the ability to make a reference source tree read-only often comes for free if the only files generated by the build are the binary files placed in the output tree. However, if source files are generated, then we must take care that they are placed in the binary tree.

In the simpler "compile from binary tree" approach, the generated files are written into the binary tree automatically because the `yacc` and `lex` programs are executed from the binary tree. In the "compile from source tree" approach, we are forced to provide explicit paths for our source and target files, so specifying the path to a binary tree file is no extra work, except that we must remember to do it.

The other obstacles to making the reference source tree read only are usually self-imposed. Often a legacy build system will include actions that create files in the source tree because the original author had not considered the advantages to a read-only source tree. Examples include generated documentation, log files, and temporary files. Moving these files to the output tree can sometimes be arduous, but if building multiple binary trees from a single source is necessary, the alternative is to maintain multiple, identical source trees and keep them in sync.

## Dependency Generation

We gave a brief introduction to dependency generation in the section "Automatic Dependency Generation" in Chapter 2, but it left several problems unaddressed. Therefore, this section offers some alternatives to the simple solution already described.* In particular, the simple approach described earlier and in the GNU `make` manual suffer from these failings:

- It is inefficient. When `make` discovers that a dependency file is missing or out of date, it updates the *.d* file and restarts itself. Rereading the *makefile* can be inefficient if it performs many tasks during the reading of the *makefile* and the analysis of the dependency graph.
- `make` generates a warning when you build a target for the first time and each time you add new source files. At these times the dependency file associated with a

* Much of the material in this section was invented by Tom Tromey (*tromey@cygnus.com*) for the GNU `automake` utility and is taken from the excellent summary article by Paul Smith (the maintainer of GNU `make`) from his web site *http://make.paulandlesley.org*.

new source file does not yet exist, so when make attempts to read the dependency file it will produce a warning message before generating the dependency file. This is not fatal, merely irritating.

- If you remove a source file, make stops with a fatal error during subsequent builds. In this situation, there exists a dependency file containing the removed file as a prerequisite. Since make cannot find the removed file and doesn't know how to make it, make prints the message:

  ```
  make: *** No rule to make target foo.h, needed by foo.d.  Stop.
  ```

  Furthermore, make cannot rebuild the dependency file because of this error. The only recourse is to remove the dependency file by hand, but since these files are often hard to find, users typically delete all the dependency files and perform a clean build. This error also occurs when files are renamed.

  Note that this problem is most noticeable with removed or renamed header files rather than *.c* files. This is because *.c* files will be removed from the list of dependency files automatically and will not trouble the build.

## Tromey's Way

Let's address these problems individually.

How can we avoid restarting make?

On careful consideration, we can see that restarting make is unnecessary. If a dependency file is updated, it means that at least one of its prerequisites has changed, which means we must update the target. Knowing more than that isn't necessary in this execution of make because more dependency information won't change make's behavior. But we want the dependency file updated so that the next run of make will have complete dependency information.

Since we don't need the dependency file in this execution of make, we could generate the file at the same time as we update the target. We can do this by rewriting the compilation rule to also update the dependency file.

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $1 | \
  $(SED) 's,\($$(notdir $2)\) *:,$$(dir $2) $3: ,' > $3.tmp
  $(MV) $3.tmp $3
endef

%.o: %.c
        $(call make-depend,$<,$@,$(subst .o,.d,$@))
        $(COMPILE.c) -o $@ $<
```

We implement the dependency generation feature with the function make-depend that accepts the source, object, and dependency filenames. This provides maximum flexibility if we need to reuse the function later in a different context. When we modify our compilation rule this way, we must delete the `%.d: %.c` pattern rule we wrote to avoid generating the dependency files twice.

Now, the object file and dependency file are logically linked: if one exists the other must exist. Therefore, we don't really care if a dependency file is missing. If it is, the object file is also missing and both will be updated by the next build. So we can now ignore any warnings that result from missing *.d* files.

In the section "Include and Dependencies" in Chapter 3, I introduced an alternate form of `include` directive, `-include` (or `sinclude`), that ignores errors and does not generate warnings:

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif
```

This solves the second problem, that of an annoying message when a dependency file does not yet exist.

Finally, we can avoid the warning when missing prerequisites are discovered with a little trickery. The trick is to create a target for the missing file that has no prerequisites and no commands. For example, suppose our dependency file generator has created this dependency:

```
target.o target.d: header.h
```

Now suppose that, due to code refactoring, *header.h* no longer exists. The next time we run the *makefile* we'll get the error:

```
make: *** No rule to make target header.h, needed by target.d.  Stop.
```

But if we add a target with no command for *header.h* to the dependency file, the error does not occur:

```
target.o target.d: header.h
header.h:
```

This is because, if *header.h* does not exist, it will simply be considered out of date and any targets that use it as a prerequisite will be updated. So the dependency file will be regenerated without *header.h* because it is no longer referenced. If *header.h* does exist, `make` considers it up to date and continues. So, all we need to do is ensure that every prerequisite has an associated empty rule. You may recall that we first encountered this kind of rule in the section "Phony Targets" in Chapter 2. Here is a version of `make-depend` that adds the new targets:

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $1 |        \
  $(SED) 's,\($$(notdir $2)\) *:,$$(dir $2) $3: ,' > $3.tmp
  $(SED) -e 's/#.*//'                                        \
         -e 's/^[^:]*: *//'                                  \
         -e 's/ *\\$$$$//'                                   \
         -e '/^$$$$/ d'                                      \
          -e 's/$$$$/ :/' $3.tmp >> $3.tmp
  $(MV) $3.tmp $3
endef
```

We execute a new sed command on the dependency file to generate the additional rules. This chunk of sed code performs five transformations:

1. Deletes comments
2. Deletes the target file(s) and subsequent spaces
3. Deletes trailing spaces
4. Deletes blank lines
5. Adds a colon to the end of every line

(GNU sed is able to read from a file and append to it in a single command line, saving us from having to use a second temporary file. This feature may not work on other systems.) The new sed command will take input that looks like:

```
# any comments
target.o target.d: prereq1 prereq2 prereq3 \
     prereq4
```

and transform it into:

```
prereq1 prereq2 prereq3:
prereq4:
```

So make-depend appends this new output to the original dependency file. This solves the "No rule to make target" error.

## makedepend Programs

Up to now we have been content to use the -M option provided by most compilers, but what if this option doesn't exist? Alternatively, are there better options than our simple -M?

These days most C compilers have some support for generating make dependencies from the source, but not long ago this wasn't true. In the early days of the X Window System project, they implemented a tool, makedepend, that computes the dependencies from a set of C or C++ sources. This tool is freely available over the Internet. Using makedepend is a little awkward because it is written to append its output to the *makefile*, which we do not want to do. The output of makedepend assumes the object files reside in the same directory as the source. This means that, again, our sed expression must change:

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(MAKEDEPEND) -f- $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $1 | \
  $(SED) 's,^.*/\([^/]*\.o\) *:,$(dir $2)\1 $3: ,' > $3.tmp
  $(SED) -e 's/#.*//'                                    \
         -e 's/^[^:]*: *//'                              \
         -e 's/ *\\$$$$//'                              \
         -e '/^$$$$/ d'                                 \
         -e 's/$$$$/ :/' $3.tmp >> $3.tmp
  $(MV) $3.tmp $3
endef
```

The `-f-` option tells `makedepend` to write its dependency information to the standard output.

An alternative to using `makedepend` or your native compiler is to use `gcc`. It sports a bewildering set of options for generating dependency information. The ones that seem most apropos for our current requirements are:

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(GCC) -MM           \
         -MF $3        \
         -MP           \
         -MT $2        \
         $(CFLAGS)     \
         $(CPPFLAGS)   \
         $(TARGET_ARCH) \
         $1
endef

%.o: %.c
        $(call make-depend,$<,$@,$(subst .o,.d,$@))
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

The `-MM` option causes `gcc` to omit "system" headers from the prerequisites list. This is useful because these files rarely, if ever, change and, as the build system gets more complex, reducing the clutter helps. Originally, this may have been done for performance reasons. With today's processors, the performance difference is barely measurable.

The `-MF` option specifies the dependency filename. This will be the object filename with the *.d* suffix substituted for *.o*. There is another `gcc` option, `-MD` or `-MMD`, that automatically generates the output filename using a similar substitution. Ideally we would prefer to use this option, but the substitution fails to include the proper relative path to the object file directory and instead places the *.d* file in the current directory. So, we are forced to do the job ourselves using `-MF`.

The `-MP` option instructs `gcc` to include phony targets for each prerequisite. This completely eliminates the messy five-part `sed` expression in our `make-depend` function. It seems that the `automake` developers who invented the phony target technique caused this option to be added to `gcc`.

Finally, the -MT option specifies the string to use for the target in the dependency file. Again, without this option, gcc fails to include the relative path to the object file output directory.

By using gcc, we can reduce the four commands previously required for dependency generation to a single command. Even when proprietary compilers are used it may be possible to use gcc for dependency management.

## Supporting Multiple Binary Trees

Once the *makefile* is modified to write binary files into a separate tree, supporting many trees becomes quite simple. For interactive or developer-invoked builds, where a developer initiates a build from the keyboard, there is little or no preparation required. The developer creates the output directory, cd's to it and invokes make on the *makefile*.

```
$ mkdir -p ~/work/mp3_player_out
$ cd ~/work/mp3_player_out
$ make -f ~/work/mp3_player/makefile
```

If the process is more involved than this, then a shell script wrapper is usually the best solution. This wrapper can also parse the current directory and set an environment variable like BINARY_DIR for use by the *makefile*.

```
#! /bin/bash

# Assume we are in the source directory.
curr=$PWD
export SOURCE_DIR=$curr
while [[ $SOURCE_DIR ]]
do
  if [[ -e $SOURCE_DIR/[Mm]akefile ]]
  then
    break;
  fi
  SOURCE_DIR=${SOURCE_DIR%/*}
done

# Print an error if we haven't found a makefile.
if [[ ! $SOURCE_DIR ]]
then
  printf "run-make: Cannot find a makefile" > /dev/stderr
  exit 1
fi

# Set the output directory to a default, if not set.
if [[ ! $BINARY_DIR ]]
then
  BINARY_DIR=${SOURCE_DIR}_out
fi
```

```
# Create the output directory
mkdir --parents $BINARY_DIR

# Run the make.
make --directory="$BINARY_DIR" "$@"
```

This particular script is a bit fancier. It searches for the *makefile* first in the current directory and then in the parent directory on up the tree until a *makefile* is found. It then checks that the variable for the binary tree is set. If not, it is set by appending "_out" to the source directory. The script then creates the output directory and executes make.

If the build is being performed on different platforms, some method for differentiating between platforms is required. The simplest approach is to require the developer to set an environment variable for each type of platform and add conditionals to the *makefile* and source based on this variable. A better approach is to set the platform type automatically based on the output of uname.

```
space := $(empty) $(empty)
export MACHINE := $(subst $(space),-,$(shell uname -smo))
```

If the builds are being invoked automatically from cron, I've found that a helper shell script is a better approach than having cron invoke make itself. A wrapper script provides better support for setup, error recovery, and finalization of an automated build. The script is also an appropriate place to set variables and command-line parameters.

Finally, if a project supports a fixed set of trees and platforms, you can use directory names to automatically identify the current build. For example:

```
ALL_TREES := /builds/hp-386-windows-optimized \
             /builds/hp-386-windows-debug     \
             /builds/sgi-irix-optimzed        \
             /builds/sgi-irix-debug           \
             /builds/sun-solaris8-profiled    \
             /builds/sun-solaris8-debug

BINARY_DIR := $(foreach t,$(ALL_TREES),\
                $(filter $(ALL_TREES)/%,$(CURDIR)))

BUILD_TYPE := $(notdir $(subst -,/,$(BINARY_DIR)))

MACHINE_TYPE := $(strip                         \
                  $(subst /,-,                  \
                    $(patsubst %/,%,            \
                      $(dir                     \
                        $(subst -,/,            \
                          $(notdir $(BINARY_DIR)))))))
```

The ALL_TREES variable holds a list of all valid binary trees. The foreach loop matches the current directory against each of the valid binary trees. Only one can match. Once the binary tree has been identified, we can extract the build type (e.g., optimized, debug, or profiled) from the build directory name. We retrieve the last component of the directory name by transforming the dash-separated words into slash-separated words and grabbing the last word with `notdir`. Similarly, we retrieve the machine type by grabbing the last word and using the same technique to remove the last dash component.

## Partial Source Trees

On really large projects, just checking out and maintaining the source can be a burden on developers. If a system consists of many modules and a particular developer is modifying only a localized part of it, checking out and compiling the entire project can be a large time sink. Instead, a centrally managed build, performed nightly, can be used to fill in the holes in a developer's source and binary trees.

Doing so requires two types of search. First, when a missing header file is required by the compiler, it must be instructed to search in the reference source tree. Second, when the *makefile* requires a missing library, it must be told to search in the reference binary tree. To help the compiler find source, we can simply add additional `-I` options after the `-I` options specifying local directories. To help `make` find libraries, we can add additional directories to the `vpath`.

```
SOURCE_DIR     := ../mp3_player
REF_SOURCE_DIR := /reftree/src/mp3_player
REF_BINARY_DIR := /binaries/mp3_player
…
include_dirs := lib $(SOURCE_DIR)/lib $(SOURCE_DIR)/include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))                \
                $(addprefix -I $(REF_SOURCE_DIR)/,$(include_dirs))
vpath %.h       $(include_dirs)                                 \
                $(addprefix $(REF_SOURCE_DIR)/,$(include_dirs))

vpath %.a       $(addprefix $(REF_BINARY_DIR)/lib/, codec db ui)
```

This approach assumes that the "granularity" of a CVS check out is a library or program module. In this case, the `make` can be contrived to skip missing library and program directories if a developer has chosen not to check them out. When it comes time to use these libraries, the search path will automatically fill in the missing files.

In the *makefile*, the `modules` variable lists the set of subdirectories to be searched for *module.mk* files. If a subdirectory is not checked out, this list must be edited to remove the subdirectory. Alternatively, the *modules* variable can be set by wildcard:

```
modules := $(dir $(wildcard lib/*/module.mk))
```

This expression will find all the subdirectories containing a *module.mk* file and return the directory list. Note that because of how the `dir` function works, each directory will contain a trailing slash.

It is also possible for `make` to manage partial source trees at the individual file level, building libraries by gathering some object files from a local developer tree and missing files from a reference tree. However, this is quite messy and developers are not happy with it, in my experience.

## Reference Builds, Libraries, and Installers

At this point we've pretty much covered everything needed to implement reference builds. Customizing the single top-level *makefile* to support the feature is straightforward. Simply replace the simple assignments to `SOURCE_DIR` and `BINARY_DIR` with `?=` assignments. The scripts you run from `cron` can use this basic approach:

1. Redirect output and set the names of log files
2. Clean up old builds and clean the reference source tree
3. Check out fresh source
4. Set the source and binary directory variables
5. Invoke `make`
6. Scan the logs for errors
7. Compute tags files, and possibly update the locate database*
8. Post information on the success or failure of the build

It is convenient, in the reference build model, to maintain a set of old builds in case a rogue check-in corrupts the tree. I usually keep 7 or 14 nightly builds. Of course, the nightly build script logs its output to files stored near the builds themselves and the script purges old builds and logs. Scanning the logs for errors is usually done with an `awk` script. Finally, I usually have the script maintain a *latest* symbolic link. To determine if the build is valid, I include a `validate` target in each *makefile*. This target performs simple validation that the targets were built.

```
.PHONY: validate_build
validate_build:
        test $(foreach f,$(RELEASE_FILES),-s $f -a) -e .
```

This command script simply tests if a set of expected files exists and is not empty. Of course, this doesn't take the place of testing, but is a convenient sanity check for a build. If the test returns failure, the `make` returns failure and the nightly build script can leave the *latest* symbolic link pointing to the old build.

Third-party libraries are always a bit of a hassle to manage. I subscribe to the commonly held belief that it is bad to store large binary files in CVS. This is because CVS cannot store deltas as diffs and the underlying RCS files can grow to enormous size.

* The locate database is a compilation of all the filenames present on a filesystem. It is a fast way of performing a `find` by name. I have found this database invaluable for managing large source trees and like to have it updated nightly after the build has completed.

Very large files in the CVS repository can slow down many common CVS operations, thus affecting all development.

If third-party libraries are not stored in CVS, they must be managed some other way. My current preference is to create a library directory in the reference tree and record the library version number in the directory name, as shown in Figure 8-1.

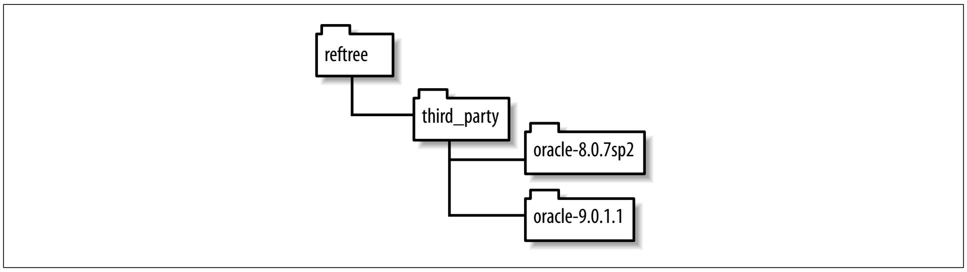

*Figure 8-1. Directory layout for third-party libraries*

These directory names are referenced by the *makefile*:

```
ORACLE_9011_DIR ?= /reftree/third_party/oracle-9.0.1.1/Ora90
ORACLE_9011_JAR ?= $(ORACLE_9011_DIR)/jdbc/lib/classes12.jar
```

When the vendor updates its libraries, create a new directory in the reference tree and declare new variables in the *makefile*. This way the *makefile*, which is properly maintained with tags and branches, always explicitly reflects the versions being used.

Installers are also a difficult issue. I believe that separating the basic build process from creating the installer image is a good thing. Current installer tools are complex and fragile. Folding them into the (also often complex and fragile) build system yields difficult-to-maintain systems. Instead, the basic build can write its results into a “release” directory that contains all (or most of) the data required by the installer build tool. This tool may be driven from its own *makefile* that ultimately yields an executable setup image.

CHAPTER 9

# Java

Many Java developers like Integrated Development Environments (IDEs) such as Eclipse. Given such well-known alternatives as Java IDEs and `Ant`, readers could well ask why they should even think of using `make` on Java projects. This chapter explores the value of `make` in these situations; in particular, it presents a generalized *makefile* that can be dropped into just about any Java project with minimal modification and carry out all the standard rebuilding tasks.

Using `make` with Java raises several issues and introduces some opportunities. This is primarily due to three factors: the Java compiler, `javac`, is extremely fast; the standard Java compiler supports the `@filename` syntax for reading "command-line parameters" from a file; and if a Java package is specified, the Java language specifies a path to the *.class* file.

Standard Java compilers are very fast. This is primarily due to the way the `import` directive works. Similar to a `#include` in C, this directive is used to allow access to externally defined symbols. However, rather than rereading source code, which then needs to be reparsed and analyzed, Java reads the class files directly. Because the symbols in a class file cannot change during the compilation process, the class files are cached by the compiler. In even medium-sized projects, this means the Java compiler can avoid rereading, parsing, and analyzing literally millions of lines of code compared with C. A more modest performance improvement is due to the bare minimum of optimization performed by most Java compilers. Instead, Java relies on sophisticated just-in-time (JIT) optimizations performed by the Java virtual machine (JVM) itself.

Most large Java projects make extensive use of Java's *package* feature. A class is declared to be encapsulated in a `package` that forms a scope around the symbols defined by the file. Package names are hierarchical and implicitly define a file structure. For instance, the package `a.b.c` would implicitly define a directory structure *a/b/c*. Code declared to be within the `a.b.c` package would be compiled to class files in the *a/b/c* directory. This means that `make`'s normal algorithm for associating a binary file with its source fails. But it also means that there is no need to specify a `-o` option to indicate where output files should be placed. Indicating the root of the output tree, which is the same for all files, is sufficient. This, in turn, means that source files from different directories can be compiled with the same command-line invocation.

The standard Java compilers all support the `@filename` syntax that allows command-line parameters to be read from a file. This is significant in conjunction with the package feature because it means that the entire Java source for a project can be compiled with a single execution of the Java compiler. This is a major performance improvement because the time it takes to load and execute the compiler is a major contributor to build times.

In summary, by composing the proper command line, compiling 400,000 lines of Java takes about three minutes on a 2.5-GHz Pentium 4 processor. Compiling an equivalent C++ application would require hours.

## Alternatives to make

As previously mentioned, the Java developer community enthusiastically adopts new technologies. Let's see how two of these, `Ant` and IDEs, relate to `make`.

### Ant

The Java community is very active, producing new tools and APIs at an impressive rate. One of these new tools is `Ant`, a build tool intended to replace `make` in the Java development process. Like `make`, `Ant` uses a description file to indicate the targets and prerequisites of a project. Unlike `make`, `Ant` is written in Java and `Ant` build files are written in XML.

To give you a feel for the XML build file, here is an excerpt from the `Ant` build file:

```
<target name="build"
       depends="prepare, check_for_optional_packages"
       description="--> compiles the source code">
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${build.classes}"/>
  <mkdir dir="${build.lib}"/>

  <javac srcdir="${java.dir}"
         destdir="${build.classes}"
         debug="${debug}"
         deprecation="${deprecation}"
         target="${javac.target}"
         optimize="${optimize}" >
    <classpath refid="classpath"/>
  </javac>

  …

  <copy todir="${build.classes}">
    <fileset dir="${java.dir}">
```

```
        <include name="**/*.properties"/>
        <include name="**/*.dtd"/>
      </fileset>
    </copy>
  </target>
```

As you can see, a target is introduced with an XML `<target>` tag. Each target has a name and dependency list specified with `<name>` and `<depends>` attributes, respectively. Actions are performed by `Ant` *tasks*. A task is written in Java and bound to an XML tag. For instance, the task of creating a directory is specified with the `<mkdir>` tag and triggers the execution of the Java method `Mkdir.execute`, which eventually calls `File.mkdir`. As far as possible, all tasks are implemented using the Java API.

An equivalent build file using `make` syntax would be:

```
# compiles the source code
build: $(all_javas) prepare check_for_optional_packages
        $(MKDIR) -p $(build.dir) $(build.classes) $(build.lib)
        $(JAVAC) -sourcepath $(java.dir)                      \
                 -d $(build.classes)                          \
                 $(debug)                                     \
                 $(deprecation)                               \
                 -target $(javac.target)                      \
                 $(optimize)                                  \
                 -classpath $(classpath)                      \
                 @$<
        …
        $(FIND) . \( -name '*.properties' -o -name '*.dtd' \) | \
        $(TAR) -c -f - -T - | $(TAR) -C $(build.classes) -x -f -
```

This snippet of `make` uses techniques that this book hasn't discussed yet. Suffice to say that the prerequisite *all.javas* contains a list of all `java` files to be compiled. The `Ant` tasks `<mkdir>`, `<javac>`, and `<copy>` also perform dependency checking. That is, if the directory already exists, `mkdir` is not executed. Likewise, if the Java class files are newer than the source files, the source files are not compiled. Nevertheless, the `make` command script performs essentially the same functions. `Ant` includes a generic task, called `<exec>`, to run a local program.

`Ant` is a clever and fresh approach to build tools; however, it presents some issues worth considering:

- Although `Ant` has found wide acceptance in the Java community, it is still relatively unknown elsewhere. Also, it seems doubtful that its popularity will spread much beyond Java (for the reasons listed here). `make`, on the other hand, has consistently been applied to a broad range of fields including software development, document processing and typesetting, and web site and workstation maintenance, to name a few. Understanding `make` is important for anyone who needs to work on a variety of software systems.
- The choice of XML as the description language is appropriate for a Java-based tool. But XML is not particularly pleasant to write or to read (for many). Good

XML editors can be difficult to find and often do not integrate well with existing tools (either my integrated development environment includes a good XML editor or I must leave my IDE and find a separate tool). As you can see from the previous example, XML and the `Ant` dialect, in particular, are verbose compared with `make` and shell syntax. And the XML is filled with its own idiosyncrasies.

- When writing `Ant` build files you must contend with another layer of indirection. The `Ant <mkdir>` task does not invoke the underlying `mkdir` program for your system. Instead, it executes the Java `mkdir( )` method of the `java.io.File` class. This may or may not do what you expect. Essentially, any knowledge a programmer brings to `Ant` about the behavior of common tools is suspect and must be checked against the `Ant` documentation, Java documentation, or the `Ant` source. In addition, to invoke the Java compiler, for instance, I may be forced to navigate through a dozen or more unfamiliar XML attributes, such as `<srcdir>`, `<debug>`, etc., that are not documented in the compiler manual. In contrast, the `make` script is completely transparent, that is, I can typically type the commands directly into a shell to see how they behave.
- Although `Ant` is certainly portable, so is `make`. As shown in Chapter 7, writing portable *makefile*s, like writing portable `Ant` files, requires experience and knowledge. Programmers have been writing portable *makefile*s for two decades. Furthermore, the `Ant` documentation notes that there are portability issues with symbolic links on Unix and long filenames on Windows, that MacOS X is the only supported Apple operating system, and that support for other platforms is not guaranteed. Also, basic operations like setting the execution bit on a file cannot be performed from the Java API. An external program must be used. Portability is never easy or complete.
- The `Ant` tool does not explain precisely what it is doing. Since `Ant` tasks are not generally implemented by executing shell commands, the `Ant` tool has a difficult time displaying its actions. Typically, the display consists of natural language prose from `print` statements added by the task author. These print statements cannot be executed by a user from a shell. In contrast, the lines echoed by `make` are usually command lines that a user can copy and paste into a shell for reexecution. This means the `Ant` build is less useful to developers trying to understand the build process and tools. Also, it is not possible for a developer to reuse parts of a task, impromptu, at the keyboard.
- Last and most importantly, `Ant` shifts the build paradigm from a scripted to a nonscripted programming language. `Ant` tasks are written in Java. If a task does not exist or does not do what you want, you must either write your own task in Java or use the `<exec>` task. (Of course, if you use the `<exec>` task often, you would do far better to simply use `make` with its macros, functions, and more compact syntax.)

> Scripting languages, on the other hand, were invented and flourish precisely to address this type of issue. make has existed for nearly 30 years and can be used in the most complex situations without extending its implementation. Of course, there have been a handful of extensions in those 30 years. Many of them conceived and implemented in GNU make.

Ant is a marvelous tool that is widely accepted in the Java community. However, before embarking on a new project, consider carefully if Ant is appropriate for your development environment. This chapter will hopefully prove to you that make can powerfully meet your Java build needs.

## IDEs

Many Java developers use Integrated Development Environments (IDEs) that bundle an editor, compiler, debugger, and code browser in a single (typically) graphical environment. Examples include the open source Eclipse (*http://www.eclipse.org*) and Emacs JDEE (*http://jdee.sunsite.dk*), and, from commercial vendors, Sun Java Studio (*http://wwws.sun.com/software/sundev/jde*) and JBuilder (*http://www.borland.com/jbuilder*). These environments typically have the notion of a project-build process that compiles the necessary files and enables the application execution.

If the IDEs support all this, why should we consider using make? The most obvious reason is portability. If there is ever a need to build the project on another platform, the build may fail when ported to the new target. Although Java itself is portable across platforms, the support tools are often not. For instance, if the configuration files for your project include Unix- or Windows-style paths, these may generate errors when the build is run on the other operating system. A second reason to use make is to support unattended builds. Some IDEs support batch building and some do not. The quality of support for this feature also varies. Finally, the build support included is often limited. If you hope to implement customized release directory structures, integrate help files from external applications, support automated testing, and handle branching and parallel lines of development, you may find the integrated build support inadequate.

In my own experience, I have found the IDEs to be fine for small scale or localized development, but production builds require the more comprehensive support that make can provide. I typically use an IDE to write and debug code, and write a *makefile* for production builds and releases. During development I use the IDE to compile the project to a state suitable for debugging. But if I change many files or modify files that are input to code generators, then I run the *makefile*. The IDEs I've used do not have good support for external source code generation tools. Usually the result of an IDE build is not suitable for release to internal or external customers. For that task I use make.

## A Generic Java Makefile

Example 9-1 shows a generic *makefile* for Java; I'll explain each of its parts later in the chapter.

*Example 9-1. Generic makefile for Java*

```
# A generic makefile for a Java project.

VERSION_NUMBER := 1.0

# Location of trees.
SOURCE_DIR  := src
OUTPUT_DIR  := classes

# Unix tools
AWK         := awk
FIND        := /bin/find
MKDIR       := mkdir -p
RM          := rm -rf
SHELL       := /bin/bash

# Path to support tools
JAVA_HOME   := /opt/j2sdk1.4.2_03
AXIS_HOME   := /opt/axis-1_1
TOMCAT_HOME := /opt/jakarta-tomcat-5.0.18
XERCES_HOME := /opt/xerces-1_4_4
JUNIT_HOME  := /opt/junit3.8.1

# Java tools
JAVA        := $(JAVA_HOME)/bin/java
JAVAC       := $(JAVA_HOME)/bin/javac

JFLAGS      := -sourcepath $(SOURCE_DIR)        \
               -d $(OUTPUT_DIR)                 \
               -source 1.4

JVMFLAGS    := -ea                              \
               -esa                             \
               -Xfuture

JVM         := $(JAVA) $(JVMFLAGS)

JAR         := $(JAVA_HOME)/bin/jar
JARFLAGS    := cf

JAVADOC     := $(JAVA_HOME)/bin/javadoc
JDFLAGS     := -sourcepath $(SOURCE_DIR)        \
               -d $(OUTPUT_DIR)                 \
               -link http://java.sun.com/products/jdk/1.4/docs/api

# Jars
COMMONS_LOGGING_JAR   := $(AXIS_HOME)/lib/commons-logging.jar
```

*Example 9-1. Generic makefile for Java (continued)*

```
LOG4J_JAR            := $(AXIS_HOME)/lib/log4j-1.2.8.jar
XERCES_JAR           := $(XERCES_HOME)/xerces.jar
JUNIT_JAR            := $(JUNIT_HOME)/junit.jar

# Set the Java classpath
class_path := OUTPUT_DIR                \
              XERCES_JAR                \
              COMMONS_LOGGING_JAR       \
              LOG4J_JAR                 \
              JUNIT_JAR

# space - A blank space
space := $(empty) $(empty)

# $(call build-classpath, variable-list)
define build-classpath
$(strip                                         \
  $(patsubst :%,%,                              \
    $(subst : ,:,                               \
      $(strip                                   \
        $(foreach j,$1,$(call get-file,$j):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                       \
    $($1)                                       \
    $(if $(call file-exists-eval,$1),,          \
      $(warning The file referenced by variable \
                '$1' ($($1)) cannot be found)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip                                       \
    $(if $($1),,$(warning '$1' has no value))   \
    $(wildcard $($1)))

# $(call brief-help, makefile)
define brief-help
  $(AWK) '$$1 ~ /^[^.][-A-Za-z0-9]*:/                   \
         { print substr($$1, 1, length($$1)-1) }' $1 |  \
  sort |                                                \
  pr -T -w 80 -4
endef

# $(call file-exists, wildcard-pattern)
file-exists = $(wildcard $1)

# $(call check-file, file-list)
define check-file
  $(foreach f, $1,                              \
```

*Example 9-1. Generic makefile for Java (continued)*

```
    $(if $(call file-exists, $($f)),,           \
      $(warning $f ($($f)) is missing)))
endef

# #(call make-temp-dir, root-opt)
define make-temp-dir
  mktemp -t $(if $1,$1,make).XXXXXXXXXX
endef

# MANIFEST_TEMPLATE - Manifest input to m4 macro processor
MANIFEST_TEMPLATE := src/manifest/manifest.mf
TMP_JAR_DIR       := $(call make-temp-dir)
TMP_MANIFEST      := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"                       \
     --define=IMPL_VERSION=$(VERSION_NUMBER)            \
     --define=SPEC_VERSION=$(VERSION_NUMBER)            \
     $(if $3,$3,$(MANIFEST_TEMPLATE))                   \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef

# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
        cd $(OUTPUT_DIR); \
        $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
        $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef

# Set the CLASSPATH
export CLASSPATH := $(call build-classpath, $(class_path))

# make-directories - Ensure output directory exists.
make-directories := $(shell $(MKDIR) $(OUTPUT_DIR))

# help - The default goal
.PHONY: help
help:
        @$(call brief-help, $(CURDIR)/Makefile)

# all - Perform all tasks for a complete build
.PHONY: all
all: compile jars javadoc
```

*Example 9-1. Generic makefile for Java (continued)*

```
# all_javas - Temp file for holding source file list
all_javas := $(OUTPUT_DIR)/all.javas

# compile - Compile the source
.PHONY: compile
compile: $(all_javas)
	$(JAVAC) $(JFLAGS) @$<

# all_javas - Gather source file list
.INTERMEDIATE: $(all_javas)
$(all_javas):
	$(FIND) $(SOURCE_DIR) -name '*.java' > $@

# jar_list - List of all jars to create
jar_list := server_jar ui_jar

# jars - Create all jars
.PHONY: jars
jars: $(jar_list)

# server_jar - Create the $(server_jar)
server_jar_name     := $(OUTPUT_DIR)/lib/a.jar
server_jar_manifest := src/com/company/manifest/foo.mf
server_jar_packages := com/company/m com/company/n

# ui_jar - create the $(ui_jar)
ui_jar_name     := $(OUTPUT_DIR)/lib/b.jar
ui_jar_manifest := src/com/company/manifest/bar.mf
ui_jar_packages := com/company/o com/company/p

# Create an explicit rule for each jar
# $(foreach j, $(jar_list), $(eval $(call make-jar,$j)))
$(eval $(call make-jar,server_jar))
$(eval $(call make-jar,ui_jar))

# javadoc - Generate the Java doc from sources
.PHONY: javadoc
javadoc: $(all_javas)
	$(JAVADOC) $(JDFLAGS) @$<

.PHONY: clean
clean:
	$(RM) $(OUTPUT_DIR)

.PHONY: classpath
classpath:
	@echo CLASSPATH='$(CLASSPATH)'

.PHONY: check-config
check-config:
	@echo Checking configuration...
	$(call check-file, $(class_path) JAVA_HOME)
```

*Example 9-1. Generic makefile for Java (continued)*

```
.PHONY: print
print:
	$(foreach v, $(V), \
	  $(warning $v = $($v)))
```

## Compiling Java

Java can be compiled with make in two ways: the traditional approach, one javac execution per source file; or the fast approach outlined previously using the @filename syntax.

### The Fast Approach: All-in-One Compile

Let's start with the fast approach. As you can see in the generic *makefile*:

```
    # all_javas - Temp file for holding source file list
    all_javas := $(OUTPUT_DIR)/all.javas

    # compile - Compile the source
    .PHONY: compile
    compile: $(all_javas)
            $(JAVAC) $(JFLAGS) @$<

    # all_javas - Gather source file list
    .INTERMEDIATE: $(all_javas)
    $(all_javas):
            $(FIND) $(SOURCE_DIR) -name '*.java' > $@
```

The phony target compile invokes javac once to compile all the source of the project.

The $(all_javas) prerequisite is a file, *all.javas*, containing a list of Java files, one filename per line. It is not necessary for each file to be on its own line, but this way it is much easier to filter files with grep -v if the need ever arises. The rule to create *all.javas* is marked .INTERMEDIATE so that make will remove the file after each run and thus create a new one before each compile. The command script to create the file is straightforward. For maximum maintainability we use the find command to retrieve all the java files in the source tree. This command can be a bit slow, but is guaranteed to work correctly with virtually no modification as the source tree changes.

If you have a list of source directories readily available in the *makefile*, you can use faster command scripts to build *all.javas*. If the list of source directories is of medium length so that the length of the command line does not exceed the operating system's limits, this simple script will do:

```
    $(all_javas):
            shopt -s nullglob; \
            printf "%s\n" $(addsuffix /*.java,$(PACKAGE_DIRS)) > $@
```

This script uses shell wildcards to determine the list of Java files in each directory. If, however, a directory contains no Java files, we want the wildcard to yield the empty string, not the original globbing pattern (the default behavior of many shells). To achieve this effect, we use the `bash` option `shopt -s nullglob`. Most other shells have similar options. Finally, we use globbing and `printf` rather than `ls -1` because these are built-in to `bash`, so our command script executes only a single program regardless of the number of package directories.

Alternately, we can avoid shell globbing by using `wildcard`:

```
$(all_javas):
        print "%s\n" $(wildcard \
                       $(addsuffix /*.java,$(PACKAGE_DIRS))) > $@
```

If you have very many source directories (or very long paths), the above script may exceed the command-line length limit of the operating system. In that case, the following script may be preferable:

```
.INTERMEDIATE: $(all_javas)
$(all_javas):
        shopt -s nullglob;            \
        for f in $(PACKAGE_DIRS);     \
        do                            \
          printf "%s\n" $$f/*.java;   \
        done > $@
```

Notice that the `compile` target and the supporting rule follow the nonrecursive `make` approach. No matter how many subdirectories there are, we still have one *makefile* and one execution of the compiler. If you want to compile all of the source, this is as fast as it gets.

Also, we completely discarded all dependency information. With these rules, `make` neither knows nor cares about which file is newer than which. It simply compiles everything on every invocation. As an added benefit, we can execute the *makefile* from the source tree, instead of the binary tree. This may seem like a silly way to organize the *makefile* considering `make`'s abilities to manage dependencies, but consider this:

- The alternative (which we will explore shortly) uses the standard dependency approach. This invokes a new `javac` process for each file, adding a lot of overhead. But, if the project is small, compiling all the source files will not take significantly longer than compiling a few files because the `javac` compiler is so fast and process creation is typically slow. Any build that takes less than 15 seconds is basically equivalent regardless of how much work it does. For instance, compiling approximately 500 source files (from the Ant distribution) takes 14 seconds on my 1.8-GHz Pentium 4 with 512 MB of RAM. Compiling one file takes five seconds.
- Most developers will be using some kind of development environment that provides fast compilation for individual files. The *makefile* will most likely be used

when changes are more extensive, complete rebuilds are required, or unattended builds are necessary.

- As we shall see, the effort involved in implementing and maintaining dependencies is equal to the separate source and binary tree builds for C/C++ (described in Chapter 8). Not a task to be underestimated.

As we will see in later examples, the `PACKAGE_DIRS` variable has uses other than simply building the *all.javas* file. But maintaining this variables can be a labor-intensive, and potentially difficult, step. For smaller projects, the list of directories can be maintained by hand in the *makefile*, but when the number grows beyond a hundred directories, hand editing becomes error-prone and irksome. At this point, it might be prudent to use `find` to scan for these directories:

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs =                       \
  $(patsubst %/,%,                            \
    $(sort                                    \
      $(dir                                   \
        $(shell $(FIND) $1 -name '*.java'))))

PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

The `find` command returns a list of files, `dir` discards the file leaving only the directory, `sort` removes duplicates from the list, and `patsubst` strips the trailing slash. Notice that `find-compilation-dirs` finds the list of files to compile, only to discard the filenames, then the *all.javas* rule uses wildcards to restore the filenames. This seems wasteful, but I have often found that a list of the packages containing source code is very useful in other parts of the build, for instance to scan for EJB configuration files. If your situation does not require a list of packages, then by all means use one of the simpler methods previously mentioned to build *all.javas*.

## Compiling with Dependencies

To compile with full dependency checking, you first need a tool to extract dependency information from the Java source files, something similar to `cc -M`. Jikes (*http://www.ibm.com/developerworks/opensource/jikes*) is an open source Java compiler that supports this feature with the `-makefile` or `+M` option. Jikes is not ideal for separate source and binary compilation because it always writes the dependency file in the same directory as the source file, but it is freely available and it works. On the plus side, it generates the dependency file while compiling, avoiding a separate pass.

Here is a dependency processing function and a rule to use it:

```
%.class: %.java
        $(JAVAC) $(JFLAGS) +M $<
        $(call java-process-depend,$<,$@)

# $(call java-process-depend, source-file, object-file)
define java-process-depend
```

```
  $(SED) -e 's/^.*\.class *:/$2 $(subst .class,.d,$2):/'   \
         $(subst .java,.u,$1) > $(subst .class,.tmp,$2)
  $(SED) -e 's/#.*//'                                      \
         -e 's/^[^:]*: *//'                                \
         -e 's/ *\\$$$$//'                                 \
         -e '/^$$$$/ d'                                    \
         -e 's/$$$$/ :/' $(subst .class,.tmp,$2)           \
         >>  $(subst .class,.tmp,$2)
  $(MV) $(subst .class,.tmp,$2).tmp  $(subst .class,.d,$2)
endef
```

This requires that the *makefile* be executed from the binary tree and that the `vpath` be set to find the source. If you want to use the Jikes compiler only for dependency generation, resorting to a different compiler for actual code generation, you can use the `+B` option to prevent Jikes from generating bytecodes.

In a simple timing test compiling 223 Java files, the single line compile described previously as the fast approach required 9.9 seconds on my machine. The same 223 files compiled with individual compilation lines required 411.6 seconds or 41.5 times longer. Furthermore, with separate compilation, any build that required compiling more than four files was slower than compiling all the source files with a single compile line. If the dependency generation and compilation were performed by separate programs, the discrepancy would increase.

Of course, development environments vary, but it is important to carefully consider your goals. Minimizing the number of files compiled will not always minimize the time it takes to build a system. For Java in particular, full dependency checking and minimizing the number of files compiled does not appear to be necessary for normal program development.

## Setting CLASSPATH

One of the most important issues when developing software with Java is setting the `CLASSPATH` variable correctly. This variable determines which code is loaded when a class reference is resolved. To compile a Java application correctly, the *makefile* must include the proper `CLASSPATH`. The `CLASSPATH` can quickly become long and complex as Java packages, APIs, and support tools are added to a system. If the `CLASSPATH` can be difficult to set properly, it makes sense to set it in one place.

A technique I've found useful is to use the *makefile* to set the `CLASSPATH` for itself and other programs. For instance, a target `classpath` can return the `CLASSPATH` to the shell invoking the *makefile*:

```
.PHONY: classpath
classpath:
        @echo "export CLASSPATH='$(CLASSPATH)'"
```

Developers can set their `CLASSPATH` with this (if they use `bash`):

```
$ eval $(make classpath)
```

The CLASSPATH in the Windows environment can be set with this invocation:

```
.PHONY: windows_classpath
windows_classpath:
        regtool set /user/Environment/CLASSPATH "$(subst /,\\,$(CLASSPATH))"
        control sysdm.cpl,@1,3 &
        @echo "Now click Environment Variables, then OK, then OK again."
```

The program regtool is a utility in the Cygwin development system that manipulates the Windows Registry. Simply setting the Registry doesn't cause the new values to be read by Windows, however. One way to do this is to visit the Environment Variable dialog box and simply exit by clicking OK.

The second line of the command script causes Windows to display the System Properties dialog box with the Advanced tab active. Unfortunately, the command cannot display the Environment Variables dialog box or activate the OK button, so the last line prompts the user to complete the task.

Exporting the CLASSPATH to other programs, such as Emacs JDEE or JBuilder project files, is not difficult.

Setting the CLASSPATH itself can also be managed by make. It is certainly reasonable to set the CLASSPATH variable in the obvious way with:

```
CLASSPATH = /third_party/toplink-2.5/TopLink.jar:/third_party/…
```

For maintainability, using variables is preferred:

```
CLASSPATH = $(TOPLINK_25_JAR):$(TOPLINKX_25_JAR):…
```

But we can do better than this. As you can see in the generic *makefile*, we can build the CLASSPATH in two stages: first list the elements in the path as make variables, then transform those variables into the string value of the environment variable:

```
# Set the Java classpath
class_path := OUTPUT_DIR                \
              XERCES_JAR                \
              COMMONS_LOGGING_JAR       \
              LOG4J_JAR                 \
              JUNIT_JAR
…
# Set the CLASSPATH
export CLASSPATH := $(call build-classpath, $(class_path))
```

(The CLASSPATH in Example 9-1 is meant to be more illustrative than useful.) A well-written build-classpath function solves several irritating problems:

- It is very easy to compose a CLASSPATH in pieces. For instance, if different applications servers are used, the CLASSPATH might need to change. The different versions of the CLASSPATH could then be enclosed in ifdef sections and selected by setting a make variable.
- Casual maintainers of the *makefile* do not have to worry about embedded blanks, newlines, or line continuation, because the build-classpath function handles them.

- The path separator can be selected automatically by the build-classpath function. Thus, it is correct whether run on Unix or Windows.
- The validity of path elements can be verified by the build-classpath function. In particular, one irritating problem with make is that undefined variables collapse to the empty string without an error. In most cases this is very useful, but occasionally it gets in the way. In this case, it quietly yields a bogus value for the CLASSPATH variable.* We can solve this problem by having the build-classpath function check for the empty valued elements and warn us. The function can also check that each file or directory exists.
- Finally, having a hook to process the CLASSPATH can be useful for more advanced features, such as help accommodating embedded spaces in path names and search paths.

Here is an implementation of build-classpath that handles the first three issues:

```
# $(call build-classpath, variable-list)
define build-classpath
$(strip                                   \
  $(patsubst %:,%,                        \
    $(subst : ,:,                         \
      $(strip                             \
        $(foreach c,$1,$(call get-file,$c):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                 \
    $($1)                                 \
    $(if $(call file-exists-eval,$1),,    \
      $(warning The file referenced by variable \
                '$1' ($($1)) cannot be found)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip                                 \
    $(if $($1),,$(warning '$1' has no value))   \
    $(wildcard $($1)))
endef
```

The build-classpath function iterates through the words in its argument, verifying each element and concatenating them with the path separator (: in this case). Selecting the path separator automatically is easy now. The function then strips spaces added by the get-file function and foreach loop. Next, it strips the final separator

* We could try using the --warn-undefined-variables option to identify this situation, but this also flags many other empty variables that are desirable.

added by the foreach loop. Finally, the whole thing is wrapped in a `strip` so errant spaces introduced by line continuation are removed.

The `get-file` function returns its filename argument, then tests whether the variable refers to an existing file. If it does not, it generates a warning. It returns the value of the variable regardless of the existence of the file because the value may be useful to the caller. On occasion, `get-file` may be used with a file that will be generated, but does not yet exist.

The last function, `file-exists-eval`, accepts a variable name containing a file reference. If the variable is empty, a warning is issued; otherwise, the `wildcard` function is used to resolve the value into a file (or a list of files for that matter).

When the `build-classpath` function is used with some suitable bogus values, we see these errors:

```
Makefile:37: The file referenced by variable 'TOPLINKX_25_JAR'
             (/usr/java/toplink-2.5/TopLinkX.jar) cannot be found
...
Makefile:37: 'XERCES_142_JAR' has no value
Makefile:37: The file referenced by variable
             'XERCES_142_JAR' ( ) cannot be found
```

This represents a great improvement over the silence we would get from the simple approach.

The existence of the `get-file` function suggests that we could generalize the search for input files.

```
# $(call get-jar, variable-name)
define get-jar
  $(strip                                                   \
    $(if $($1),,$(warning '$1' is empty))                   \
    $(if $(JAR_PATH),,$(warning JAR_PATH is empty))         \
    $(foreach d, $(dir $($1)) $(JAR_PATH),                  \
      $(if $(wildcard $d/$(notdir $($1))),                  \
        $(if $(get-jar-return),,                            \
          $(eval get-jar-return := $d/$(notdir $($1))))))   \
    $(if $(get-jar-return),                                 \
      $(get-jar-return)                                     \
      $(eval get-jar-return :=),                            \
      $($1)                                                 \
      $(warning get-jar: File not found '$1' in $(JAR_PATH))))
endef
```

Here we define the variable `JAR_PATH` to contain a search path for files. The first file found is returned. The parameter to the function is a variable name containing the path to a jar. We want to look for the jar file first in the path given by the variable, then in the `JAR_PATH`. To accomplish this, the directory list in the `foreach` loop is composed of the directory from the variable, followed by the `JAR_PATH`. The two other uses of the parameter are enclosed in `notdir` calls so the jar name can be composed from a path from this list. Notice that we cannot exit from a `foreach` loop.

Instead, therefore, we use eval to set a variable, get-jar-return, to remember the first file we found. After the loop, we return the value of our temporary variable or issue a warning if nothing was found. We must remember to reset our return value variable before terminating the macro.

This is essentially reimplementing the vpath feature in the context of setting the CLASSPATH. To understand this, recall that the vpath is a search path used implicitly by make to find prerequisites that cannot be found from the current directory by a relative path. In these cases, make searches the vpath for the prerequisite file and inserts the completed path into the $^, $?, and $+ automatic variables. To set the CLASSPATH, we want make to search a path for each jar file and insert the completed path into the CLASSPATH variable. Since make has no built-in support for this, we've added our own. You could, of course, simply expand the jar path variable with the appropriate jar filenames and let Java do the searching, but CLASSPATHs already get long quickly. On some operating systems, environment variable space is limited and long CLASSPATHs are in danger of being truncated. On Windows XP, there is a limit of 1023 characters for a single environment variable. In addition, even if the CLASSPATH is not truncated, the Java virtual machine must search the CLASSPATH when loading classes, thus slowing down the application.

## Managing Jars

Building and managing jars in Java presents different issues from C/C++ libraries. There are three reasons for this. First, the members of a jar include a relative path, so the precise filenames passed to the jar program must be carefully controlled. Second, in Java there is a tendency to merge jars so that a single jar can be released to represent a program. Finally, jars include other files than classes, such as manifests, property files, and XML.

The basic command to create a jar in GNU make is:

```
JAR      := jar
JARFLAGS := -cf

$(FOO_JAR): prerequisites…
        $(JAR) $(JARFLAGS) $@ $^
```

The jar program can accept directories instead of filenames, in which case, all the files in the directory trees are included in the jar. This can be very convenient, especially when used with the -C option for changing directories:

```
JAR      := jar
JARFLAGS := -cf

.PHONY: $(FOO_JAR)
$(FOO_JAR):
        $(JAR) $(JARFLAGS) $@ -C $(OUTPUT_DIR) com
```

Here the jar itself is declared `.PHONY`. Otherwise subsequent runs of the *makefile* would not recreate the file, because it has no prerequisites. As with the `ar` command described in an earlier chapter, there seems little point in using the update flag, `-u`, since it takes the same amount of time or longer as recreating the jar from scratch, at least for most updates.

A jar often includes a manifest that identifies the vendor, API and version number the jar implements. A simple manifest might look like:

```
Name: JAR_NAME
Specification-Title: SPEC_NAME
Implementation-Version: IMPL_VERSION
Specification-Vendor: Generic Innovative Company, Inc.
```

This manifest includes three placeholders, `JAR_NAME`, `SPEC_NAME`, and `IMPL_VERSION`, that can be replaced at jar creation time by `make` using `sed`, `m4`, or your favorite stream editor. Here is a function to process a manifest:

```
MANIFEST_TEMPLATE := src/manifests/default.mf
TMP_JAR_DIR       := $(call make-temp-dir)
TMP_MANIFEST      := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"                      \
     --define=IMPL_VERSION=$(VERSION_NUMBER)           \
     --define=SPEC_VERSION=$(VERSION_NUMBER)           \
     $(if $3,$3,$(MANIFEST_TEMPLATE))                  \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef
```

The `add-manifest` function operates on a manifest file similar to the one shown previously. The function first creates a temporary directory, then expands the sample manifest. Next, it updates the jar, and finally deletes the temporary directory. Notice that the last parameter to the function is optional. If the manifest file path is empty, the function uses the value from `MANIFEST_TEMPLATE`.

The generic *makefile* bundles these operations into a generic function to write an explicit rule for creating a jar:

```
# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
        cd $(OUTPUT_DIR); \
        $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
        $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef
```

It accepts a single argument, the prefix of a make variable, that identifies a set of variables describing four jar parameters: the target name, the jar name, the packages in the jar, and the jar's manifest file. For example, for a jar named *ui.jar*, we would write:

```
ui_jar_name     := $(OUTPUT_DIR)/lib/ui.jar
ui_jar_manifest := src/com/company/ui/manifest.mf
ui_jar_packages := src/com/company/ui \
                   src/com/company/lib

$(eval $(call make-jar,ui_jar))
```

By using variable name composition, we can shorten the calling sequence of our function and allow for a very flexible implementation of the function.

If we have many jar files to create, we can automate this further by placing the jar names in a variable:

```
jar_list := server_jar ui_jar

.PHONY: jars $(jar_list)
jars: $(jar_list)

$(foreach j, $(jar_list),\
  $(eval $(call make-jar,$j)))
```

Occasionally, we need to expand a jar file into a temporary directory. Here is a simple function to do that:

```
# $(call burst-jar, jar-file, target-directory)
define burst-jar
  $(call make-dir,$2)
  cd $2; $(JAR) -xf $1
endef
```

## Reference Trees and Third-Party Jars

To use a single, shared reference tree to support partial source trees for developers, simply have the nightly build create jars for the project and include those jars in the CLASSPATH of the Java compiler. The developer can check out the parts of the source tree he needs and run the compile (assuming the source file list is dynamically created by something like find). When the Java compiler requires symbols from a missing source file, it will search the CLASSPATH and discover the *.class* file in the jar.

Selecting third-party jars from a reference tree is also simple. Just place the path to the jar in the CLASSPATH. The *makefile* can be a valuable tool for managing this process as previously noted. Of course, the get-file function can be used to automatically select beta or stable, local or remote jars by simply setting the JAR_PATH variable.

## Enterprise JavaBeans

Enterprise JavaBeans™ is a powerful technique to encapsulate and reuse business logic in the framework of remote method invocation. EJB sets up Java classes used to implement server APIs that are ultimately used by remote clients. These objects and services are configured using XML-based control files. Once the Java classes and XML control files are written, they must be bundled together in a jar. Then a special EJB compiler builds stubs and ties to implement the RPC support code.

The following code can be plugged into Example 9-1 to provide generic EJB support:

```
EJB_TMP_JAR = $(EJB_TMP_DIR)/temp.jar
META_INF    = $(EJB_TMP_DIR)/META-INF

# $(call compile-bean, jar-name,
#                      bean-files-wildcard, manifest-name-opt)
define compile-bean
  $(eval EJB_TMP_DIR := $(shell mktemp -d $(TMPDIR)/compile-bean.XXXXXXXX))
  $(MKDIR) $(META_INF)
  $(if $(filter %.xml, $2),cp $(filter %.xml, $2) $(META_INF))
  cd $(OUTPUT_DIR) &&                          \
  $(JAR) -cf0 $(EJB_TMP_JAR)                   \
         $(call jar-file-arg,$(META_INF))      \
         $(filter-out %.xml, $2)
  $(JVM) weblogic.ejbc $(EJB_TMP_JAR) $1
  $(call add-manifest,$(if $3,$3,$1),,)
  $(RM) $(EJB_TMP_DIR)
endef

# $(call jar-file-arg, jar-file)
jar-file-arg = -C "$(patsubst %/,%,$(dir $1))" $(notdir $1)
```

The `compile-bean` function comaccepts three parameters: the name of the jar to create, the list of files in the jar, and an optional manifest file. The function first creates a clean temporary directory using the `mktemp` program and saves the directory name in the variable `EJB_TMP_DIR`. By embedding the assignment in an `eval`, we ensure that `EJB_TMP_DIR` is reset to a new temporary directory once for each expansion of `compile-bean`. Since `compile-bean` is used in the command script part of a rule, the function is expanded only when the command script is executed. Next, it copies any XML files in the bean file list into the *META-INF* directory. This is where EJB configuration files live. Then, the function builds a temporary jar that is used as input to the EJB compiler. The `jar-file-arg` function converts filenames of the form *dir1/dir2/dir3* into `-C dir1/dir2 dir3` so the relative path to the file in the jar is correct. This is the appropriate format for indicating the *META-INF* directory to the jar command. The bean file list contains *.xml* files that have already been placed in the *META-INF* directory, so we filter these files out. After building the temporary jar, the WebLogic EJB compiler is invoked, generating the output jar. A manifest is then added to the compiled jar. Finally, our temporary directory is removed.

Using the new function is straightforward:

```
bean_files = com/company/bean/FooInterface.class       \
             com/company/bean/FooHome.class            \
             src/com/company/bean/ejb-jar.xml          \
             src/com/company/bean/weblogic-ejb-jar.xml

.PHONY: ejb_jar $(EJB_JAR)
ejb_jar: $(EJB_JAR)
$(EJB_JAR):
        $(call compile-bean, $@, $(bean_files), weblogic.mf)
```

The bean_files list is a little confusing. The *.class* files it references will be accessed relative to the *classes* directory, while the *.xml* files will be accessed relative to the directory of the *makefile*.

This is fine, but what if you have lots of bean files in your bean jar. Can we build the file list automatically? Certainly:

```
src_dirs := $(SOURCE_DIR)/com/company/...

bean_files =                                          \
  $(patsubst $(SOURCE_DIR)/%,%,                       \
    $(addsuffix /*.class,                             \
      $(sort                                          \
        $(dir                                         \
          $(wildcard                                  \
            $(addsuffix /*Home.java,$(src_dirs)))))))

.PHONY: ejb_jar $(EJB_JAR)
ejb_jar: $(EJB_JAR)
$(EJB_JAR):
        $(call compile-bean, $@, $(bean_files), weblogic.mf)
```

This assumes that all the directories with EJB source are contained in the src_dirs variable (there can also be directories that do not contain EJB source) and that any file ending in *Home.java* identifies a package containing EJB code. The expression for setting the bean_files variable first adds the wildcard suffix to the directories, then invokes wildcard to gather the list of *Home.java* files. The filenames are discarded to leave the directories, which are sorted to remove duplicates. The wildcard /*.class suffix is added so that the shell will expand the list to the actual class files. Finally, the source directory prefix (which is not valid in the *classes* tree) is removed. Shell wildcard expansion is used instead of make's wildcard because we can't rely on make to perform its expansion after the class files have been compiled. If make evaluated the wildcard function too early it would find no files and directory caching would prevent it from ever looking again. The wildcard in the source tree is perfectly safe because (we assume) no source files will be added while make is running.

The above code works when we have a small number of bean jars. Another style of development places each EJB in its own jar. Large projects may have dozens of jars. To handle this case automatically, we need to generate an explicit rule for each EJB jar. In this example, EJB source code is self-contained: each EJB is located in a single directory with its associated XML files. EJB directories can be identified by files that end with *Session.java*.

The basic approach is to search the source tree for EJBs, then build an explicit rule to create each EJB and write these rules into a file. The EJB rules file is then included in our *makefile*. The creation of the EJB rules file is triggered by make's own dependency handling of include files.

```
# session_jars - The EJB jars with their relative source path.
session_jars =
  $(subst .java,.jar,                          \
    $(wildcard                                 \
      $(addsuffix /*Session.java, $(COMPILATION_DIRS))))

# EJBS - A list of all EJB jars we need to build.
EJBS = $(addprefix $(TMP_DIR)/,$(notdir $(session_jars)))

# ejbs - Create all EJB jar files.
.PHONY: ejbs
ejbs: $(EJBS)
$(EJBS):
        $(call compile-bean,$@,$^,)
```

We find the *Session.java* files by calling a wildcard on all the compilation directories. In this example, the jar file is the name of the Session file with the *.jar* suffix. The jars themselves will be placed in a temporary binary directory. The EJBS variable contains the list of jars with their binary directory path. These EJB jars are the targets we want to update. The actual command script is our compile-bean function. The tricky part is that the file list is recorded in the prerequisites for each jar file. Let's see how they are created.

```
-include $(OUTPUT_DIR)/ejb.d

# $(call ejb-rule, ejb-name)
ejb-rule = $(TMP_DIR)/$(notdir $1):          \
             $(addprefix $(OUTPUT_DIR)/,      \
               $(subst .java,.class,          \
                 $(wildcard $(dir $1)*.java))) \
             $(wildcard $(dir $1)*.xml)

# ejb.d - EJB dependencies file.
$(OUTPUT_DIR)/ejb.d: Makefile
        @echo Computing ejb dependencies...
        @for f in $(session_jars);           \
        do                                   \
          echo "\$$(call ejb-rule,$$f)";     \
        done > $@
```

The dependencies for each EJB jar are recorded in a separate file, *ejb.d*, that is included by the *makefile*. The first time make looks for this include file it does not exist. So make invokes the rule for updating the include file. This rule writes one line for each EJB, something like:

```
$(call ejb-rule,src/com/company/foo/FooSession.jar)
```

The function ejb-rule will expand to the target jar and its list of prerequisites, something like:

```
classes/lib/FooSession.jar: classes/com/company/foo/FooHome.jar \
            classes/com/company/foo/FooInterface.jar          \
            classes/com/company/foo/FooSession.jar            \
            src/com/company/foo/ejb-jar.xml                   \
            src/com/company/foo/ejb-weblogic-jar.xml
```

In this way, a large number of jars can be managed in make without incurring the overhead of maintaining a set of explicit rules by hand.

CHAPTER 10

# Improving the Performance of make

make plays a critical role in the development process. It combines the elements of a project to create an application while allowing the developer to avoid the subtle errors caused by accidentally omitting steps of the build. However, if developers avoid using make, because they feel the *makefile* is too slow, all the benefits of make are lost. It is important, therefore, to ensure that the *makefile* be crafted to be as efficient as possible.

Performance issues are always tricky, but become even more so when the perception of users and different paths through the code are considered. Not every target of a *makefile* is worth optimizing. Even radical optimizations might not be worth the effort depending on your environment. For instance, reducing the time of an operation from 90 minutes to 45 minutes may be immaterial since even the faster time is a "go get lunch" operation. On the other hand, reducing a task from 2 minutes to 1 might be received with cheers if developers are twiddling their thumbs during that time.

When writing a *makefile* for efficient execution, it is important to know the costs of various operations and to know what operations are being performed. In the following sections, we will perform some simple benchmarking to quantify these general comments and present techniques to help identify bottlenecks.

A complementary approach to improving performance is to take advantage of parallelism and local network topology. By running more than one command script at a time (even on a uniprocessor), build times can be reduced.

## Benchmarking

Here we measure the performance of some basic operations in make. Table 10-1 shows the results of these measurements. We'll explain each test and suggest how they might affect *makefile*s you write.

*Table 10-1. Cost of operations*

| Operation | Executions | Seconds per execution (Windows) | Executions per second (Windows) | Seconds per execution (Linux) | Executions per second (Linux) |
|---|---|---|---|---|---|
| `make` (bash) | 1000 | 0.0436 | 22 | 0.0162 | 61 |
| `make` (ash) | 1000 | 0.0413 | 24 | 0.0151 | 66 |
| `make` (sh) | 1000 | 0.0452 | 22 | 0.0159 | 62 |
| `assignment` | 10,000 | 0.0001 | 8130 | 0.0001 | 10,989 |
| `subst` (short) | 10,000 | 0.0003 | 3891 | 0.0003 | 3846 |
| `subst` (long) | 10,000 | 0.0018 | 547 | 0.0014 | 704 |
| `sed` (bash) | 1000 | 0.0910 | 10 | 0.0342 | 29 |
| `sed` (ash) | 1000 | 0.0699 | 14 | 0.0069 | 144 |
| `sed` (sh) | 1000 | 0.0911 | 10 | 0.0139 | 71 |
| `shell` (bash) | 1000 | 0.0398 | 25 | 0.0261 | 38 |
| `shell` (ash) | 1000 | 0.0253 | 39 | 0.0018 | 555 |
| `shell` (sh) | 1000 | 0.0399 | 25 | 0.0050 | 198 |

The Windows tests were run on a 1.9-GHz Pentium 4 (approximately 3578 BogoMips)[*] with 512 MB RAM running Windows XP. The Cygwin version of `make` 3.80 was used, started from an `rxvt` window. The Linux tests were run on a 450-MHz Pentium 2 (891 BogoMips) with 256 MB of RAM running Linux RedHat 9.

The subshell used by `make` can have a significant effect on the overall performance of the *makefile*. The `bash` shell is a complex, fully featured shell, and therefore large. The `ash` shell is a much smaller, with fewer features but adequate for most tasks. To complicate matters, if `bash` is invoked from the filename */bin/sh*, it alters its behavior significantly to conform more closely to the standard shell. On most Linux systems the file */bin/sh* is a symbolic link to `bash`, while in Cygwin */bin/sh* is really *ash*. To account for these differences, some of the tests were run three times, each time using a different shell. The shell used is indicated in parentheses. When "(sh)" appears, it means that `bash` was linked to the file named */bin/sh*.

The first three tests, labeled make, give an indication of how expensive it is to run `make` if there is nothing to do. The *makefile* contains:

```
SHELL := /bin/bash
.PHONY: x
x:
        $(MAKE) --no-print-directory --silent --question make-bash.mk; \
        …this command repeated 99 more times…
```

The word "bash" is replaced with the appropriate shell name as required.

[*] See *http://www.clifton.nl/bogomips.html* for an explanation of BogoMips.

We use the `--no-print-directory` and `--silent` commands to eliminate unnecessary computation that might skew the timing test and to avoid cluttering the timing output values with irrelevant text. The `--question` option tells `make` to simply check the dependencies without executing any commands and return an exit status of zero if the files are up to date. This allows `make` to do as little work as possible. No commands will be executed by this *makefile* and dependencies exist for only one `.PHONY` target. The command script executes `make` 100 times. This *makefile*, called *make-bash.mk*, is executed 10 times by a parent *makefile* with this code:

```
define ten-times
  TESTS += $1
  .PHONY: $1
  $1:
        @echo $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2
endef

.PHONY: all
all:

$(eval $(call ten-times, make-bash, -f make-bash.mk))

all: $(TESTS)
```

The time for these 1,000 executions is then averaged.

As you can see from the table, the Cygwin `make` ran at roughly 22 executions per second or 0.044 seconds per run, while the Linux version (even on a drastically slower CPU) performed roughly 61 executions per second or 0.016 seconds per run. To verify these results, the native Windows version of `make` was also tested and did not yield any dramatic speed up. Conclusion: while process creation in Cygwin `make` is slightly slower than a native Windows `make`, both are dramatically slower than Linux. It also suggests that use of recursive `make` on a Windows platform may perform significantly slower than the same build run on Linux.

As you would expect, the shell used in this test had no effect on execution time. Because the command script contained no shell special characters, the shell was not invoked at all. Rather, `make` executed the commands directly. This can be verified by setting the `SHELL` variable to a completely bogus value and noting that the test still runs correctly. The difference in performance between the three shells must be attributed to normal system variance.

The next benchmark measures the speed of variable assignment. This calibrates the most elementary make operation. The *makefile*, called *assign.mk*, contains:

```
# 10000 assignments
z := 10
…repeated 10000 times…
.PHONY: x
x: ;
```

This *makefile* is then run using our ten-times function in the parent *makefile*.

The assignment is obviously very fast. Cygwin make will execute 8130 assignments per second while the Linux system can do 10,989. I believe the performance of Windows for most of these operations is actually better than the benchmark indicates because the cost of creating the make process 10 times cannot be reliably factored out of the time. Conclusion: because it is unlikely that the average *makefile* would perform 10,000 assignments, the cost of variable assignment in an average *makefile* is negligible.

The next two benchmarks measure the cost of a subst function call. The first uses a short 10-character string with three substitutions:

```
# 10000 subst on a 10 char string
dir := ab/cd/ef/g
x := $(subst /, ,$(dir))
…repeated 10000 times…
.PHONY: x
x: ;
```

This operation takes roughly twice as long as a simple assignment, or 3891 operations per second on Windows. Again, the Linux system appears to outperform the Windows system by a wide margin. (Remember, the Linux system is running at less than one quarter the clock speed of the Windows system.)

The longer substitution operates on a 1000-character string with roughly 100 substitutions:

```
# Ten character file
dir := ab/cd/ef/g
# 1000 character path
p100 := $(dir);$(dir);$(dir);$(dir);$(dir);…
p1000 := $(p100)$(p100)$(p100)$(p100)$(p100)…

# 10000 subst on a 1000 char string
x := $(subst ;, ,$(p1000))
…repeated 10000 times…
.PHONY: x
x: ;
```

The next three benchmarks measure the speed of the same substitution using sed. The benchmark contains:

```
# 100 sed using bash
SHELL := /bin/bash
```

```
.PHONY: sed-bash
sed-bash:
        echo '$(p1000)' | sed 's/;/ /g' > /dev/null
        …repeated 100 times…
```

As usual, this *makefile* is executed using the ten-times function. On Windows, sed execution takes about 50 times longer than the subst function. On our Linux system, sed is only 24 times slower.

When we factor in the cost of the shell, we see that ash on Windows does provide a useful speed-up. With ash, the sed is only 39 times slower than subst! (wink) On Linux, the shell used has a much more profound effect. Using ash, the sed is only five times slower than subst. Here we also notice the curious effect of renaming bash to sh. On Cygwin, there is no difference between a bash named /bin/bash and one named /bin/sh, but on Linux, a bash linked to /bin/sh performs significantly better.

The final benchmark simply invokes the make shell command to evaluate the cost of running a subshell. The *makefile* contains:

```
# 100 $(shell ) using bash
SHELL := /bin/bash
x := $(shell :)
…repeated 100 times…
.PHONY: x
x: ;
```

There are no surprises here. The Windows system is slower than Linux, with ash having an edge over bash. The performance gain of ash is more pronounced—about 50% faster. The Linux system performs best with ash and slowest with bash (when named "bash").

Benchmarking is a never-ending task, however, the measurements we've made can provide some useful insight. Create as many variables as you like if they help clarify the structure of the *makefile* because they are essentially free. Built-in make functions are preferred over running commands even if you are required by the structure of your code to reexecute the make function repeatedly. Avoid recursive make or unnecessary process creation on Windows. While on Linux, use ash if you are creating many processes.

Finally, remember that in most *makefile*s, the time a *makefile* takes to run is due almost entirely to the cost of the programs run, not make or the structure of the *makefile*. Usually, reducing the number of programs run will be most helpful in reducing the execution time of a *makefile*.

## Identifying and Handling Bottlenecks

Unnecessary delays in *makefile*s come from several sources: poor structuring of the *makefile*, poor dependency analysis, and poor use of make functions and variables.

These problems can be masked by make functions such as shell that invoke commands without echoing them, making it difficult to find the source of the delay.

Dependency analysis is a two-edged sword. On the one hand, if complete dependency analysis is performed, the analysis itself may incur significant delays. Without special compiler support, such as supplied by gcc or jikes, creating a dependency file requires running another program, nearly doubling compilation time.* The advantage of complete dependency analysis is that it allows make to perform fewer compiles. Unfortunately, developers may not believe this benefit is realized and write *makefile*s with less complete dependency information. This compromise almost always leads to an increase in development problems, leading other developers to overcompensate by compiling more code than would be required with the original, complete dependency information.

To formulate a dependency analysis strategy, begin by understanding the dependencies inherent in the project. Once complete dependency information is understood, you can choose how much to represent in the *makefile* (computed or hardcoded) and what shortcuts can be taken during the build. Although none of this is exactly simple, it is straightforward.

Once you've determined your *makefile* structure and necessary dependencies, implementing an efficient *makefile* is usually a matter of avoiding some simple pitfalls.

## Simple Variables Versus Recursive

One of the most common performance-related problems is using recursive variables instead of simple variables. For example, because the following code uses the = operator instead of :=, it will execute the date command every time the DATE variable is used:

```
DATE = $(shell date +%F)
```

The +%F option instructs date to return the date in "yyyy-mm-dd" format, so for most users the repeated execution of date would never be noticed. Of course, developers working around midnight might get a surprise!

Because make doesn't echo commands executed from the shell function, it can be difficult to determine what is actually being run. By resetting the SHELL variable to /bin/sh -x, you can trick make into revealing all the commands it executes.

* In practice, compilation time grows linearly with the size of the input text and this time is almost always dominated by disk I/O. Similarly, the time to compute dependencies using the simple -M option is linear and bound by disk I/O.

This *makefile* creates its output directory before performing any actions. The name of the output directory is composed of the word "out" and the date:

```
DATE = $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p $(OUTPUT_DIR))

all: ;
```

When run with a debugging shell, we can see:

```
$ make SHELL='/bin/sh -x'
+ date +%F
+ date +%F
+ '[' -d out-2004-03-30 ']'
+ mkdir -p out-2004-03-30
make: all is up to date.
```

This clearly shows us that the `date` command was executed twice. If you need to perform this kind of shell trace often, you can make it easier to access with:

```
ifdef DEBUG_SHELL
  SHELL = /bin/sh -x
endif
```

## Disabling @

Another way commands are hidden is through the use of the silent command modifier, `@`. It can be useful at times to be able to disable this feature. You can make this easy by defining a variable, `QUIET`, to hold the @ sign and use the variable in commands:

```
ifndef VERBOSE
  QUIET := @
endif
…
target:
        $(QUIET) echo Building target...
```

When it becomes necessary to see commands hidden by the silent modifier, simply define `VERBOSE` on the command line:

```
$ make VERBOSE=1
echo Building target...
Building target...
```

## Lazy Initialization

When simple variables are used in conjunction with the `shell` function, `make` evaluates all the `shell` function calls as it reads the *makefile*. If there are many of these, or if they perform expensive computations, `make` can feel sluggish. The responsiveness of `make` can be measured by timing `make` when invoked with a nonexistent target:

```
$ time make no-such-target
make: *** No rule to make target no-such-target.  Stop.
```

```
real    0m0.058s
user    0m0.062s
sys     0m0.015s
```

This code times the overhead that make will add to any command executed, even trivial or erroneous commands.

Because recursive variables reevaluate their righthand side every time they are expanded, there is a tendency to express complex calculations as simple variables. However, this decreases the responsiveness of make for all targets. It seems that there is a need for another kind of variable, one whose righthand side is evaluated only once the first time the variable is evaluated, but not before.

An example illustrating the need for this type of initialization is the find-compilation-dirs function introduced in the section "The Fast Approach: All-in-One Compile" in Chapter 9:

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs =                      \
  $(patsubst %/,%,                           \
    $(sort                                   \
      $(dir                                  \
        $(shell $(FIND) $1 -name '*.java'))))

PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

Ideally, we would like to perform this find operation only once per execution, but only when the PACKAGE_DIRS variable is actually used. This might be called *lazy initialization*. We can build such a variable using eval like this:

```
PACKAGE_DIRS = $(redefine-package-dirs) $(PACKAGE_DIRS)

redefine-package-dirs = \
   $(eval PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR)))
```

The basic approach is to define PACKAGE_DIRS first as a recursive variable. When expanded, the variable evaluates the expensive function, here find-compilation-dirs, and redefines itself as a simple variable. Finally, the (now simple) variable value is returned from the original recursive variable definition.

Let's go over this in detail:

1. When make reads these variables, it simply records their righthand side because the variables are recursive.
2. The first time the PACKAGE_DIRS variable is used, make retrieves the righthand side and expands the first variable, redefine-package-dirs.
3. The value of redefine-package-dirs is a single function call, eval.
4. The body of the eval redefines the recursive variable, PACKAGE_DIRS, as a simple variable whose value is the set of directories returned by find-compilation-dirs. Now PACKAGE_DIRS has been initialized with the directory list.

5. The `redefine-package-dirs` variable is expanded to the empty string (because eval expands to the empty string).
6. Now `make` continues to expand the original righthand side of `PACKAGE_DIRS`. The only thing left to do is expand the variable `PACKAGE_DIRS`. `make` looks up the value of the variable, sees a simple variable, and returns its value.

The only really tricky part of this code is relying on `make` to evaluate the righthand side of a recursive variable from left to right. If, for instance, `make` decided to evaluate `$(PACKAGE_DIRS)` before `$(redefine-package-dirs)`, the code would fail.

The procedure I just described can be refactored into a function, `lazy-init`:

```
# $(call lazy-init,variable-name,value)
define lazy-init
  $1 = $$(redefine-$1) $$($1)
  redefine-$1 = $$(eval $1 := $2)
endef

# PACKAGE_DIRS - a lazy list of directories
$(eval                           \
  $(call lazy-init,PACKAGE_DIRS, \
    $$(call find-compilation-dirs,$(SOURCE_DIRS))))
```

## Parallel make

Another way to improve the performance of a build is to take advantage of the parallelism inherent in the problem the *makefile* is solving. Most *makefile*s perform many tasks that are easily carried out in parallel, such as compiling C source to object files or creating libraries out of object files. Furthermore, the very structure of a well-written *makefile* provides all the information necessary to automatically control the concurrent processes.

Example 10-1 shows our `mp3_player` program executed with the jobs option, `--jobs=2` (or `-j 2`). Figure 10-1 shows the same `make` run in a pseudo UML sequence diagram. Using `--jobs=2` tells `make` to update two targets in parallel when that is possible. When `make` updates targets in parallel, it echos commands in the order in which they are executed, interleaving them in the output. This can make reading the output from parallel `make` more difficult. Let's look at this output more carefully.

*Example 10-1. Output of make when --jobs = 2*

```
$ make -f ../ch07-separate-binaries/makefile --jobs=2

1  bison -y  --defines ../ch07-separate-binaries/lib/db/playlist.y

2  flex  -t ../ch07-separate-binaries/lib/db/scanner.l > lib/db/scanner.c

3  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
   ../ch07-separate-binaries/app/player/play_mp3.c | \
```

*Example 10-1. Output of make when --jobs = 2 (continued)*

```
    sed 's,\(play_mp3\.o\) *:,app/player/\1 app/player/play_mp3.d: ,' > app/player/play_
mp3.d.tmp

  4  mv -f y.tab.c lib/db/playlist.c

  5  mv -f y.tab.h lib/db/playlist.h

  6  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
     ../ch07-separate-binaries/lib/codec/codec.c | \
    sed 's,\(codec\.o\) *:,lib/codec/\1 lib/codec/codec.d: ,' > lib/codec/codec.d.tmp

  7  mv -f app/player/play_mp3.d.tmp app/player/play_mp3.d

  8  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
     lib/db/playlist.c | \
    sed 's,\(playlist\.o\) *:,lib/db/\1 lib/db/playlist.d: ,' > lib/db/playlist.d.tmp

  9  mv -f lib/codec/codec.d.tmp lib/codec/codec.d

 10  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
     ../ch07-separate-binaries/lib/ui/ui.c | \
    sed 's,\(ui\.o\) *:,lib/ui/\1 lib/ui/ui.d: ,' > lib/ui/ui.d.tmp

 11  mv -f lib/db/playlist.d.tmp lib/db/playlist.d

 12  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
     lib/db/scanner.c | \
    sed 's,\(scanner\.o\) *:,lib/db/\1 lib/db/scanner.d: ,' > lib/db/scanner.d.tmp

 13  mv -f lib/ui/ui.d.tmp lib/ui/ui.d

 14  mv -f lib/db/scanner.d.tmp lib/db/scanner.d

 15  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o app/player/play_mp3.o ../ch07-separate-binaries/app/player/play_mp3.c

 16  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o lib/codec/codec.o ../ch07-separate-binaries/lib/codec/codec.c

 17  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o lib/db/playlist.o lib/db/playlist.c

 18  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o lib/db/scanner.o lib/db/scanner.c
    ../ch07-separate-binaries/lib/db/scanner.l: In function yylex:
    ../ch07-separate-binaries/lib/db/scanner.l:9: warning: return makes integer from
pointer without a cast

 19  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o lib/ui/ui.o ../ch07-separate-binaries/lib/ui/ui.c

 20  ar rv lib/codec/libcodec.a lib/codec/codec.o
```

*Example 10-1. Output of make when --jobs = 2 (continued)*

```
    ar: creating lib/codec/libcodec.a
    a - lib/codec/codec.o

21  ar rv lib/db/libdb.a lib/db/playlist.o lib/db/scanner.o
    ar: creating lib/db/libdb.a
    a - lib/db/playlist.o
    a - lib/db/scanner.o

22  ar rv lib/ui/libui.a lib/ui/ui.o
    ar: creating lib/ui/libui.a
    a - lib/ui/ui.o

23  gcc   app/player/play_mp3.o lib/codec/libcodec.a lib/db/libdb.a lib/ui/libui.a   -o
    app/player/play_mp3
```

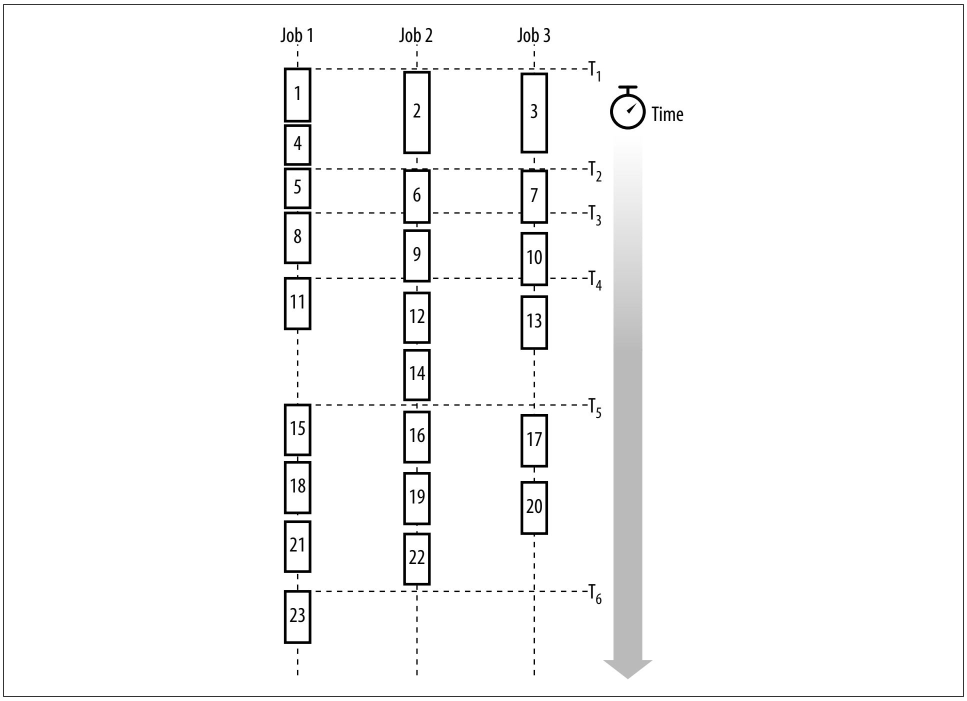

*Figure 10-1. Diagram of make when --jobs = 2*

First, make must build the generated source and dependency files. The two generated source files are the output of yacc and lex. This accounts for commands 1 and 2. The third command generates the dependency file for *play_mp3.c* and is clearly begun before the dependency files for either *playlist.c* or *scanner.c* are completed (by commands 4, 5, 8, 9, 12, and 14). Therefore, this make is running three jobs in parallel, even though the command-line option requests two jobs.

The mv commands, 4 and 5, complete the *playlist.c* source code generation started with command 1. Command 6 begins another dependency file. Each command script is always executed by a single make, but each target and prerequisite forms a separate job. Therefore, command 7, which is the second command of the dependency generation script, is being executed by the same make process as command 3. While command 6 is probably being executed by a make spawned immediately following the completion of the make that executed commands 1-4-5 (processing the yacc grammar), but before the generation of the dependency file in command 8.

The dependency generation continues in this fashion until command 14. All dependency files must be complete before make can move on to the next phase of processing, re-reading the *makefile*. This forms a natural synchronization point that make automatically obeys.

Once the *makefile* is reread with the dependency information, make can continue the build process in parallel again. This time make chooses to compile all the object files before building each of the archive libraries. This order is nondeterministic. That is, if the *makefile* is run again, it may be that the *libcodec.a* library might be built before the *playlist.c* is compiled, since that library doesn't require any objects other than *codec.o*. Thus, the example represents one possible execution order amongst many.

Finally, the program is linked. For this *makefile*, the link phase is also a natural synchronization point and will always occur last. If, however, the goal was not a single program but many programs or libraries, the last command executed might also vary.

Running multiple jobs on a multiprocessor obviously makes sense, but running more than one job on a uniprocessor can also be very useful. This is because of the latency of disk I/O and the large amount of cache on most systems. For instance, if a process, such as gcc, is idle waiting for disk I/O it may be that data for another task such as mv, yacc, or ar is currently in memory. In this case, it would be good to allow the task with available data to proceed. In general, running make with two jobs on a uniprocessor is almost always faster than running one job, and it is not uncommon for three or even four tasks to be faster than two.

The --jobs option can be used without a number. If so, make will spawn as many jobs as there are targets to be updated. This is usually a bad idea, because a large number of jobs will usually swamp a processor and can run much slower than even a single job.

Another way to manage multiple jobs is to use the system load average as a guide. The load average is the number of runnable processes averaged over some period of time, typically 1 minute, 5 minutes, and 15 minutes. The load average is expressed as a floating point number. The --load-average (or -l) option gives make a threshold above which new jobs cannot be spawned. For example, the command:

```
$ make --load-average=3.5
```

tells make to spawn new jobs only when the load average is less than or equal to 3.5. If the load average is greater, make waits until the average drops below this number, or until all the other jobs finish.

When writing a *makefile* for parallel execution, attention to proper prerequisites is even more important. As mentioned previously, when --jobs is 1, a list of prerequisites will usually be evaluated from left to right. When --jobs is greater than 1, these prerequisites may be evaluated in parallel. Therefore, any dependency relationship that was implicitly handled by the default left to right evaluation order must be made explicit when run in parallel.

Another hazard of parallel make is the problem of shared intermediate files. For example, if a directory contains both *foo.y* and *bar.y*, running yacc twice in parallel could result in one of them getting the other's instance of *y.tab.c* or *y.tab.h* or both moved into its own *.c* or *.h* file. You face a similar hazard with any procedure that stores temporary information in a scratch file that has a fixed name.

Another common idiom that hinders parallel execution is invoking a recursive make from a shell for loop:

```
dir:
        for d in $(SUBDIRS);       \
        do                         \
          $(MAKE) --directory=$$d; \
        done
```

As mentioned in the section "Recursive make" in Chapter 6, make cannot execute these recursive invocations in parallel. To achieve parallel execution, declare the directories .PHONY and make them targets:

```
.PHONY: $(SUBDIRS)
$(SUBDIRS):
        $(MAKE) --directory=$@
```

## Distributed make

GNU make supports a little known (and only slightly tested) build option for managing builds that uses multiple systems over a network. The feature relies upon the Customs library distributed with Pmake. Pmake is an alternate version of make written in about 1989 by Adam de Boor (and maintained ever since by Andreas Stolcke) for the Sprite operating system. The Customs library helps to distribute a make execution across many machines in parallel. As of version 3.77, GNU make has included support for the Customs library for distributing make.

To enable Customs library support, you must rebuild make from sources. The instructions for this process are in the *README.customs* file in the make distribution. First, you must download and build the pmake distribution (the URL is in the README), then build make with the --with-customs option.

The heart of the Customs library is the customs daemon that runs on each host participating in the distributed make network. These hosts must all share a common view of the filesystem, such as NFS provides. One instance of the customs daemon is designated the master. The master monitors hosts in the participating hosts list and allocates jobs to each member. When make is run with the --jobs flag greater than 1, make contacts the master and together they spawn jobs on available hosts in the network.

The Customs library supports a wide range of features. Hosts can be grouped by architecture and rated for performance. Arbitrary attributes can be assigned to hosts and jobs can be allocated to hosts based on combinations of attributes and boolean operators. Additionally, host status such as idle time, free disk space, free swap space, and current load average can also be accounted for when processing jobs.

If your project is implemented in C, C++, or Objective-C you should also consider distcc (*http://distcc.samba.org*) for distributing compiles across several hosts. distcc was written by Martin Pool and others to speedup Samba builds. It is a robust and complete solution for projects written in C, C++, or Objective-C. The tool is used by simply replacing the C compiler with the distcc program:

```
$ make --jobs=8 CC=distcc
```

For each compilation, distcc uses the local compiler to preprocess the output, then ships the expanded source to an available remote machine for compilation. Finally, the remote host returns the resulting object file to the master. This approach removes the necessity for having a shared filesystem, greatly simplifying installation and configuration.

The set of worker or *volunteer* hosts can be specified in several ways. The simplest is to list the volunteer hosts in an environment variable before starting distcc:

```
$ export DISTCC_HOSTS='localhost wasatch oops'
```

distcc is very configurable with options for handling host lists, integrating with the native compiler, managing compression, search paths, and handling failure and recovery.

ccache is another tool for improving compilation performance, written by Samba project leader Andrew Tridgell. The idea is simple, cache the results of previous compiles. Before performing a compile, check if the cache already contains the resulting object files. This does not require multiple hosts, or even a network. The author reports a 5 to 10 times speed up in common compilations. The easiest way to use ccache is to prefix your compiler command with ccache:

```
$ make CC='ccache gcc'
```

ccache can be used together with distcc for even greater performance improvements. In addition, both tools are available in the Cygwin tool set.

CHAPTER 11

# Example Makefiles

The *makefile*s shown throughout this book are industrial strength and quite suitable for adapting to your most advanced needs. But it's still worthwhile looking at some *makefile*s from real-life projects to see what people have done with make under the stress of providing deliverables. Here, we discuss several example *makefile*s in detail. The first example is the *makefile* to build this book. The second is the *makefile* used to build the 2.6.7 Linux kernel.

## The Book Makefile

Writing a book on programming is in itself an interesting exercise in building systems. The text of the book consists of many files, each of which needs various preprocessing steps. The examples are real programs that should be run and their output collected, post-processed, and included in the main text (so that they don't have to be cut and pasted, with the risk of introducing errors). During composition, it is useful to be able to view the text in different formats. Finally, delivering the material requires packaging. Of course, all of this must be repeatable and relatively easy to maintain.

Sounds like a job for make! This is one of the great beauties of make. It can be applied to an amazing variety of problems. This book was written in DocBook format (i.e., XML). Applying make to $\TeX$, $\LaTeX$, or troff is standard procedure when using those tools.

Example 11-1 shows the entire *makefile* for the book. It is about 440 lines long. The *makefile* is divided into these basic tasks:

- Managing the examples
- Preprocessing the XML
- Generating various output formats
- Validating the source
- Basic maintenance

*Example 11-1. The makefile to build the book*

```
# Build the book!
#
# The primary targets in this file are:
#
# show_pdf      Generate the pdf and start a viewer
# pdf           Generate the pdf
# print         Print the pdf
# show_html     Generate the html and start a viewer
# html          Generate the html
# xml           Generate the xml
# release       Make a release tarball
# clean         Clean up generated files
#

BOOK_DIR     := /test/book
SOURCE_DIR   := text
OUTPUT_DIR   := out
EXAMPLES_DIR := examples

QUIET            = @

SHELL            =  bash
AWK              := awk
CP               := cp
EGREP            := egrep
HTML_VIEWER      := cygstart
KILL             := /bin/kill
M4               := m4
MV               := mv
PDF_VIEWER       := cygstart
RM               := rm -f
MKDIR            := mkdir -p
LNDIR            := lndir
SED              := sed
SORT             := sort
TOUCH            := touch
XMLTO            := xmlto
XMLTO_FLAGS      =  -o $(OUTPUT_DIR) $(XML_VERBOSE)
process-pgm      := bin/process-includes
make-depend      := bin/make-depend

m4-macros        := text/macros.m4

# $(call process-includes, input-file, output-file)
#   Remove tabs, expand macros, and process include directives.
define process-includes
  expand $1 |                                                 \
  $(M4) --prefix-builtins --include=text $(m4-macros) - |       \
  $(process-pgm) > $2
endef

# $(call file-exists, file-name)
#   Return non-null if a file exists.
```

*Example 11-1. The makefile to build the book (continued)*

```
file-exists = $(wildcard $1)

# $(call maybe-mkdir, directory-name-opt)
#   Create a directory if it doesn't exist.
#   If directory-name-opt is omitted use $@ for the directory-name.
maybe-mkdir = $(if $(call file-exists,          \
                     $(if $1,$1,$(dir $@))),,   \
                $(MKDIR) $(if $1,$1,$(dir $@)))

# $(kill-acroread)
#   Terminate the acrobat reader.
define kill-acroread
  $(QUIET) ps -W |                                   \
  $(AWK) 'BEGIN { FIELDWIDTHS = "9 47 100" }         \
          /AcroRd32/ {                               \
                       print "Killing " $$3;         \
                       system( "$(KILL) -f " $$1 )   \
                     }'
endef

# $(call source-to-output, file-name)
#   Transform a source tree reference to an output tree reference.
define source-to-output
$(subst $(SOURCE_DIR),$(OUTPUT_DIR),$1)
endef

# $(call run-script-example, script-name, output-file)
#   Run an example makefile.
define run-script-example
  ( cd $(dir $1);                                    \
    $(notdir $1) 2>&1 |                              \
    if $(EGREP) --silent '\$$\(MAKE\)' [mM]akefile;  \
    then                                             \
      $(SED) -e 's/^++*/$$/';                        \
    else                                             \
      $(SED) -e 's/^++*/$$/'                         \
             -e '/ing directory /d'                  \
             -e 's/\[[0-9]\]//';                     \
    fi )                                             \
  > $(TMP)/out.$$$$ &                               \
  $(MV) $(TMP)/out.$$$$ $2
endef

# $(call generic-program-example,example-directory)
#   Create the rules to build a generic example.
define generic-program-example
  $(eval $1_dir      := $(OUTPUT_DIR)/$1)
  $(eval $1_make_out := $($1_dir)/make.out)
  $(eval $1_run_out  := $($1_dir)/run.out)
  $(eval $1_clean    := $($1_dir)/clean)
  $(eval $1_run_make := $($1_dir)/run-make)
  $(eval $1_run_run  := $($1_dir)/run-run)
  $(eval $1_sources  := $(filter-out %/CVS, $(wildcard $(EXAMPLES_DIR)/$1/*)))
```

*Example 11-1. The makefile to build the book (continued)*

```
  $($1_run_out): $($1_make_out) $($1_run_run)
        $$(call run-script-example, $($1_run_run), $$@)

  $($1_make_out): $($1_clean) $($1_run_make)
        $$(call run-script-example, $($1_run_make), $$@)

  $($1_clean): $($1_sources) Makefile
        $(RM) -r $($1_dir)
        $(MKDIR) $($1_dir)
        $(LNDIR) -silent ../../$(EXAMPLES_DIR)/$1 $($1_dir)
        $(TOUCH) $$@

  $($1_run_make):
        printf "#! /bin/bash -x\nmake\n" > $$@
endef


# Book output formats.
BOOK_XML_OUT     := $(OUTPUT_DIR)/book.xml
BOOK_HTML_OUT    := $(subst xml,html,$(BOOK_XML_OUT))
BOOK_FO_OUT      := $(subst xml,fo,$(BOOK_XML_OUT))
BOOK_PDF_OUT     := $(subst xml,pdf,$(BOOK_XML_OUT))
ALL_XML_SRC      := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT      := $(call source-to-output,$(ALL_XML_SRC))
DEPENDENCY_FILES := $(call source-to-output,$(subst .xml,.d,$(ALL_XML_SRC)))


# xml/html/pdf - Produce the desired output format for the book.
.PHONY: xml html pdf
xml:  $(OUTPUT_DIR)/validate
html: $(BOOK_HTML_OUT)
pdf:  $(BOOK_PDF_OUT)

# show_pdf - Generate a pdf file and display it.
.PHONY: show_pdf show_html print
show_pdf: $(BOOK_PDF_OUT)
        $(kill-acroread)
        $(PDF_VIEWER) $(BOOK_PDF_OUT)

# show_html - Generate an html file and display it.
show_html: $(BOOK_HTML_OUT)
        $(HTML_VIEWER) $(BOOK_HTML_OUT)

# print - Print specified pages from the book.
print: $(BOOK_FO_OUT)
        $(kill-acroread)
        java -Dstart=15 -Dend=15 $(FOP) $< -print > /dev/null

# $(BOOK_PDF_OUT) - Generate the pdf file.
$(BOOK_PDF_OUT): $(BOOK_FO_OUT) Makefile

# $(BOOK_HTML_OUT) - Generate the html file.
$(BOOK_HTML_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile
```

*Example 11-1. The makefile to build the book (continued)*

```
# $(BOOK_FO_OUT) - Generate the fo intermediate output file.
.INTERMEDIATE: $(BOOK_FO_OUT)
$(BOOK_FO_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# $(BOOK_XML_OUT) - Process all the xml input files.
$(BOOK_XML_OUT): Makefile

#################################################################
# FOP Support
#
FOP := org.apache.fop.apps.Fop

# DEBUG_FOP - Define this to see fop processor output.
ifndef DEBUG_FOP
  FOP_FLAGS  := -q
  FOP_OUTPUT := | $(SED) -e '/not implemented/d'        \
                         -e '/relative-align/d'         \
                         -e '/xsl-footnote-separator/d'
endif

# CLASSPATH - Compute the appropriate CLASSPATH for fop.
export CLASSPATH
CLASSPATH = $(patsubst %;,%,                                    \
              $(subst ; ,;,                                     \
                $(addprefix c:/usr/xslt-process-2.2/java/,      \
                  $(addsuffix .jar;,                            \
                    xalan                                       \
                    xercesImpl                                  \
                    batik                                       \
                    fop                                         \
                    jimi-1.0                                    \
                    avalon-framework-cvs-20020315))))

# %.pdf - Pattern rule to produce pdf output from fo input.
%.pdf: %.fo
        $(kill-acroread)
        java -Xmx128M $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)

# %.fo - Pattern rule to produce fo output from xml input.
PAPER_SIZE := letter
%.fo: %.xml
        XSLT_FLAGS="--stringparam paper.type $(PAPER_SIZE)" \
        $(XMLTO) $(XMLTO_FLAGS) fo $<

# %.html - Pattern rule to produce html output from xml input.
%.html: %.xml
        $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<

# fop_help - Display fop processor help text.
.PHONY: fop_help
fop_help:
        -java org.apache.fop.apps.Fop -help
        -java org.apache.fop.apps.Fop -print help
```

*Example 11-1. The makefile to build the book (continued)*

```
################################################################
# release - Produce a release of the book.
#
RELEASE_TAR   := mpwm-$(shell date +%F).tar.gz
RELEASE_FILES := README Makefile *.pdf bin examples out text

.PHONY: release
release: $(BOOK_PDF_OUT)
        ln -sf $(BOOK_PDF_OUT) .
        tar --create                            \
            --gzip                              \
            --file=$(RELEASE_TAR)               \
            --exclude=CVS                       \
            --exclude=semantic.cache            \
            --exclude=*~                        \
            $(RELEASE_FILES)
        ls -l $(RELEASE_TAR)

################################################################
# Rules for Chapter 1 examples.
#

# Here are all the example directories.
EXAMPLES :=                                     \
                ch01-bogus-tab                  \
                ch01-cw1                        \
                ch01-hello                      \
                ch01-cw2                        \
                ch01-cw2a                       \
                ch02-cw3                        \
                ch02-cw4                        \
                ch02-cw4a                       \
                ch02-cw5                        \
                ch02-cw5a                       \
                ch02-cw5b                       \
                ch02-cw6                        \
                ch02-make-clean                 \
                ch03-assert-not-null            \
                ch03-debug-trace                \
                ch03-debug-trace-1              \
                ch03-debug-trace-2              \
                ch03-filter-failure             \
                ch03-find-program-1             \
                ch03-find-program-2             \
                ch03-findstring-1               \
                ch03-grep                       \
                ch03-include                    \
                ch03-invalid-variable           \
                ch03-kill-acroread              \
                ch03-kill-program               \
                ch03-letters                    \
                ch03-program-variables-1        \
```

*Example 11-1. The makefile to build the book (continued)*

```
                ch03-program-variables-2        \
                ch03-program-variables-3        \
                ch03-program-variables-5        \
                ch03-scoping-issue              \
                ch03-shell                      \
                ch03-trailing-space             \
                ch04-extent                     \
                ch04-for-loop-1                 \
                ch04-for-loop-2                 \
                ch04-for-loop-3                 \
                ch06-simple                     \
                appb-defstruct                  \
                appb-arithmetic

# I would really like to use this foreach loop, but a bug in 3.80
# generates a fatal error.
#$(foreach e,$(EXAMPLES),$(eval $(call generic-program-example,$e)))

# Instead I expand the foreach by hand here.
$(eval $(call generic-program-example,ch01-bogus-tab))
$(eval $(call generic-program-example,ch01-cw1))
$(eval $(call generic-program-example,ch01-hello))
$(eval $(call generic-program-example,ch01-cw2))
$(eval $(call generic-program-example,ch01-cw2a))
$(eval $(call generic-program-example,ch02-cw3))
$(eval $(call generic-program-example,ch02-cw4))
$(eval $(call generic-program-example,ch02-cw4a))
$(eval $(call generic-program-example,ch02-cw5))
$(eval $(call generic-program-example,ch02-cw5a))
$(eval $(call generic-program-example,ch02-cw5b))
$(eval $(call generic-program-example,ch02-cw6))
$(eval $(call generic-program-example,ch02-make-clean))
$(eval $(call generic-program-example,ch03-assert-not-null))
$(eval $(call generic-program-example,ch03-debug-trace))
$(eval $(call generic-program-example,ch03-debug-trace-1))
$(eval $(call generic-program-example,ch03-debug-trace-2))
$(eval $(call generic-program-example,ch03-filter-failure))
$(eval $(call generic-program-example,ch03-find-program-1))
$(eval $(call generic-program-example,ch03-find-program-2))
$(eval $(call generic-program-example,ch03-findstring-1))
$(eval $(call generic-program-example,ch03-grep))
$(eval $(call generic-program-example,ch03-include))
$(eval $(call generic-program-example,ch03-invalid-variable))
$(eval $(call generic-program-example,ch03-kill-acroread))
$(eval $(call generic-program-example,ch03-kill-program))
$(eval $(call generic-program-example,ch03-letters))
$(eval $(call generic-program-example,ch03-program-variables-1))
$(eval $(call generic-program-example,ch03-program-variables-2))
$(eval $(call generic-program-example,ch03-program-variables-3))
$(eval $(call generic-program-example,ch03-program-variables-5))
$(eval $(call generic-program-example,ch03-scoping-issue))
$(eval $(call generic-program-example,ch03-shell))
```

*Example 11-1. The makefile to build the book (continued)*

```
$(eval $(call generic-program-example,ch03-trailing-space))
$(eval $(call generic-program-example,ch04-extent))
$(eval $(call generic-program-example,ch04-for-loop-1))
$(eval $(call generic-program-example,ch04-for-loop-2))
$(eval $(call generic-program-example,ch04-for-loop-3))
$(eval $(call generic-program-example,ch06-simple))
$(eval $(call generic-program-example,ch10-echo-bash))
$(eval $(call generic-program-example,appb-defstruct))
$(eval $(call generic-program-example,appb-arithmetic))


################################################################
# validate
#
# Check for 1) unexpanded m4 macros; b) tabs; c) FIXME comments; d)
# RM: responses to Andy; e) duplicate m4 macros
#
validation_checks := $(OUTPUT_DIR)/chk_macros_tabs              \
                     $(OUTPUT_DIR)/chk_fixme                    \
                     $(OUTPUT_DIR)/chk_duplicate_macros         \
                     $(OUTPUT_DIR)/chk_orphaned_examples

.PHONY: validate-only
validate-only: $(OUTPUT_DIR)/validate
$(OUTPUT_DIR)/validate: $(validation_checks)
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_macros_tabs: $(ALL_XML_OUT)
        # Looking for macros and tabs...
        $(QUIET)! $(EGREP) --ignore-case                        \
                           --line-number                        \
                           --regexp='\b(m4_|mp_)'               \
                           --regexp='\011'                      \
                           $^
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_fixme: $(ALL_XML_OUT)
        # Looking for RM: and FIXME...
        $(QUIET)$(AWK)                                          \
                '/FIXME/ { printf "%s:%s: %s\n", FILENAME, NR, $$0 }    \
                 /^ *RM:/  {                                    \
                           if ( $$0 !~ /RM: Done/ )             \
                           printf "%s:%s: %s\n", FILENAME, NR, $$0      \
                         }' $(subst $(OUTPUT_DIR)/,$(SOURCE_DIR)/,$^)
        $(TOUCH) $@

$(OUTPUT_DIR)/chk_duplicate_macros: $(SOURCE_DIR)/macros.m4
        # Looking for duplicate macros...
        $(QUIET)! $(EGREP) --only-matching              \
              "\`[^']+'," $< |                          \
        $(SORT) |                                       \
        uniq -c |                                       \
```

*Example 11-1. The makefile to build the book (continued)*

```
        $(AWK) '$$1 > 1 { printf "$<:0: %s\n", $$0 }' | \
        $(EGREP) "^"
        $(TOUCH) $@

ALL_EXAMPLES := $(TMP)/all_examples

$(OUTPUT_DIR)/chk_orphaned_examples: $(ALL_EXAMPLES) $(DEPENDENCY_FILES)
        $(QUIET)$(AWK) -F/ '/(EXAMPLES|OUTPUT)_DIR/ { print $$3 }'      \
                $(filter %.d,$^) |                                      \
        $(SORT) -u |                                                    \
        comm -13 - $(filter-out %.d,$^)
        $(TOUCH) $@

.INTERMEDIATE: $(ALL_EXAMPLES)
$(ALL_EXAMPLES):
        # Looking for unused examples...
        $(QUIET) ls -p $(EXAMPLES_DIR) |        \
        $(AWK) '/CVS/ { next }                  \
                /\//  { print substr($$0, 1, length - 1) }' > $@

#################################################################
# clean
#
clean:
        $(kill-acroread)
        $(RM) -r $(OUTPUT_DIR)
        $(RM) $(SOURCE_DIR)/*~ $(SOURCE_DIR)/*.log semantic.cache
        $(RM) book.pdf

#################################################################
# Dependency Management
#
# Don't read or remake includes if we are doing a clean.
#
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(DEPENDENCY_FILES)
endif

vpath %.xml $(SOURCE_DIR)
vpath %.tif $(SOURCE_DIR)
vpath %.eps $(SOURCE_DIR)

$(OUTPUT_DIR)/%.xml: %.xml $(process-pgm) $(m4-macros)
        $(call process-includes, $<, $@)

$(OUTPUT_DIR)/%.tif: %.tif
        $(CP) $< $@

$(OUTPUT_DIR)/%.eps: %.eps
        $(CP) $< $@

$(OUTPUT_DIR)/%.d: %.xml $(make-depend)
        $(make-depend) $< > $@
```

*Example 11-1. The makefile to build the book (continued)*

```
#################################################################
# Create Output Directory
#
# Create the output directory if necessary.
#
DOCBOOK_IMAGES := $(OUTPUT_DIR)/release/images
DRAFT_PNG      := /usr/share/docbook-xsl/images/draft.png

ifneq "$(MAKECMDGOALS)" "clean"
  _CREATE_OUTPUT_DIR :=                                         \
    $(shell                                                     \
      $(MKDIR) $(DOCBOOK_IMAGES) &                             \
      $(CP) $(DRAFT_PNG) $(DOCBOOK_IMAGES);                     \
      if ! [[ $(foreach d,                                      \
                $(notdir                                        \
                  $(wildcard $(EXAMPLES_DIR)/ch*)),             \
                -e $(OUTPUT_DIR)/$d &) -e . ]];                \
      then                                                      \
        echo Linking examples... > /dev/stderr;                 \
        $(LNDIR) $(BOOK_DIR)/$(EXAMPLES_DIR) $(BOOK_DIR)/$(OUTPUT_DIR); \
      fi)
endif
```

The *makefile* is written to run under Cygwin with no serious attempt at portability to Unix. Nevertheless, I believe there are few, if any, incompatibilities with Unix that cannot be resolved by redefining a variable or possibly introducing an additional variable.

The global variables section first defines the location of the root directory and the relative locations of the text, examples, and output directories. Each nontrivial program used by the *makefile* is defined as a variable.

## Managing Examples

The first task, managing the examples, is the most complex. Each example is stored in its own directory under *book/examples/chn-<title>*. Examples consist of a *makefile* along with any supporting files and directories. To process an example we first create a directory of symbolic links to the output tree and work there so that no artifacts of running the *makefile* are left in the source tree. Furthermore, most of the examples require setting the current working directory to that of the *makefile*, in order to generate the expected output. After symlinking the source, we execute a shell script, `run-make`, to invoke the *makefile* with the appropriate arguments. If no shell script is present in the source tree, we can generate a default version. The output of the `run-make` script is saved in *make.out*. Some examples produce an executable, which must also be run. This is accomplished by running the script `run-run` and saving its output in the file *run.out*.

Creating the tree of symbolic links is performed by this code at the end of the *makefile*:

```
ifneq "$(MAKECMDGOALS)" "clean"
  _CREATE_OUTPUT_DIR :=                                          \
    $(shell                                                      \
      …
      if ! [[ $(foreach d,                                       \
                $(notdir                                         \
                  $(wildcard $(EXAMPLES_DIR)/ch*)),              \
                -e $(OUTPUT_DIR)/$d &&) -e . ]];                 \
      then                                                       \
        echo Linking examples... > /dev/stderr;                  \
        $(LNDIR) $(BOOK_DIR)/$(EXAMPLES_DIR) $(BOOK_DIR)/$(OUTPUT_DIR); \
      fi)
endif
```

The code consists of a single, simple variable assignment wrapped in an `ifneq` conditional. The conditional is there to prevent `make` from creating the output directory structure during a `make clean`. The actual variable is a dummy whose value is never used. However, the `shell` function on the right-hand side is executed immediately when `make` reads the *makefile*. The `shell` function checks if each example directory exists in the output tree. If any is missing, the `lndir` command is invoked to update the tree of symbolic links.

The test used by the `if` is worth examining more closely. The test itself consists of one `-e` test (i.e., does the file exist?) for each example directory. The actual code goes something like this: use `wildcard` to determine all the examples and strip their directory part with `notdir`, then for each example directory produce the text `-e $(OUTPUT_DIR)/`*`dir`* `&&`. Now, concatenate all these pieces, and embed them in a `bash [[...]]` test. Finally, negate the result. One extra test, `-e .`, is included to allow the `foreach` loop to simply add `&&` to every clause.

This is sufficient to ensure that new directories are always added to the build when they are discovered.

The next step is to create rules that will update the two output files, *make.out* and *run.out*. This is done for each example *.out* file with a user-defined function:

```
# $(call generic-program-example,example-directory)
#   Create the rules to build a generic example.
define generic-program-example
  $(eval $1_dir      := $(OUTPUT_DIR)/$1)
  $(eval $1_make_out := $($1_dir)/make.out)
  $(eval $1_run_out  := $($1_dir)/run.out)
  $(eval $1_clean    := $($1_dir)/clean)
  $(eval $1_run_make := $($1_dir)/run-make)
  $(eval $1_run_run  := $($1_dir)/run-run)
  $(eval $1_sources  := $(filter-out %/CVS, $(wildcard $(EXAMPLES_DIR)/$1/*)))

  $($1_run_out): $($1_make_out) $($1_run_run)
        $$(call run-script-example, $($1_run_run), $$@)
  $($1_make_out): $($1_clean) $($1_run_make)
        $$(call run-script-example, $($1_run_make), $$@)

  $($1_clean): $($1_sources) Makefile
        $(RM) -r $($1_dir)
        $(MKDIR) $($1_dir)
        $(LNDIR) -silent ../../$(EXAMPLES_DIR)/$1 $($1_dir)
        $(TOUCH) $$@

  $($1_run_make):
        printf "#! /bin/bash -x\nmake\n" > $$@
endef
```

This function is intended to be invoked once for each example directory:

```
$(eval $(call generic-program-example,ch01-bogus-tab))
$(eval $(call generic-program-example,ch01-cw1))
$(eval $(call generic-program-example,ch01-hello))
$(eval $(call generic-program-example,ch01-cw2))
```

The variable definitions at the beginning of the function are mostly for convenience and to improve readability. Further improvement comes from performing the assignments inside `eval` so their value can be used immediately by the macro without extra quoting.

The heart of the function is the first two targets: `$($1_run_out)` and `$($1_make_out)`. These update the *run.out* and *make.out* targets for each example, respectively. The variable names are composed from the example directory name and the indicated suffix, *_run_out* or *_make_out*.

The first rule says that *run.out* depends upon *make.out* and the *run-run* script. That is, rerun the example program if `make` has been run or the *run-run* control script has been updated. The target is updated with the `run-script-example` function:

```
# $(call run-script-example, script-name, output-file)
#   Run an example makefile.
define run-script-example
  ( cd $(dir $1);                                   \
    $(notdir $1) 2>&1 |                             \
    if $(EGREP) --silent '\$$\(MAKE\)' [mM]akefile;     \
    then                                            \
      $(SED) -e 's/^++*/$$/';                       \
    else                                            \
      $(SED) -e 's/^++*/$$/'                        \
             -e '/ing directory /d'                 \
             -e 's/\[[0-9]\]//';                    \
    fi )                                            \
  > $(TMP)/out.$$$$ &&                              \
  $(MV) $(TMP)/out.$$$$ $2
endef
```

This function requires the path to the script and the output filename. It changes to the script's directory and runs the script, piping both the standard output and error output through a filter to clean them up.[*]

The *make.out* target is similar but has an added complication. If new files are added to an example, we would like to detect the situation and rebuild the example. The `_CREATE_OUTPUT_DIR` code rebuilds symlinks only if a new directory is discovered, not when new files are added. To detect this situation, we drop a timestamp file in each example directory indicating when the last `lndir` was performed. The `$($1_clean)` target updates this timestamp file and depends upon the actual source files in the examples directory (not the symlinks in the output directory). If `make`'s dependency analysis discovers a newer file in the examples directory than the *clean* timestamp file, the command script will delete the symlinked output directory, recreate it, and drop a new *clean* timestamp file. This action is also performed when the *makefile* itself is modified.

Finally, the *run-make* shell script invoked to run the *makefile* is typically a two-line script.

```
#! /bin/bash -x
make
```

It quickly became tedious to produce these boilerplate scripts, so the `$($1_run_make)` target was added as a prerequisite to `$($1_make_out)` to create it. If the prerequisite is missing, the *makefile* generates it in the output tree.

The `generic-program-example` function, when executed for each example directory, creates all the rules for running examples and preparing the output for inclusion in the XML files. These rules are triggered by computed dependencies included in the *makefile*. For example, the dependency file for Chapter 1 is:

```
out/ch01.xml: $(EXAMPLES_DIR)/ch01-hello/Makefile
out/ch01.xml: $(OUTPUT_DIR)/ch01-hello/make.out
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/count_words.c
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/lexer.l
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw1/Makefile
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw1/make.out
out/ch01.xml: $(EXAMPLES_DIR)/ch01-cw2/lexer.l
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw2/make.out
out/ch01.xml: $(OUTPUT_DIR)/ch01-cw2/run.out
out/ch01.xml: $(OUTPUT_DIR)/ch01-bogus-tab/make.out
```

[*] The cleaning process gets complex. The *run-run* and *run-make* scripts often use `bash -x` to allow the actual `make` command line to be echoed. The `-x` option puts `++` before each command in the output, which the cleaning script transforms into a simple `$` representing the shell prompt. However, commands are not the only information to appear in the output. Because `make` is running the example and eventually starts another `make`, simple *makefile*s include extra, unwanted output such as the messages `Entering directory...` and `Leaving directory...` as well as displaying a `make` level number in messages. For simple *makefile*s that do not recursively invoke `make`, we strip this inappropriate output to present the output of `make` as if it were run from a top-level shell.

These dependencies are generated by a simple awk script, imaginatively named make-depend:

```
#! /bin/awk -f

function generate_dependency( prereq )
{
  filename = FILENAME
  sub( /text/, "out", filename )
  print filename ": " prereq
}

/^ *include-program/ {
  generate_dependency( "$(EXAMPLES_DIR)/" $2 )
}

/^ *mp_program\(/ {
  match( $0, /\((.*)\)/, names )
  generate_dependency( "$(EXAMPLES_DIR)/" names[1] )
}

/^ *include-output/ {
  generate_dependency( "$(OUTPUT_DIR)/" $2 )
}

/^ *mp_output\(/ {
  match( $0, /\((.*)\)/, names )
  generate_dependency( "$(OUTPUT_DIR)/" names[1] )
}

/graphic fileref/ {
  match( $0, /"(.*)"/, out_file )
  generate_dependency( out_file[1] );
}
```

The script searches for patterns like:

```
mp_program(ch01-hello/Makefile)
mp_output(ch01-hello/make.out)
```

(The mp_program macro uses the program listing format, while the mp_output macro uses the program output format.) The script generates the dependency from the source filename and the filename parameter.

Finally, the generation of dependency files is triggered by a make include statement, in the usual fashion:

```
# $(call source-to-output, file-name)
#   Transform a source tree reference to an output tree reference.
define source-to-output
$(subst $(SOURCE_DIR),$(OUTPUT_DIR),$1)
endef
...
ALL_XML_SRC      := $(wildcard $(SOURCE_DIR)/*.xml)
DEPENDENCY_FILES := $(call source-to-output,$(subst .xml,.d,$(ALL_XML_SRC)))
```

```
...
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(DEPENDENCY_FILES)
endif

vpath %.xml $(SOURCE_DIR)
...
$(OUTPUT_DIR)/%.d: %.xml $(make-depend)
        $(make-depend) $< > $@
```

This completes the code for handling examples. Most of the complexity stems from the desire to include the actual source of the *makefile*s as well as the actual output from make and the example programs. I suspect there is also a little bit of the "put up or shut up" syndrome here. If I believe make is so great, it should be able to handle this complex task and, by golly, it can.

## XML Preprocessing

At the risk of branding myself as a philistine for all posterity, I must admit I don't like XML very much. I find it awkward and verbose. So, when I discovered that the manuscript must be written in DocBook, I looked for more traditional tools that would help ease the pain. The m4 macro processor and awk were two tools that helped immensely.

There were two problems with DocBook and XML that m4 was perfect for: avoiding the verbose syntax of XML and managing the XML identifiers used in cross-referencing. For instance, to emphasize a word in DocBook, you must write:

```
<emphasis>not</emphasis>
```

Using m4, I wrote a simple macro that allowed me to instead write:

```
mp_em(not)
```

Ahh, that feels better. In addition, I introduced many symbolic formatting styles appropriate for the material, such as mp_variable and mp_target. This allowed me to select a trivial format, such as literal, and change it later to whatever the production department preferred without having to perform a global search and replace.

I'm sure the XML aficionados will probably send me boat loads of email telling me how to do this with entities or some such, but remember Unix is about getting the job done now with the tools at hand, and as Larry Wall loves to say, "there's more than one way to do it." Besides, I'm afraid learning too much XML will rot my brain.

The second task for m4 was handling the XML identifiers used for cross-referencing. Each chapter, section, example, and table is labeled with an identifier:

```
<sect1 id="MPWM-CH-7-SECT-1">
```

References to a chapter must use this identifier. This is clearly an issue from a programming standpoint. The identifiers are complex constants sprinkled throughout the "code." Furthermore, the symbols themselves have no meaning. I have no idea what section 1 of Chapter 7 might have been about. By using m4, I could avoid duplicating complex literals, and provide a more meaningful name:

```
<sect1 id="mp_se_makedepend">
```

Most importantly, if chapters or sections shift, as they did many times, the text could be updated by changing a few constants in a single file. The advantage was most noticeable when sections were renumbered in a chapter. Such an operation might require a half dozen global search and replace operations across all files if I hadn't used symbolic references.

Here is an example of several m4 macros[*]:

```
m4_define(`mp_tag',    `<$1>`$2'</$1>')
m4_define(`mp_lit',    `mp_tag(literal, `$1')')

m4_define(`mp_cmd',    `mp_tag(command,`$1')')
m4_define(`mp_target', `mp_lit($1)')

m4_define(`mp_all',    `mp_target(all)')
m4_define(`mp_bash',   `mp_cmd(bash)')

m4_define(`mp_ch_examples',     `MPWM-CH-11')
m4_define(`mp_se_book',         `MPWM-CH-11.1')
m4_define(`mp_ex_book_makefile',`MPWM-CH-11-EX-1')
```

The other preprocessing task was to implement an include feature for slurping in the example text previously discussed. This text needed to have its tabs converted to spaces (since O'Reilly's DocBook converter cannot handle tabs and *makefile*s have lots of tabs!), must be wrapped in a [CDATA[...]] to protect special characters, and finally, has to trim the extra newlines at the beginning and end of examples. I accomplished this with another little awk program called process-includes:

```
#! /usr/bin/awk -f
function expand_cdata( dir )
{
  start_place = match( $1, "include-" )
  if ( start_place > 0 )
  {
    prefix = substr( $1, 1, start_place - 1 )
  }
  else
  {
    print "Bogus include '" $0 "'" > "/dev/stderr"
  }

  end_place = match( $2, "(</(programlisting|screen)>.*)$", tag )
```

[*] The mp prefix stands for Managing Projects (the book's title), macro processor, or make pretty. Take your pick.

```
    if ( end_place > 0 )
    {
      file = dir substr( $2, 1, end_place - 1 )
    }
    else
    {
      print "Bogus include '" $0 "'" > "/dev/stderr"
    }

    command = "expand " file

    printf "%s>&33;&91;CDATA[", prefix
    tail = 0
    previous_line = ""
    while ( (command | getline line) > 0 )
    {
      if ( tail )
        print previous_line;

      tail = 1
      previous_line = line
    }

    printf "%s&93;&93;&62;%s\n", previous_line, tag[1]
    close( command )
  }

  /include-program/ {
    expand_cdata( "examples/" )
    next;
  }

  /include-output/ {
    expand_cdata( "out/" )
    next;
  }

  /<(programlisting|screen)> *$/ {
    # Find the current indentation.
    offset = match( $0, "<(programlisting|screen)>" )

    # Strip newline from tag.
    printf $0

    # Read the program...
    tail = 0
    previous_line = ""
    while ( (getline line) > 0 )
    {
      if ( line ~ "</(programlisting|screen)>" )
      {
        gsub( /^ */, "", line )
        break
      }
      if ( tail )
        print previous_line

      tail = 1
      previous_line = substr( line, offset + 1 )
    }

    printf "%s%s\n", previous_line, line

    next
  }

  {
    print
  }
```

In the *makefile*, we copy the XML files from the source tree to the output tree, transforming tabs, macros, and include files in the process:

```
process-pgm := bin/process-includes
m4-macros   := text/macros.m4

# $(call process-includes, input-file, output-file)
#   Remove tabs, expand macros, and process include directives.
define process-includes
  expand $1 |                                        \
  $(M4) --prefix-builtins --include=text $(m4-macros) - | \
  $(process-pgm) > $2
endef

vpath %.xml $(SOURCE_DIR)

$(OUTPUT_DIR)/%.xml: %.xml $(process-pgm) $(m4-macros)
        $(call process-includes, $<, $@)
```

The pattern rule indicates how to get an XML file from the source tree into the output tree. It also says that all the output XML files should be regenerated if the macros or the include processor change.

## Generating Output

So far, nothing we've covered has actually formatted any text or created anything that can be printed or displayed. Obviously, a very important feature if the *makefile* is to format a book. There were two formats that I was interested in: HTML and PDF.

I figured out how to format to HTML first. There's a great little program, `xsltproc`, and its helper script, `xmlto`, that I used to do the job. Using these tools, the process was fairly simple:

```
# Book output formats.
BOOK_XML_OUT     := $(OUTPUT_DIR)/book.xml
BOOK_HTML_OUT    := $(subst xml,html,$(BOOK_XML_OUT))
```

```
ALL_XML_SRC      := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT      := $(call source-to-output,$(ALL_XML_SRC))

# html - Produce the desired output format for the book.
.PHONY: html
html: $(BOOK_HTML_OUT)

# show_html - Generate an html file and display it.
.PHONY: show_html
show_html: $(BOOK_HTML_OUT)
        $(HTML_VIEWER) $(BOOK_HTML_OUT)

# $(BOOK_HTML_OUT) - Generate the html file.
$(BOOK_HTML_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# %.html - Pattern rule to produce html output from xml input.
%.html: %.xml
        $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<
```

The pattern rule does most of the work of converting an XML file into an HTML file. The book is organized as a single top-level file, *book.xml*, that includes each chapter. The top-level file is represented by `BOOK_XML_OUT`. The HTML counterpart is `BOOK_HTML_OUT`, which is a target. The `BOOK_HTML_OUT` file has its included XML files a prerequisites. For convenience, there are two phony targets, `html` and `show_html`, that create the HTML file and display it in the local browser, respectively.

Although easy in principle, generating PDF was considerably more complex. The `xsltproc` program is able to produce PDF directly, but I was unable to get it to work. All this work was done on Windows with Cygwin and the Cygwin version of `xsltproc` wanted POSIX paths. The custom version of DocBook I was using and the manuscript itself contained Windows-specific paths. This difference, I believe, gave `xsltproc` fits that I could not quell. Instead, I chose to use `xsltproc` to generate XML formatting objects and the Java program FOP (*http://xml.apache.org/fop*) for generating the PDF.

Thus, the code to generate PDF is somewhat longer:

```
# Book output formats.
BOOK_XML_OUT     := $(OUTPUT_DIR)/book.xml
BOOK_FO_OUT      := $(subst xml,fo,$(BOOK_XML_OUT))
BOOK_PDF_OUT     := $(subst xml,pdf,$(BOOK_XML_OUT))
ALL_XML_SRC      := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT      := $(call source-to-output,$(ALL_XML_SRC))

# pdf - Produce the desired output format for the book.
.PHONY: pdf
pdf:  $(BOOK_PDF_OUT)

# show_pdf - Generate a pdf file and display it.
.PHONY: show_pdf
show_pdf: $(BOOK_PDF_OUT)
        $(kill-acroread)
        $(PDF_VIEWER) $(BOOK_PDF_OUT)
```

```
# $(BOOK_PDF_OUT) - Generate the pdf file.
$(BOOK_PDF_OUT): $(BOOK_FO_OUT) Makefile

# $(BOOK_FO_OUT) - Generate the fo intermediate output file.
.INTERMEDIATE: $(BOOK_FO_OUT)
$(BOOK_FO_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# FOP Support
FOP := org.apache.fop.apps.Fop

# DEBUG_FOP - Define this to see fop processor output.
ifndef DEBUG_FOP
  FOP_FLAGS  := -q
  FOP_OUTPUT := | $(SED) -e '/not implemented/d'        \
                       -e '/relative-align/d'         \
                       -e '/xsl-footnote-separator/d'
endif

# CLASSPATH - Compute the appropriate CLASSPATH for fop.
export CLASSPATH
CLASSPATH = $(patsubst %;,%,                                \
              $(subst ; ,;,                                 \
                $(addprefix c:/usr/xslt-process-2.2/java/,  \
                  $(addsuffix .jar;,                        \
                    xalan                                   \
                    xercesImpl                              \
                    batik                                   \
                    fop                                     \
                    jimi-1.0                                \
                    avalon-framework-cvs-20020315))))

# %.pdf - Pattern rule to produce pdf output from fo input.
%.pdf: %.fo
        $(kill-acroread)
        java -Xmx128M $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)

# %.fo - Pattern rule to produce fo output from xml input.
PAPER_SIZE := letter
%.fo: %.xml
        XSLT_FLAGS="--stringparam paper.type $(PAPER_SIZE)" \
        $(XMLTO) $(XMLTO_FLAGS) fo $<

# fop_help - Display fop processor help text.
.PHONY: fop_help
fop_help:
        -java org.apache.fop.apps.Fop -help
        -java org.apache.fop.apps.Fop -print help
```

As you can see, there are now two pattern rules reflecting the two-stage process I used. The *.xml* to *.fo* rule invokes `xmlto`. The *.fo* to *.pdf* rule first kills any running Acrobat reader (because the program locks the PDF file, preventing FOP from writing the file), then runs FOP. FOP is a very chatty program, and scrolling through hundreds of lines of pointless warnings got old fast, so I added a simple sed filter, FOP_OUTPUT, to remove the irritating warnings. Occasionally, however, those warnings had some real data in them, so I added a debugging feature, DEBUG_FOP, to disable my filter. Finally, like the HTML version, I added two convenience targets, pdf and show_pdf, to kick the whole thing off.

## Validating the Source

What with DocBook's allergy to tabs, macro processors, include files and comments from editors, making sure the source text is correct and complete is not easy. To help, I implemented four validation targets that check for various forms of correctness.

```
validation_checks := $(OUTPUT_DIR)/chk_macros_tabs         \
                     $(OUTPUT_DIR)/chk_fixme               \
                     $(OUTPUT_DIR)/chk_duplicate_macros    \
                     $(OUTPUT_DIR)/chk_orphaned_examples

.PHONY: validate-only
validate-only: $(OUTPUT_DIR)/validate
$(OUTPUT_DIR)/validate: $(validation_checks)
        $(TOUCH) $@
```

Each target generates a timestamp file, and they are all prerequisites of a top-level timestamp file, *validate*.

```
$(OUTPUT_DIR)/chk_macros_tabs: $(ALL_XML_OUT)
        # Looking for macros and tabs...
        $(QUIET)! $(EGREP) --ignore-case           \
                           --line-number           \
                           --regexp='\b(m4_|mp_)'   \
                           --regexp='\011'         \
                           $^
        $(TOUCH) $@
```

This first check looks for m4 macros that were not expanded during preprocessing. This indicates either a misspelled macro or a macro that has never been defined. The check also scans for tab characters. Of course, neither of these situations should ever happen, but they did! One interesting bit in the command script is the exclamation point after $(QUIET). The purpose is to negate the exit status of egrep. That is, make should consider the command a failure if egrep *does* find one of the patterns.

```
$(OUTPUT_DIR)/chk_fixme: $(ALL_XML_OUT)
        # Looking for RM: and FIXME...
        $(QUIET)$(AWK)                                              \
                '/FIXME/ { printf "%s:%s: %s\n", FILENAME, NR, $$0 }    \
                 /^ *RM:/  {                                        \
                           if ( $$0 !~ /RM: Done/ )                 \
                           printf "%s:%s: %s\n", FILENAME, NR, $$0      \
                         }' $(subst $(OUTPUT_DIR)/,$(SOURCE_DIR)/,$^)
        $(TOUCH) $@
```

This check is for unresolved notes to myself. Obviously, any text labeled FIXME should be fixed and the label removed. In addition, any occurrence of RM: that is not followed immediately by Done should be flagged. Notice how the format of the printf function follows the standard format for compiler errors. This way, standard tools that recognize compiler errors will properly process these warnings.

```
$(OUTPUT_DIR)/chk_duplicate_macros: $(SOURCE_DIR)/macros.m4
        # Looking for duplicate macros...
        $(QUIET)! $(EGREP) --only-matching              \
              "\[^]+'," $< |                            \
        $(SORT) |                                       \
        uniq -c |                                       \
        $(AWK) '$$1 > 1 { printf "$>:0: %s\n", $$0 }' | \
        $(EGREP) "^"
        $(TOUCH) $@
```

This checks for duplicate macro definitions in the m4 macro file. The m4 processor does not consider redefinition to be an error, so I added a special check. The pipeline goes like this: grab the defined symbol in each macro, sort, count duplicates, filter out all lines with a count of one, then use egrep one last time purely for its exit status. Again, note the negation of the exit status to produce a make error only when something is found.

```
ALL_EXAMPLES := $(TMP)/all_examples

$(OUTPUT_DIR)/chk_orphaned_examples: $(ALL_EXAMPLES) $(DEPENDENCY_FILES)
        $(QUIET)$(AWK) -F/ '/(EXAMPLES|OUTPUT)_DIR/ { print $$3 }' \
                $(filter %.d,$^) |                                 \
        $(SORT) -u |                                               \
        comm -13 - $(filter-out %.d,$^)
        $(TOUCH) $@

.INTERMEDIATE: $(ALL_EXAMPLES)
$(ALL_EXAMPLES):
        # Looking for unused examples...
        $(QUIET) ls -p $(EXAMPLES_DIR) |        \
        $(AWK) '/CVS/ { next }                  \
                /\// { print substr($$0, 1, length - 1) }' > $@
```

The final check looks for examples that are not referenced in the text. This target uses a funny trick. It requires two sets of input files: all the example directories, and all the XML dependency files. The prerequisites list is separated into these two sets using filter and filter-out. The list of example directories is generated by using ls -p (this appends a slash to each directory) and scanning for slashes. The pipeline first grabs the XML dependency files from the prerequisite list, outputs the example directories it finds in them, and removes any duplicates. These are the examples actually referenced in the text. This list is fed to comm's standard input, while the list of all known example directories is fed as the second file. The -13 option indicates that comm should print only lines found in column two (that is, directories that are not referenced from a dependency file).

## The Linux Kernel Makefile

The Linux kernel *makefile* is an excellent example of using make in a complex build environment. While it is beyond the scope of this book to explain how the Linux kernel is structured and built, we can examine several interesting uses of make employed by the kernel build system. See *http://macarchive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Germaschewski-OLS2003.pdf* for a more complete discussion of the 2.5/2.6 kernel build process and its evolution from the 2.4 approach.

Since the *makefile* has so many facets, we will discuss just a few features that are applicable to a variety of applications. First, we'll look at how single-letter make variables are used to simulate single-letter command-line options. We'll see how the source and binary trees are separated in a way that allows users to invoke make from the source tree. Next, we'll examine the way the *makefile* controls the verboseness of the output. Then we'll review the most interesting user-defined functions and see how they reduce code duplication, improve readability, and provide encapsulation. Finally, we'll look at the way the *makefile* implements a simple help facility.

The Linux kernel build follows the familiar configure, build, install pattern used by my most free software. While many free and open software packages use a separate *configure* script (typically built by autoconf), the Linux kernel *makefile* implements configuration with make, invoking scripts and helper programs indirectly.

When the configuration phase is complete, a simple make or make all will build the bare kernel, all the modules, and produce a compressed kernel image (these are the vmlinux, modules, and bzImage targets, respectively). Each kernel build is given a unique version number in the file *version.o* linked into the kernel. This number (and the *version.o* file) are updated by the *makefile* itself.

Some *makefile* features you might want to adapt to your own *makefile* are: the handling of command line options, analyzing command-line goals, saving build status between builds, and managing the output of make.

### Command-Line Options

The first part of the *makefile* contains code for setting common build options from the command line. Here is an excerpt that controls the verbose flag:

```
# To put more focus on warnings, be less verbose as default
# Use 'make V=1' to see the full commands
ifdef V
  ifeq ("$(origin V)", "command line")
    KBUILD_VERBOSE = $(V)
  endif
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif
```

The nested ifdef/ifeq pair ensures that the KBUILD_VERBOSE variable is set only if V is set on the command line. Setting V in the environment or *makefile* has no effect. The following ifndef conditional will then turn off the verbose option if KBUILD_VERBOSE has not yet been set. To set the verbose option from either the environment or *makefile*, you must set KBUILD_VERBOSE and not V.

Notice, however, that setting KBUILD_VERBOSE directly on the command line is allowed and works as expected. This can be useful when writing shell scripts (or aliases) to invoke the *makefile*. These scripts would then be more self-documenting, similar to using GNU long options.

The other command-line options, sparse checking (C) and external modules (M), both use the same careful checking to avoid accidentally setting them from within the *makefile*.

The next section of the *makefile* handles the output directory option (O). This is a fairly involved piece of code. To highlight its structure, we've replaced some parts of this excerpt with ellipses:

```
# kbuild supports saving output files in a separate directory.
# To locate output files in a separate directory two syntax'es are supported.
# In both cases the working directory must be the root of the kernel src.
# 1) O=
# Use "make O=dir/to/store/output/files/"
#
# 2) Set KBUILD_OUTPUT
# Set the environment variable KBUILD_OUTPUT to point to the directory
# where the output files shall be placed.
# export KBUILD_OUTPUT=dir/to/store/output/files/
# make
#
# The O= assigment takes precedence over the KBUILD_OUTPUT environment variable.
# KBUILD_SRC is set on invocation of make in OBJ directory
# KBUILD_SRC is not intended to be used by the regular user (for now)
ifeq ($(KBUILD_SRC),)

  # OK, Make called in directory where kernel src resides
  # Do we want to locate output files in a separate directory?
  ifdef O
    ifeq ("$(origin O)", "command line")
      KBUILD_OUTPUT := $(O)
    endif
  endif
  …
  ifneq ($(KBUILD_OUTPUT),)
    …
    .PHONY: $(MAKECMDGOALS)

    $(filter-out _all,$(MAKECMDGOALS)) _all:
            $(if $(KBUILD_VERBOSE:1=),@)$(MAKE) -C $(KBUILD_OUTPUT)        \
            KBUILD_SRC=$(CURDIR)         KBUILD_VERBOSE=$(KBUILD_VERBOSE)  \
            KBUILD_CHECK=$(KBUILD_CHECK) KBUILD_EXTMOD="$(KBUILD_EXTMOD)"  \
            -f $(CURDIR)/Makefile $@
    # Leave processing to above invocation of make
    skip-makefile := 1
  endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)

# We process the rest of the Makefile if this is the final invocation of make
ifeq ($(skip-makefile),)
  …the rest of the makefile here…
endif   # skip-makefile
```

Essentially, this says that if `KBUILD_OUTPUT` is set, invoke `make` recursively in the output directory defined by `KBUILD_OUTPUT`. Set `KBUILD_SRC` to the directory where `make` was originally executed, and grab the *makefile* from there as well. The rest of the *makefile* will not be seen by `make`, since `skip-makefile` will be set. The recursive `make` will reread this same *makefile* again, only this time `KBUILD_SRC` will be set, so `skip-makefile` will be undefined, and the rest of the *makefile* will be read and processed.

This concludes the processing of command-line options. The bulk of the *makefile* follows in the `ifeq ($(skip-makefile),)` section.

## Configuration Versus Building

The *makefile* contains configuration targets and build targets. The configuration targets have the form `menuconfig`, `defconfig`, etc. Maintenance targets like `clean` are treated as configuration targets as well. Other targets such as `all`, `vmlinux`, and `modules` are build targets. The primary result of invoking a configuration target is two files: *.config* and *.config.cmd*. These two files are included by the *makefile* for build targets but are not included for configuration targets (since the configuration target creates them). It is also possible to mix both configuration targets and build targets on a single `make` invocation, such as:

```
$ make oldconfig all
```

In this case, the *makefile* invokes itself recursively handling each target individually, thus handling configuration targets separately from build targets.

The code controlling configuration, build, and mixed targets begins with:

```
# To make sure we do not include .config for any of the *config targets
# catch them early, and hand them over to scripts/kconfig/Makefile
# It is allowed to specify more targets when calling make, including
# mixing *config targets and build targets.
# For example 'make oldconfig all'.
# Detect when mixed targets is specified, and make a second invocation
# of make so .config is not included in this case either (for *config).
no-dot-config-targets := clean mrproper distclean \
                         cscope TAGS tags help %docs check%

config-targets := 0
mixed-targets  := 0
dot-config     := 1
```

The variable `no-dot-config-targets` lists additional targets that do not require a *.config* file. The code then initializes the `config-targets`, `mixed-targets`, and `dot-config` variables. The `config-targets` variable is 1 if there are any configuration targets on the command line. The `dot-config` variable is 1 if there are build targets on the command line. Finally, `mixed-targets` is 1 if there are both configuration and build targets.

The code to set `dot-config` is:

```
ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
  ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
    dot-config := 0
  endif
endif
```

The `filter` expression is non-empty if there are configuration targets in `MAKECMDGOALS`. The `ifneq` part is true if the `filter` expression is not empty. The code is hard to follow partly because it contains a double negative. The `ifeq` expression is true if `MAKECMDGOALS` contains only configuration targets. So, `dot-config` will be set to 0 if there are configuration targets and only configuration targets in `MAKECMDGOALS`. A more verbose implementation might make the meaning of these two conditionals more clear:

```
config-target-list := clean mrproper distclean \
                         cscope TAGS tags help %docs check%

config-target-goal := $(filter $(config-target-list), $(MAKECMDGOALS))
build-target-goal := $(filter-out $(config-target-list), $(MAKECMDGOALS))

ifdef config-target-goal
  ifndef build-target-goal
    dot-config := 0
  endif
endif
```

The `ifdef` form can be used instead of `ifneq`, because empty variables are treated as undefined, but care must be taken to ensure a variable does not contain merely a string of blanks (which would cause it to be defined).

The `config-targets` and `mixed-targets` variables are set in the next code block:

```
ifeq ($(KBUILD_EXTMOD),)
  ifneq ($(filter config %config,$(MAKECMDGOALS)),)
    config-targets := 1
    ifneq ($(filter-out config %config,$(MAKECMDGOALS)),)
      mixed-targets := 1
    endif
  endif
endif
```

`KBUILD_EXTMOD` will be non-empty when external modules are being built, but not during normal builds. The first `ifneq` will be true when `MAKECMDGOALS` contains a goal with the config suffix. The second ifneq will be true when MAKECMDGOALS contains nonconfig targets, too.

Once the variables are set, they are used in an if-else chain with four branches. The code has been condensed and indented to highlight its structure:

```
ifeq ($(mixed-targets),1)
  # We're called with mixed targets (*config and build targets).
  # Handle them one by one.
  %:: FORCE
        $(Q)$(MAKE) -C $(srctree) KBUILD_SRC= $@
else
  ifeq ($(config-targets),1)
    # *config targets only - make sure prerequisites are updated, and descend
    # in scripts/kconfig to make the *config target
    %config: scripts_basic FORCE
          $(Q)$(MAKE) $(build)=scripts/kconfig $@
  else
    # Build targets only - this includes vmlinux, arch specific targets, clean
    # targets and others. In general all targets except *config targets.
    …
    ifeq ($(dot-config),1)
      # In this section, we need .config
      # Read in dependencies to all Kconfig* files, make sure to run
      # oldconfig if changes are detected.
      -include .config.cmd
      include .config

      # If .config needs to be updated, it will be done via the dependency
      # that autoconf has on .config.
      # To avoid any implicit rule to kick in, define an empty command
      .config: ;

      # If .config is newer than include/linux/autoconf.h, someone tinkered
      # with it and forgot to run make oldconfig
      include/linux/autoconf.h: .config
            $(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
    else
      # Dummy target needed, because used as prerequisite
      include/linux/autoconf.h: ;
    endif

    include $(srctree)/arch/$(ARCH)/Makefile
    … lots more make code …
  endif #ifeq ($(config-targets),1)
endif #ifeq ($(mixed-targets),1)
```

The first branch, ifeq ($(mixed-targets),1), handles mixed command-line arguments. The only target in this branch is a completely generic pattern rule. Since there are no specific rules to handle targets (those rules are in another conditional branch), each target invokes the pattern rule once. This is how a command line with both configuration targets and build targets is separated into a simpler command line. The command script for the generic pattern rule invokes make recursively for each target, causing this same logic to be applied, only this time with no mixed command-line targets. The FORCE prerequisite is used instead of .PHONY, because pattern rules like:

```
%:: FORCE
```

cannot be declared .PHONY. So it seems reasonable to use FORCE consistently everywhere.

The second branch of the if-else chain, ifeq ($(config-targets),1), is invoked when there are only configuration targets on the command line. Here the primary target in the branch is the pattern rule %config (other targets have been omitted). The command script invokes make recursively in the *scripts/kconfig* subdirectory and passes along the target. The curious $(build) construct is defined at the end of the *makefile*:

```
# Shorthand for $(Q)$(MAKE) -f scripts/Makefile.build obj=dir
# Usage:
# $(Q)$(MAKE) $(build)=dir
build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/Makefile.build obj
```

If KBUILD_SRC is set, the -f option is given a full path to the *scripts makefile*, otherwise a simple relative path is used. Next, the obj variable is set to the righthand side of the equals sign.

The third branch, ifeq ($(dot-config),1), handles build targets that require including the two generated configuration files, *.config* and *.config.cmd*. The final branch merely includes a dummy target for *autoconf.h* to allow it to be used as a prerequisite, even if it doesn't exist.

Most of the remainder of the *makefile* follows the third and fourth branches. It contains the code for building the kernel and modules.

## Managing Command Echo

The kernel *makefile*s use a novel technique for managing the level of detail echoed by commands. Each significant task is represented in both a verbose and a quiet version. The verbose version is simply the command to be executed in its natural form and is stored in a variable named cmd_*action*. The brief version is a short message describing the action and is stored in a variable named quiet_cmd_*action*. For example, the command to produce emacs tags is:

```
quiet_cmd_TAGS = MAKE $@
      cmd_TAGS = $(all-sources) | etags -
```

A command is executed by calling the cmd function:

```
# If quiet is set, only print short version of command
cmd = @$(if $($(quiet)cmd_$(1)),\
        echo '  $($(quiet)cmd_$(1))' &&) $(cmd_$(1))
```

To invoke the code for building emacs tags, the *makefile* would contain:

```
TAGS:
        $(call cmd,TAGS)
```

Notice the cmd function begins with an @, so the only text echoed by the function is text from the echo command. In normal mode, the variable quiet is empty, and the test in the if, $($(quiet)cmd_$(1)), expands to $(cmd_TAGS). Since this variable is not empty, the entire function expands to:

```
echo '  $(all-sources) | etags -' && $(all-sources) | etags -
```

If the quiet version is desired, the variable quiet contains the value quiet_ and the function expands to:

```
echo '  MAKE $@' && $(all-sources) | etags -
```

The variable can also be set to silent_. Since there is no command silent_cmd_TAGS, this value causes the cmd function to echo nothing at all.

Echoing the command sometimes becomes more complex, particularly if commands contain single quotes. In these cases, the *makefile* contains this code:

```
$(if $($(quiet)cmd_$(1)),echo '  $(subst ','\'',$($(quiet)cmd_$(1)))';)
```

Here the echo command contains a substitution that replaces single quotes with escaped single quotes to allow them to be properly echoed.

Minor commands that do not warrant the trouble of writing cmd_ and quiet_cmd_ variables are prefixed with $(Q), which contains either nothing or @:

```
ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif

# If the user is running make -s (silent mode), suppress echoing of
# commands

ifneq ($(findstring s,$(MAKEFLAGS)),)
  quiet=silent_
endif
```

## User-Defined Functions

The kernel *makefile* defines a number of functions. Here we cover the most interesting ones. The code has been reformatted to improve readability.

The check_gcc function is used to select a gcc command-line option.

```
# $(call check_gcc,preferred-option,alternate-option)
check_gcc =                                         \
    $(shell if $(CC) $(CFLAGS) $(1) -S -o /dev/null \
                -xc /dev/null > /dev/null 2>&1;      \
            then                                     \
              echo "$(1)";                           \
            else                                     \
              echo "$(2)";                           \
            fi ;)
```

The function works by invoking gcc on a null input file with the preferred command-line option. The output file, standard output, and standard error files are discarded. If the gcc command succeeds, it means the preferred command-line option is valid for this architecture and is returned by the function. Otherwise, the option is invalid and the alternate option is returned. An example use can be found in *arch/i386/Makefile*:

```
# prevent gcc from keeping the stack 16 byte aligned
CFLAGS += $(call check_gcc,-mpreferred-stack-boundary=2,)
```

The `if_changed_dep` function generates dependency information using a remarkable technique.

```
# execute the command and also postprocess generated
# .d dependencies file
if_changed_dep =                                        \
    $(if                                                \
      $(strip $?                                        \
        $(filter-out FORCE $(wildcard $^),$^)           \
        $(filter-out $(cmd_$(1)),$(cmd_$@))             \
        $(filter-out $(cmd_$@),$(cmd_$(1)))),           \
      @set -e;                                          \
      $(if $($(quiet)cmd_$(1)),                         \
        echo '  $(subst ','\'',$($(quiet)cmd_$(1)))';)  \
      $(cmd_$(1));                                      \
      scripts/basic/fixdep                              \
         $(depfile)                                     \
         $@                                             \
         '$(subst $$,$$$$,$(subst ','\'',$(cmd_$(1))))' \
         > $(@D)/.$(@F).tmp;                            \
      rm -f $(depfile);                                 \
      mv -f $(@D)/.$(@F).tmp $(@D)/.$(@F).cmd)
```

The function consists of a single `if` clause. The details of the test are pretty obscure, but it is clear the intent is to be non-empty if the dependency file should be regenerated. Normal dependency information is concerned with the modification timestamps on files. The kernel build system adds another wrinkle to this task. The kernel build uses a wide variety of compiler options to control the construction and behavior of components. To ensure that command-line options are properly accounted for during a build, the *makefile* is implemented so that if command-line options used for a particular target change, the file is recompiled. Let's see how this is accomplished.

In essence, the command used to compile each file in the kernel is saved in a *.cmd* file. When a subsequent build is executed, make reads the *.cmd* files and compares the current compile command with the last command. If they are different, the *.cmd* dependency file is regenerated causing the object file to be rebuilt. The *.cmd* file usually contains two items: the dependencies that represent actual files for the target file and a single variable recording the command-line options. For example, the file *arch/i386/kernel/cpu/mtrr/if.c* yields this (abbreviated) file:

```
cmd_arch/i386/kernel/cpu/mtrr/if.o := gcc -Wp,-MD …; if.c

deps_arch/i386/kernel/cpu/mtrr/if.o := \
  arch/i386/kernel/cpu/mtrr/if.c \
  …

arch/i386/kernel/cpu/mtrr/if.o: $(deps_arch/i386/kernel/cpu/mtrr/if.o)
$(deps_arch/i386/kernel/cpu/mtrr/if.o):
```

Getting back to the `if_changed_dep` function, the first argument to the `strip` is simply the prerequisites that are newer than the target, if any. The second argument to `strip` is all the prerequisites other than files and the empty target `FORCE`. The really obscure bit is the last two `filter-out` calls:

```
$(filter-out $(cmd_$(1)),$(cmd_$@))
$(filter-out $(cmd_$@),$(cmd_$(1)))
```

One or both of these calls will expand to a non-empty string if the command-line options have changed. The macro `$(cmd_$(1))` is the current command and `$(cmd_$@)` will be the previous command, for instance the variable `cmd_arch/i386/kernel/cpu/mtrr/if.o` just shown. If the new command contains additional options, the first `filter-out` will be empty, and the second will expand to the new options. If the new command contains fewer options, the first command will contain the deleted options and the second will be empty. Interestingly, since `filter-out` accepts a list of words (each treated as an independent pattern), the order of options can change and the `filter-out` will still accurately identify added or removed options. Pretty nifty.

The first statement in the command script sets a shell option to exit immediately on error. This prevents the multiline script from corrupting files in the event of problems. For simple scripts another way to achieve this effect is to connect statements with `&&` rather than semicolons.

The next statement is an `echo` command written using the techniques described in the section “Managing Command Echo” earlier in this chapter, followed by the dependency generating command itself. The command writes `$(depfile)`, which is then transformed by `scripts/basic/fixdep`. The nested `subst` function in the `fixdep` command line first escapes single quotes, then escapes occurrences of `$$` (the current process number in shell syntax).

Finally, if no errors have occurred, the intermediate file `$(depfile)` is removed and the generated dependency file (with its *.cmd* suffix) is moved into place.

The next function, `if_changed_rule`, uses the same comparison technique as `if_changed_dep` to control the execution of a command:

```
# Usage: $(call if_changed_rule,foo)
# will check if $(cmd_foo) changed, or any of the prequisites changed,
# and if so will execute $(rule_foo)

if_changed_rule =                                  \
    $(if $(strip $?                                \
           $(filter-out $(cmd_$(1)),$(cmd_$(@F)))   \
           $(filter-out $(cmd_$(@F)),$(cmd_$(1)))), \
      @$(rule_$(1)))
```

In the topmost *makefile*, this function is used to link the kernel with these macros:

```
# This is a bit tricky: If we need to relink vmlinux, we want
# the version number incremented, which means recompile init/version.o
# and relink init/init.o. However, we cannot do this during the
# normal descending-into-subdirs phase, since at that time
# we cannot yet know if we will need to relink vmlinux.
# So we descend into init/ inside the rule for vmlinux again.
…

quiet_cmd_vmlinux__ = LD $@
define cmd_vmlinux__
  $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) \
    …
endef

# set -e makes the rule exit immediately on error

define rule_vmlinux__
  +set -e;                                            \
  $(if $(filter .tmp_kallsyms%,$^),,                  \
    echo '  GEN     .version';                        \
    . $(srctree)/scripts/mkversion > .tmp_version;    \
    mv -f .tmp_version .version;                      \
    $(MAKE) $(build)=init;)                           \
  $(if $($(quiet)cmd_vmlinux__),                      \
    echo '  $($(quiet)cmd_vmlinux__)' &&)             \
  $(cmd_vmlinux__);                                   \
  echo 'cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).cmd
endef

define rule_vmlinux
  $(rule_vmlinux__);         \
  $(NM) $@ |                 \
  grep -v '\(compiled\)\|…' | \
  sort > System.map
endef
```

The `if_changed_rule` function is used to invoke `rule_vmlinux`, which performs the link and builds the final *System.map*. As the comment in the *makefile* notes, the `rule_vmlinux__` function must regenerate the kernel version file and relink *init.o* before relinking *vmlinux*. This is controlled by the first `if` in `rule_vmlinux__`. The second `if` controls the echoing of the link command, `$(cmd_vmlinux__)`. After the link command, the actual command executed is recorded in a *.cmd* file for comparison in the next build.

CHAPTER 12
# Debugging Makefiles

Debugging *makefile*s is somewhat of a black art. Unfortunately, there is no such thing as a *makefile* debugger to examine how a particular rule is being evaluated or a variable expanded. Instead, most debugging is performed with simple print statements and by inspection of the *makefile*. GNU make provides some help with various built-in functions and command-line options.

One of the best ways to debug a *makefile* is to add debugging hooks and use defensive programming techniques that you can fall back on when things go awry. I'll present a few basic debugging techniques and defensive coding practices I've found most helpful.

## Debugging Features of make

The warning function is very useful for debugging wayward *makefile*s. Because the warning function expands to the empty string, it can be placed anywhere in a *makefile*: at the top-level, in target or prerequisite lists, and in command scripts. This allows you to print the value of variables wherever it is most convenient to inspect them. For example:

```
$(warning A top-level warning)

FOO := $(warning Right-hand side of a simple variable)bar
BAZ = $(warning Right-hand side of a recursive variable)boo

$(warning A target)target: $(warning In a prerequisite list)makefile $(BAZ)
        $(warning In a command script)
        ls
$(BAZ):
```

yields the output:

```
$ make
makefile:1: A top-level warning
makefile:2: Right-hand side of a simple variable
makefile:5: A target
```

```
makefile:5: In a prerequisite list
makefile:5: Right-hand side of a recursive variable
makefile:8: Right-hand side of a recursive variable
makefile:6: In a command script
ls
makefile
```

Notice that the evaluation of the warning function follows the normal make algorithm for immediate and deferred evaluation. Although the assignment to BAZ contains a warning, the message does not print until BAZ is evaluated in the prerequisites list.

The ability to inject a warning call anywhere makes it an essential debugging tool.

## Command-Line Options

There are three command-line options I find most useful for debugging: --just-print (-n), --print-data-base (-p), and --warn-undefined-variables.

### --just-print

The first test I perform on a new *makefile* target is to invoke make with the --just-print (-n) option. This causes make to read the *makefile* and print every command it would normally execute to update the target but without executing them. As a convenience, GNU make will also echo commands marked with the silent modifier (@).

The option is supposed to suppress all command execution. While this may be true in one sense, practically speaking, you must take care. While make will not execute command scripts, it will evaluate shell function calls that occur within an immediate context. For instance:

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); \
             do                               \
               [[ -d $$d ]] || mkdir -p $$d;  \
             done)

$(objects) : $(sources)
```

As we've seen before, the purpose of the _MKDIRS simple variable is to trigger the creation of essential directories. When this is executed with --just-print, the shell command will be executed as usual when the *makefile* is read. Then make will echo (without executing) each compilation command required to update the $(objects) file list.

### --print-data-base

The --print-data-base (-p) option is another one you'll use often. It executes the *makefile*, displaying the GNU copyright followed by the commands as they are run by make, then it will dump its internal database. The data is collected into groups of values: variables, directories, implicit rules, pattern-specific variables, files (explicit rules), and the vpath search path:

```
# GNU Make 3.80
# Copyright (C) 2002  Free Software Foundation, Inc.
# This is free software; see the source for copying conditions.
# There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE.
normal command execution occurs here

# Make data base, printed on Thu Apr 29 20:58:13 2004

# Variables
…
# Directories
…
# Implicit Rules
…
# Pattern-specific variable values
…
# Files
…
# VPATH Search Paths
```

Let's examine these sections in more detail.

The variables section lists each variable along with a descriptive comment:

```
# automatic
<D = $(patsubst %/,%,$(dir $<))
# environment
EMACS_DIR = C:/usr/emacs-21.3.50.7
# default
CWEAVE = cweave
# makefile (from `../mp3_player/makefile', line 35)
CPPFLAGS = $(addprefix -I ,$(include_dirs))
# makefile (from `../ch07-separate-binaries/makefile', line 44)
RM := rm -f
# makefile (from `../mp3_player/makefile', line 14)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
        $(AR) $(ARFLAGS) $$@ $$^
endef
```

Automatic variables are not printed, but convenience variables derived from them like `$(<D)` are. The comment indicates the type of the variable as returned by the `origin` function (see the section "Less Important Miscellaneous Functions" in Chapter 4). If the variable is defined in a file, the filename and line number of the definition is given. Simple and recursive variables are distinguished by the

assignment operator. The value of a simple variable will be displayed as the evaluated form of the righthand side.

The next section, labeled Directories, is more useful to make developers than to make users. It lists the directories being examined by make, including SCCS and RCS subdirectories that might exist, but usually do not. For each directory, make displays implementation details, such as the device number, inode, and statistics on file pattern matches.

The Implicit Rules section follows. This contains all the built-in and user-defined pattern rules in make's database. Again, for those rules defined in a file, a comment indicates the file and line number:

```
%.c %.h: %.y
# commands to execute (from `../mp3_player/makefile', line 73):
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%: %.c
#  commands to execute (built-in):
        $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.c
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Examining this section is a great way to become familiar with the variety and structure of make's built-in rules. Of course, not all built-in rules are implemented as pattern rules. If you don't find the rule you're looking for, check in the Files section where the old-style suffix rules are listed.

The next section catalogs the pattern-specific variables defined in the *makefile*. Recall that pattern-specific variables are variable definitions whose scope is precisely the execution time of their associated pattern rule. For example, the pattern variable YYLEXFLAG, defined as:

```
%.c %.h: YYLEXFLAG := -d
%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h
```

would be displayed as:

```
# Pattern-specific variable values

%.c :
# makefile (from `Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%
```

```
%.h :
# makefile (from `Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%

# 2 pattern-specific variable values
```

The Files section follows and lists all the explicit and suffix rules that relate to specific files:

```
# Not a target:
.p.o:
#  Implicit rule search has not been done.
#  Modification time never checked.
#  File has not been updated.
#  commands to execute (built-in):
        $(COMPILE.p) $(OUTPUT_OPTION) $<

lib/ui/libui.a: lib/ui/ui.o
#  Implicit rule search has not been done.
#  Last modified 2004-04-01 22:04:09.515625
#  File has been updated.
#  Successfully updated.
#  commands to execute (from `../mp3_player/lib/ui/module.mk', line 3):
        ar rv $@ $^

lib/codec/codec.o: ../mp3_player/lib/codec/codec.c ../mp3_player/lib/codec/codec.c ..
/mp3_player/include/codec/codec.h
#  Implicit rule search has been done.
#  Implicit/static pattern stem: `lib/codec/codec'
#  Last modified 2004-04-01 22:04:08.40625
#  File has been updated.
#  Successfully updated.
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Intermediate files and suffix rules are labeled "Not a target"; the remainder are targets. Each file includes comments indicating how make has processed the rule. Files that are found through the normal vpath search have their resolved path displayed.

The last section is labeled VPATH Search Paths and lists the value of VPATH and all the vpath patterns.

For *makefile*s that make extensive use of user-defined functions and eval to create complex variables and rules, examining this output is often the only way to verify that macro expansion has generated the expected values.

### --warn-undefined-variables

This option causes make to display a warning whenever an undefined variable is expanded. Since undefined variables expand to the empty string, it is common for typographical errors in variable names to go undetected for long periods. The problem with this option, and why I use it only rarely, is that many built-in rules include undefined variables as hooks for user-defined values. So running make with this option will inevitably produce many warnings that are not errors and have no useful relationship to the user's *makefile*. For example:

```
$ make --warn-undefined-variables -n
makefile:35: warning: undefined variable MAKECMDGOALS
makefile:45: warning: undefined variable CFLAGS
makefile:45: warning: undefined variable TARGET_ARCH
...
makefile:35: warning: undefined variable MAKECMDGOALS
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
...
make: warning: undefined variable LDFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable LOADLIBES
make: warning: undefined variable LDLIBS
```

Nevertheless, this command can be extremely valuable on occasion in catching these kinds of errors.

## The --debug Option

When you need to know how make analyzes your dependency graph, use the --debug option. This provides the most detailed information available other than by running a debugger. There are five debugging options and one modifier: basic, verbose, implicit, jobs, all, and makefile, respectively.

If the debugging option is specified as --debug, basic debugging is used. If the debugging option is given as -d, all is used. To select other combinations of options, use a comma separated list --debug=*option1*,*option2* where the option can be one of the following words (actually, make looks only at the first letter):

basic
: Basic debugging is the least detailed. When enabled, make prints each target that is found to be out-of-date and the status of the update action. Sample output looks like:

```
File all does not exist.
  File app/player/play_mp3 does not exist.
    File app/player/play_mp3.o does not exist.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

verbose
: This option sets the basic option and includes additional information about which files where parsed, prerequisites that did not need to be rebuilt, etc.:

```
File all does not exist.
 Considering target file app/player/play_mp3.
  File app/player/play_mp3 does not exist.
   Considering target file app/player/play_mp3.o.
    File app/player/play_mp3.o does not exist.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/include/player/play_mp3.h.
    Finished prerequisites of target file app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
   Pruning file app/player/play_mp3.o.
```

`implicit`

This option sets the `basic` option and includes additional information about implicit rule searches for each target:

```
File all does not exist.
  File app/player/play_mp3 does not exist.
  Looking for an implicit rule for app/player/play_mp3.
  Trying pattern rule with stem play_mp3.
  Trying implicit prerequisite app/player/play_mp3.o.
  Found an implicit rule for app/player/play_mp3.
    File app/player/play_mp3.o does not exist.
    Looking for an implicit rule for app/player/play_mp3.o.
    Trying pattern rule with stem play_mp3.
    Trying implicit prerequisite app/player/play_mp3.c.
    Found prerequisite app/player/play_mp3.c as VPATH ../mp3_player/app/player/
play_mp3.c
    Found an implicit rule for app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

`jobs`

This options prints the details of subprocesses invoked by `make`. It does not enable the `basic` option.

```
Got a SIGCHLD; 1 unreaped children.
gcc ... ../mp3_player/app/player/play_mp3.c
Putting child 0x10033800 (app/player/play_mp3.o) PID 576 on the chain.
Live child 0x10033800 (app/player/play_mp3.o) PID 576
Got a SIGCHLD; 1 unreaped children.
Reaping winning child 0x10033800 PID 576
Removing child 0x10033800 PID 576 from chain.
```

`all`

This enables all the previous options and is the default when using the `-d` option.

`makefile`

Normally, debugging information is not enabled until after the *makefile*s have been updated. This includes updating any included files, such as lists of dependencies. When you use this modifier, `make` will print the selected information while rebuilding *makefile*s and include files. This option enables the `basic` option and is also enabled by the `all` option.

## Writing Code for Debugging

As you can see, there aren't too many tools for debugging *makefile*s, just a few ways to dump `make`'s internal data structures and a couple of print statements. When it comes right down to it, it is up to you to write your *makefile*s in ways that either minimize the chance of errors or provide your own scaffolding to help debug them.

The suggestions in this section are laid out somewhat arbitrarily as coding practices, defensive coding, and debugging techniques. While specific items, such as checking the exit status of commands, could be placed in either the good coding practice section or the defensive coding section, the three categories reflect the proper bias. Focus on coding your *makefile*s well without cutting too many corners. Include plenty of defensive coding to protect the *makefile* against unexpected events and environmental conditions. Finally, when bugs do arise, use every trick you can find to squash them.

The "Keep It Simple" Principle (*http://www.catb.org/~esr/jargon/html/K/KISS-Principle.html*) is at the heart of all good design. As you've seen in previous chapters, *makefile*s can quickly become complex, even for mundane tasks, such as dependency generation. Fight the tendency to include more and more features in your build system. You'll fail, but not as badly as you would if you simply include every feature that occurs to you.

### Good Coding Practices

In my experience, most programmers do not see writing *makefile*s as programming and, therefore, do not take the same care as they do when writing in C++ or Java. But the `make` language is a complete nonprocedural language. If the reliability and maintainability of your build system is important, write it with care and use the best coding practices you can.

One of the most important aspects of programming robust *makefile*s is to check the return status of commands. Of course, `make` will check simple commands automatically, but *makefile*s often include compound commands that can fail quietly:

```
do:
        cd i-dont-exist; \
        echo *.c
```

When run, this *makefile* does not terminate with an error status, although an error most definitely occurs:

```
$ make
cd i-dont-exist; \
echo *.c
```

```
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
*.c
```

Furthermore, the globbing expression fails to find any *.c* files, so it quietly returns the globbing expression. Oops. A better way to code this command script is to use the shell's features for checking and preventing errors:

```
SHELL = /bin/bash
do:
        cd i-dont-exist && \
        shopt -s nullglob &&
        echo *.c
```

Now the cd error is properly transmitted to make, the echo command never executes, and make terminates with an error status. In addition, setting the nullglob option of bash causes the globbing pattern to return the empty string if no files are found. (Of course, your particular application may prefer the globbing pattern.)

```
$ make
cd i-dont-exist && \
echo *.c
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
make: *** [do] Error 1
```

Another good coding practice is formatting your code for maximum readability. Most *makefile*s I see are poorly formatted and, consequently, difficult to read. Which do you find easier to read?

```
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); do [[ -d $$d \
]] || mkdir -p $$d; done)
```

or:

```
_MKDIRS := $(shell                          \
             for d in $(REQUIRED_DIRS);      \
             do                              \
               [[ -d $$d ]] || mkdir -p $$d; \
             done)
```

If you're like most people, you'll find the first more difficult to parse, the semicolons harder to find, and the number of statements more difficult to count. These are not trivial concerns. A significant percentage of the syntax errors you will encounter in command scripts will be due to missing semicolons, backslashes, or other separators, such as pipe and logical operators.

Also, note that not all missing separators will generate an error. For instance, neither of the following errors will produce a shell syntax error:

```
TAGS:
        cd src \
        ctags --recurse

disk_free:
        echo "Checking free disk space..." \
        df . | awk '{ print $$4 }'
```

Formatting commands for readability will make these kinds of errors easier to catch. When formatting user-defined functions, indent the code. Occasionally, the extra spaces in the resulting macro expansion cause problems. If so, wrap the formatting in a `strip` function call. When formatting long lists of values, separate each value on its own line. Add a comment before each target, give a brief explanation, and document the parameter list.

The next good coding practice is the liberal use of variables to hold common values. As in any program, the unrestrained use of literal values creates code duplication and leads to maintenance problems and bugs. Another great advantage of variables is that you can get `make` to display them for debugging purposes during execution. I show a nice command line interface in the section "Debugging Techniques," later in this chapter.

## Defensive Coding

Defensive code is code that can execute only if one of your assumptions or expectations is wrong — an `if` test that is never true, an `assert` function that never fails, or tracing code. Of course, the value of this code that never executes is that occasionally (usually when you least expect it), it does run and produce a warning or error, or you choose to enable tracing code to allow you to view the inner workings of `make`.

You've already seen most of this code in other contexts, but for convenience it is repeated here.

Validation checking is a great example of defensive code. This code sample verifies that the currently executing version of `make` is 3.80:

```
NEED_VERSION := 3.80
$(if $(filter $(NEED_VERSION),$(MAKE_VERSION)),,              \
  $(error You must be running make version $(NEED_VERSION).))
```

For Java applications, it is useful to include a check for files in the `CLASSPATH`.

Validation code can also simply ensure that something is true. The directory creation code from the previous section is of this nature.

Another great defensive coding technique is to use the `assert` functions defined in the section "Flow Control" in Chapter 4. Here are several versions:

```
# $(call assert,condition,message)
define assert
  $(if $1,,$(error Assertion failed: $2))
endef

# $(call assert-file-exists,wildcard-pattern)
define assert-file-exists
  $(call assert,$(wildcard $1),$1 does not exist)
endef

# $(call assert-not-null,make-variable)
define assert-not-null
```

```
  $(call assert,$($1),The variable "$1" is null)
endef
```

I find sprinkling `assert` calls around the *makefile* to be a cheap and effective way of detecting missing and misspelled parameters as well as violations of other assumptions.

In Chapter 4, we wrote a pair of functions to trace the expansion of user-defined functions:

```
# $(debug-enter)
debug-enter = $(if $(debug_trace),\
                $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = $(if $(debug_trace),$(warning Leaving $0))

comma := ,
echo-args   = $(subst ' ','$(comma) ',\
                $(foreach a,1 2 3 4 5 6 7 8 9,'$($a)'))
```

You can add these macro calls to your own functions and leave them disabled until they are required for debugging. To enable them, set `debug_trace` to any nonempty value:

```
$ make debug_trace=1
```

As noted in Chapter 4, these trace macros have a number of problems of their own but can still be useful in tracking down bugs.

The final defensive programming technique is simply to make disabling the `@` command modifier easy by using it through a `make` variable, rather than literally:

```
QUIET := @
…
target:
        $(QUIET) some command
```

Using this technique, you can see the execution of the silent command by redefining `QUIET` on the command line:

```
$ make QUIET=
```

## Debugging Techniques

This section discusses general debugging techniques and issues. Ultimately, debugging is a grab-bag of whatever works for your situation. These techniques have worked for me, and I've come to rely on them to debug even the simplest *makefile* problems. Maybe they'll help you, too.

One of the very annoying bugs in 3.80 is that when `make` reports problems in *makefile*s and includes a line number, I usually find that the line number is wrong. I haven't investigated whether the problem is due to include files, multiline variable assignments, or user-defined macros, but there it is. Usually the line number make reports is larger than the actual line number. In complex *makefile*s, I've had the line number be off by as much as 20 lines.

Often the easiest way to see the value of a make variable is to print it during the execution of a target. Although adding print statements using warning is simple, the extra effort of adding a generic debug target for printing variables can save lots of time in the long run. Here is a sample debug target:

```
debug:
        $(for v,$(V), \
          $(warning $v = $($v)))
```

To use it, just set the list of variables to print on the command line, and include the debug target:

```
$ make V="USERNAME SHELL" debug
makefile:2: USERNAME = Owner
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
```

If you want to get really tricky, you can use the MAKECMDGOALS variable to avoid the assignment to the variable V:

```
debug:
        $(for v,$(V) $(MAKECMDGOALS), \
          $(if $(filter debug,$v),,$(warning $v = $($v))))
```

Now you can print variables by simply listing them on the command line. I don't recommend this technique, though, because you'll also get confusing make warnings indicating it doesn't know how to update the variables (since they are listed as targets):

```
$ make debug PATH SHELL
makefile:2: USERNAME = Owner
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
make: *** No rule to make target USERNAME.  Stop.
```

In Chapter 10, I briefly mentioned using a debugging shell to help understand some of the activities make performs behind the scenes. While make echos commands in command scripts before they are executed, it does not echo the commands executed in shell functions. Often these commands are subtle and complex, particularly since they may be executed immediately or in a deferred fashion, if they occur in a recursive variable assignment. One way to see these commands execute is to request that the subshell enable debug printing:

```
DATE := $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p $(OUTPUT_DIR))

all: ;
```

When run with sh's debugging option, we see:

```
$ make SHELL="sh -x"
+ date +%F
+ '[' -d out-2004-05-11 ']'
+ mkdir -p out-2004-05-11
```

This even provides additional debugging information beyond make warning statements, since with this option the shell also displays the value of variables and expressions.

Many of the examples in this book are written as deeply nested expressions, such as this one that checks the PATH variable on a Windows/Cygwin system:

```
$(if $(findstring /bin/,                              \
       $(firstword                                    \
         $(wildcard                                   \
           $(addsuffix /sort$(if $(COMSPEC),.exe),    \
             $(subst :, ,$(PATH)))))),,               \
  $(error Your PATH is wrong, c:/usr/cygwin/bin should \
    precede c:/WINDOWS/system32))
```

There is no good way to debug these expressions. One reasonable approach is to unroll them, and print each subexpression:

```
$(warning $(subst :, ,$(PATH)))
$(warning /sort$(if $(COMSPEC),.exe))
$(warning $(addsuffix /sort$(if $(COMSPEC),.exe),     \
             $(subst :, ,$(PATH))))
$(warning $(wildcard                                  \
            $(addsuffix /sort$(if $(COMSPEC),.exe),    \
              $(subst :, ,$(PATH)))))
```

Although a bit tedious, without a real debugger, this is the best way (and sometimes the only way) to determine the value of various subexpressions.

## Common Error Messages

The 3.81 GNU make manual includes an excellent section listing make error messages and their causes. We review a few of the most common ones here. Some of the issues described are not strictly make errors, such as syntax errors in command scripts, but are nonetheless common problems for developers. For a complete list of make errors, see the make manual.

Error messages printed by make have a standard format:

```
makefile:n: *** message. Stop.
```

or:

```
make:n: *** message. Stop.
```

The `makefile` part is the name of the *makefile* or include file in which the error occurred. The next part is the line number where the error occurred, followed by three asterisks and, finally, the error message.

Note that it is `make`'s job to run other programs and that, if errors occur, it is very likely that problems in your *makefile* will manifest themselves as errors in these other programs. For instance, shell errors may result from badly formed command scripts, or compiler errors from incorrect command-line arguments. Figuring out what program produced the error message is your first task in solving the problem. Fortunately, `make`'s error messages are fairly self-evident.

## Syntax Errors

These are usually typographical errors: missing parentheses, using spaces instead of tabs, etc.

One of the most common errors for new `make` users is omitting parentheses around variable names:

```
foo:
        for f in $SOURCES; \
        do                 \
           …               \
        done
```

This will likely result in `make` expanding `$S` to nothing, and the shell executing the loop only once with `f` having a value of `OURCES`. Depending on what you do with `f`, you may get a nice shell error message like:

```
OURCES: No such file or directory
```

but you might just as easily get no message at all. Remember to surround your `make` variables with parentheses.

### missing separator

The message:

```
makefile:2:missing separator. Stop.
```

or:

```
makefile:2:missing separator (did you mean TAB instead of 8 spaces?).  Stop.
```

usually means you have a command script that is using spaces instead of tabs.

The more literal interpretation is that `make` was looking for a `make` separator such as `:`, `=`, or a tab, and didn't find one. Instead, it found something it didn't understand.

### commands commence before first target

The tab character strikes again!

This error message was first covered in the section "Parsing Commands" in Chapter 5. This error seems to appear most often in the middle of *makefile*s when a line outside of a command script begins with a tab character. `make` does its best to disambiguate this situation, but if the line cannot be identified as a variable assignment, conditional expression, or multiline macro definition, `make` considers it a misplaced command.

#### unterminated variable reference

This is a simple but common error. It means you failed to close a variable reference or function call with the proper number of parentheses. With deeply nested function calls and variable references, `make` files can begin to look like Lisp! A good editor that does parenthesis matching, such as Emacs, is the surest way to avoid these types of errors.

## Errors in Command Scripts

There are three common types of errors in command scripts: a missing semicolon in multiline commands, an incomplete or incorrect path variable, or a command that simply encounters a problem when run.

We discussed missing semicolons in the section "Good Coding Practices," so we won't elaborate further here.

The classic error message:

```
bash: foo: command not found
```

is displayed when the shell cannot find the command `foo`. That is, the shell has searched each directory in the `PATH` variable for the executable and found no match. To correct this error, you must update your `PATH` variable, usually in your *.profile* (Bourne shell), *.bashrc* (bash), or *.cshrc* (C shell). Of course, it is also possible to set the `PATH` variable in the *makefile* itself, and export the `PATH` variable from `make`.

Finally, when a shell command fails, it terminates with a nonzero exit status. In this case, `make` reports the failure with the message:

```
$ make
touch /foo/bar
touch: creating /foo/bar: No such file or directory
make: *** [all] Error 1
```

Here the failing command is `touch`, which prints its own error message explaining the failure. The next line is `make`'s summary of the error. The failing *makefile* target is shown in square brackets followed by the exit value of the failing program. Sometimes `make` will print a more verbose message if the program exits due to a signal, rather than simply a nonzero exit status.

Note also that commands executed silently with the `@` modifier can also fail. In these cases, the error message presented may appear as if from nowhere.

In either of these cases, the error originates with the program `make` is running, rather than `make` itself.

## No Rule to Make Target

This message has two forms:

```
make: *** No rule to make target XXX. Stop.
```

and:

```
make: *** No rule to make target XXX, needed by YYY. Stop.
```

It means that `make` decided the file *XXX* needed to be updated, but `make` could not find any rule to perform the job. `make` will search all the implicit and explicit rules in its database before giving up and printing the message.

There are three possible reasons for this error:

- Your *makefile* is missing a rule required to update the file. In this case, you will have to add the rule describing how to build the target.
- There is a typo in the *makefile*. Either `make` is looking for the wrong file or the rule to update the file specifies the wrong file. Typos can be hard to find in *makefile*s due to the use of `make` variables. Sometimes the only way to really be sure of the value of a complex filename is to print it out either by printing the variable directly or examining `make`'s internal database.
- The file should exist but `make` cannot find it either, because it is missing or because `make` doesn't know where to look. Of course, sometimes `make` is absolutely correct. The file is missing—perhaps you forgot to check it out of CVS. More often, `make` simply can't find the file, because the source is placed somewhere else. Sometimes the source is in a separate source tree, or maybe the file is generated by another program and the generated file is in the binary tree.

## Overriding Commands for Target

`make` allows only one command script for a target (except for double-colon rules, which are rarely used). If a target is given more than one command script, `make` prints the warning:

```
makefile:5: warning: overriding commands for target foo
```

It may also display the warning:

```
makefile:2: warning: ignoring old commands for target foo
```

The first warning indicates the line at which the second command script is found, while the second warning indicates the location of the original command script that is being overridden.

In complex *makefile*s, targets are often specified many times, each adding its own prerequisites. One of these targets usually includes a command script, but during development or debugging it is easy to add another command script without realizing you are actually overriding an existing set of commands.

For example, we might define a generic target in an include file:

```
# Create a jar file.
$(jar_file):
        $(JAR) $(JARFLAGS) -f $@ $^
```

and allow several separate *makefile*s to add their own prerequisites. Then in a *makefile* we could write:

```
# Set the target for creating the jar and add prerequisites
jar_file = parser.jar
$(jar_file): $(class_files)
```

If we were to inadvertently add a command script to this *makefile* text, `make` would produce the overriding warning.