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.

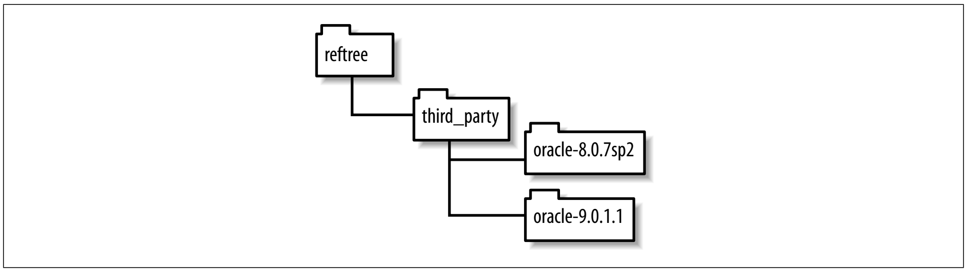

Figure 8-1. Directory layout for third-party libraries

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.