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.