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 component-wise 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.

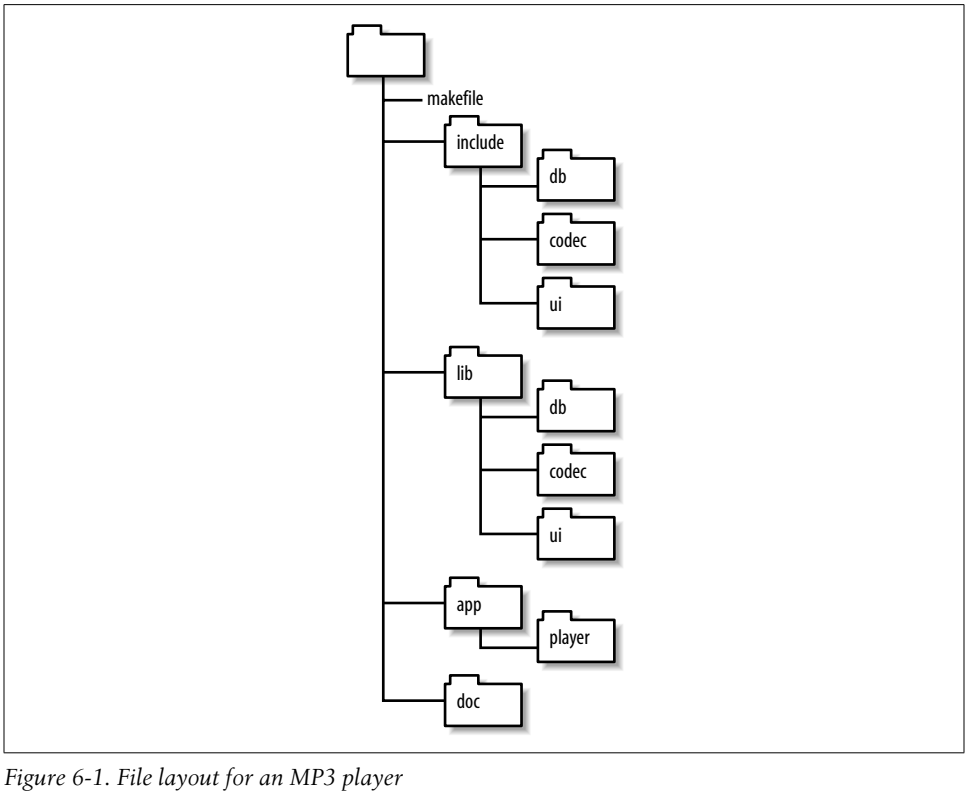

Figure 6-1. File layout for an MP3 player

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.

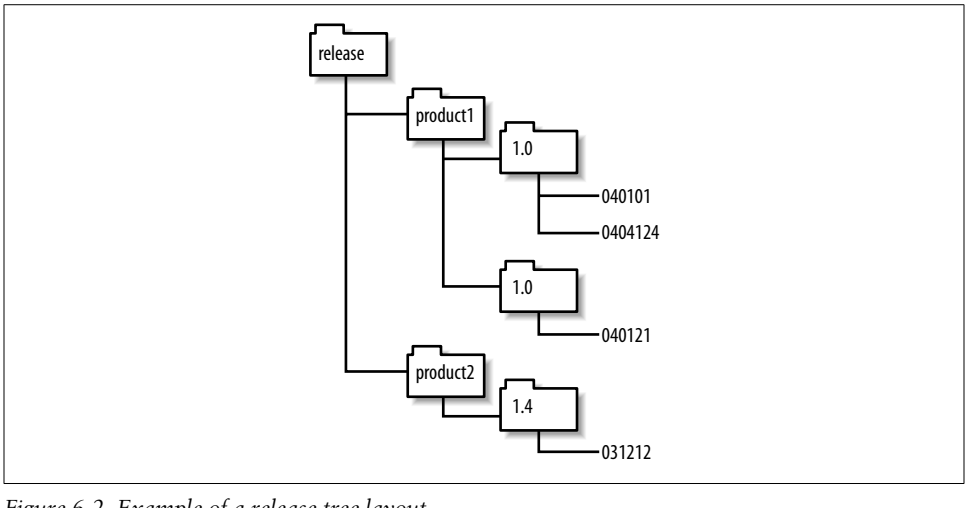

Figure 6-2. Example of a release tree layout

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.