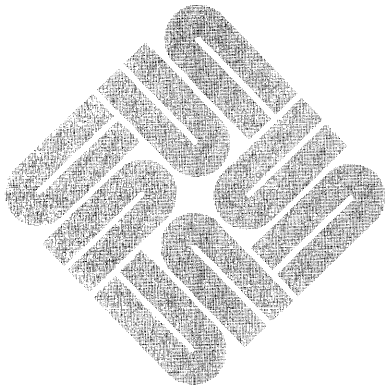
sun®
microsystems

Make User's Guide



Part Number: 800-1620-05
Revision: A of 14 May 1987

Credits

Make — A Program for Maintaining Computer Programs
by S.I. Feldman, Bell Laboratories, Murray Hill, New Jersey.

Trademarks

Sun Workstation is a trademark of Sun Microsystems Incorporated
UNIX is a trademark of AT&T Bell Laboratories

Copyright © 1987 by Sun Microsystems.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise, without prior explicit written permission from Sun Microsystems.

Contents

Chapter 1 `make` User's Guide **3**

1.1. Overview 3

Consistency Control 3

Dependency Checking: `make` vs. Shell Scripts 3

`make` Basics 4

Basic Use of Implicit Rules 4

Writing a Simple Makefile 5

Processing Dependencies 7

Missing Targets and Dependencies 10

Running Commands Silently 11

Ignoring a Command's Exit Status 11

Automatic Extraction of `sccs` Files 12

Suppressing `sccs` Extraction 13

Passing Parameters: Simple `make` Macros 13

Command Dependency Checking and `.KEEP_STATE` 15

Suppressing or Forcing Command Dependency Checking for Selected Lines 15

The State File 16

Hidden Dependencies and `.KEEP_STATE` 16

Displaying Information About a `make` Run 18

1.2. Compiling Programs With `make` 19

Compilation Strategies 19

A Simple Makefile 20

Using `make`'s Predefined Macros 20

Using Implicit Rules to Simplify a Makefile 22
Explicit Target Entries vs. Implicit Rules 24
Implicit Rules and Dynamic Macros 24
Dynamic Macro Modifiers 25
Dynamic Macros and the Dependency List: Delayed References 25
Adding Implicit Rules 26
Pattern Matching Rules: an Alternative to Implicit (Suffix) Rules 27
`make`'s Standard Implicit Rules and Predefined Macros 27
1.3. Building Object Libraries 30
Libraries, Members and Symbols 30
Library Members and Dependency Checking 30
Library Member Name Length Limit 31
`.PRECIOUS`: Preserving Libraries Against Removal Due to Interrupts 31
1.4. Maintaining Programs With `make` 32
Using Macros for Added Flexibility 32
Makefiles as Specifications 33
Suffix Replacement in Macro References 33
Using `lint` With `make` 34
Linking With System-Supplied Libraries 35
Compiling Programs for Debugging and Profiling 36
Conditional Macro Definitions 37
Maintaining a Directory of Header Files 40
Compiling and Linking With Your Own Libraries 41
Nested `make` Commands 41
Forcing A Nested `make` Command to Run 42
The `MAKEFLAGS` Macro 43
Macro Definitions and Environment Variables: Passing Parameters to Nested **`make`** Commands 44
Compiling Other Source Files 46
Compiling and Linking a C Program with Assembly Language Routines 47

Compiling `lex` and `yacc` Sources 49
Specifying Target Groups With the + Sign 50
Maintaining Shell Scripts with `make` and `sccs` 52
Running Tests with `make` 52
Delayed References to a Shell Variable 53
1.5. Maintaining Software Projects 54
Organizing A Project for Ease of Maintenance 54
Using `include` Makefiles 55
Installing Finished Programs and Libraries 56
Building the Entire Project 57
Maintaining Directory Hierarchies With Recursive Makefiles 58
Recursive `install` Targets 59
Maintaining A Large Library as a Hierarchy of Subsidiaries 61
In Conclusion 65

Appendix A `make` Enhancements Summary **69**
A.1. New Features 69
Default Makefile 69
The State File `.make.state` 69
Hidden Dependency Checking 69
Command Dependency Checking 69
Automatic `sccs` Extraction 70
Tilde Rules Superceded 70
`sccs` History Files 70
Pattern Matching Rules: Convenient Implicit Rules 70
New Options 71
Support for Modula-2 71
Naming Scheme for Predefined Macros 71
New Special-Purpose Targets 71
New Implicit Rule for `lint` 72
Macro Processing Changes 72
Macros: Definition, Substitution, and Substring Replacement 72

Improved *ar* Library Support 72
Lists of Members 72
Handling of `ar`'s Member-Name Length Limitation 73
Target Groups 73
A.2. Incompatibilities With Previous versions of `make` 73
New Meaning for `-d` Option 73
Dynamic Macros 73

Appendix B `make` Manual Page **77**
Index 89

Figures

Figure 1-1 Makefile Target Entry Format 5

Figure 1-2 A Trivial Makefile 6

Figure 1-3 A Simple Target Entry for Compiling A C Source File 7

Figure 1-4 Simple Makefile for Compiling C Sources: Everything Explicit 20

Figure 1-5 Makefile for Compiling C Sources Using Predefined Macros 22

Figure 1-6 Makefile for Compiling C Sources Using Implicit Rules 22

Figure 1-7 The Standard Suffixes List 23

Figure 1-8 Makefile With "Suffix-Replacement" Macro References 35

Figure 1-9 Makefile for a C Program With System-Supplied Libraries 36

Figure 1-10 Makefile for a C Program with Alternate Debugging and Profiling Variants 38

Figure 1-11 Makefile for a C Library with Alternate Variants 40

Figure 1-12 Makefile for C Program With User-Supplied Libraries 43

Figure 1-13 Makefile for a C Program with Assembly Routines 48

Figure 1-14 Makefile for Compiling C Programs With `lex` and `yacc` Sources 51

Figure 1-15 Recursive Makefile for Building a C Program and Subdirectories 61

Figure 1-16 Makefile for a Hierarchy of Subsidiary Libraries with Variants 65

Tables

Table 1-1 `make`'s Standard Implicit Rules 28
Table 1-2 `make`'s Predefined and Dynamic Macros 29
Table 1-3 Summary of Macro Assignment Order 46

1

make User's Guide

make User's Guide 3

1.1. Overview 3
Consistency Control 3
Dependency Checking: make vs. Shell Scripts 3
make Basics 4
Basic Use of Implicit Rules 4
Writing a Simple Makefile 5
Processing Dependencies 7
Missing Targets and Dependencies 10
Running Commands Silently 11
Ignoring a Command's Exit Status 11
Automatic Extraction of sccs Files 12
Suppressing sccs Extraction 13
Passing Parameters: Simple make Macros 13
Command Dependency Checking and .KEEP_STATE 15
Suppressing or Forcing Command Dependency Checking for Selected Lines 15
The State File 16
Hidden Dependencies and .KEEP_STATE 16
Displaying Information About a make Run 18
1.2. Compiling Programs With make 19
Compilation Strategies 19
A Simple Makefile 20
Using make's Predefined Macros 20

Using Implicit Rules to Simplify a Makefile 22
Explicit Target Entries vs. Implicit Rules 24
Implicit Rules and Dynamic Macros 24
Dynamic Macro Modifiers 25
Dynamic Macros and the Dependency List: Delayed References 25
Adding Implicit Rules 26
Pattern Matching Rules: an Alternative to Implicit (Suffix) Rules 27
make's Standard Implicit Rules and Predefined Macros 27
1.3. Building Object Libraries 30
Libraries, Members and Symbols 30
Library Members and Dependency Checking 30
Library Member Name Length Limit 31
.PRECIOUS: Preserving Libraries Against Removal Due to Interrupts 31
1.4. Maintaining Programs With make 32
Using Macros for Added Flexibility 32
Makefiles as Specifications 33
Suffix Replacement in Macro References 33
Using lint With make 34
Linking With System-Supplied Libraries 35
Compiling Programs for Debugging and Profiling 36
Conditional Macro Definitions 37
Maintaining a Directory of Header Files 40
Compiling and Linking With Your Own Libraries 41
Nested make Commands 41
Forcing A Nested make Command to Run 42
The MAKEFLAGS Macro 43
Macro Definitions and Environment Variables: Passing Parameters to Nested **make** Commands 44
Compiling Other Source Files 46
Compiling and Linking a C Program with Assembly Language Routines 47
Compiling lex and yacc Sources 49
Specifying Target Groups With the + Sign 50

Maintaining Shell Scripts with `make` and `sccs` 52
Running Tests with `make` 52
Delayed References to a Shell Variable 53
1.5. Maintaining Software Projects 54
Organizing A Project for Ease of Maintenance 54
Using `include` Makefiles 55
Installing Finished Programs and Libraries 56
Building the Entire Project 57
Maintaining Directory Hierarchies With Recursive Makefiles 58
Recursive `install` Targets 59
Maintaining A Large Library as a Hierarchy of Subsidiaries 61
In Conclusion 65

1

make User's Guide

.1. Overview

his manual describes Sun's nhanced version of make, which ıcludes new features such as hid- en dependency checking, com- ıand dependency checking, and utomatic extraction of sccs files. is highly compatible with ıakefiles written for previous ver- ons. Makefiles that rely on Sun's nhancements may not be compati- le with other versions of make. efer to *Appendix A* for a complete ummary of enhancements.

make streamlines the process of generating and maintaining object files and executable programs. It helps you to compile programs consistently, and eliminates unnecessary compilation of modules that are unaffected by source code changes.

make provides a number of features that simplify compilations, but you can also use it to automate any complicated or repetitive task that isn't interactive. For instance, you can use *make* to update and maintain object libraries, run test suites, and install validated files onto a filesystem or tape. In conjunction with sccs, you can use make to insure that all programs are built from the most recent source versions. You can also use make and sccs to build an entire software project, and to maintain the source files and directories from which that project is built.

:onsistency Control

sccs provides facilities for version control over source files. These include file locking, audit trails and commentary, among others, as described in Section 4 of *Programming Utilities for the Sun Workstation.*

make provides facilities for consistency control over the object files or other files derived from those sources. It rebuilds the files, in a modular and consistent fashion, when the source files they derive from have changed.

make reads a file that you create, called a *makefile*, which contains information about what files to build and how to build them. Once you write and test the makefile, you can forget about the processing details; make takes care of them. This gives you more time to concentrate on debugging and correcting your code; the repetitive portion of the maintenance cycle is reduced to:

think — edit — **make** — test . . .

)ependency Checking: make s. Shell Scripts

While it is possible to use a shell script to assure consistency in trivial cases, scripts are often inadequate in actual practice. On the one hand, you don't want to wait for a simple-minded script to compile every single program or object module when only one of them has changed. On the other hand, having to edit the script for each iteration can defeat the objective of consistent compilation. Although it is possible to write a script of sufficient complexity to process only those modules that require it, such scripts can often develop maintenance problems of their own. In any case, make eliminates the need for you to do so.

make assumes that only it will make changes to files being processed during the current make run. If a source file changes in the middle of the run, the files make produces may be in an inconsistent state.

make allows you to write a simple, structured listing of what to build and how to build it. It uses the mechanism of *dependency checking* to compare each module with the source files or intermediate files it derives from. make only rebuilds a module if one or more of these prerequisite files, called *dependency files*, has changed since the module was last built. To determine whether a derived file is out of date with respect to its sources, make compares the modification time of the module with that of the source file. If the module is missing, or if it is older than the source file, it is considered to be out of date; make issues the commands necessary to rebuild it. Optionally, a target can be treated as out of date if the commands used to build it have changed.

Because make does a complete dependency scan, changes to a source file are consistently propagated through any number of intermediate files or processing steps. This lets you specify a hierarchy of processing steps in a top-down fashion.

make Basics

You can think of a makefile as a type of recipe. make reads the recipe, decides which steps need to be performed, and executes only those steps that are required to produce the finished product. Each file to build, or step to perform, is called a *target*. The makefile entry for a target contains its name, and a list of commands for building it called a *rule*, along with a list of dependencies. make treats dependencies as prerequisite targets, and updates them if necessary, before processing the target that depends on them.

The file for which the target is named is also referred to as a *target file*. Each file from which a target is derived (or that the target depends on) is called a *dependency file* with respect to that target.

Basic Use of Implicit Rules

In addition to any makefile(s) that you supply, make reads in the default makefile, /usr/include/make/default.mk, which contains target entries for *implicit rules*, as well as other information.[1] An implicit rule, also known as a *suffix* rule, is a generic rule for building a target file ending in one suffix, from a dependency file ending in a different suffix. In some cases, the use of implicit rules can eliminate the need for writing a makefile entirely. For instance, to build an object module named go.o from a single C source file named go.c, you could use the command

```
make go.o
```

as shown:

```
hermes% make go.o
cc   -mc68020 -c go.c -o go.o
```

This would work equally well for building the object file nonesuch.o from the source file nonesuch.c.

[1] Implicit rules were hard-coded in earlier versions of make.

To build an executable file named go (with a null suffix) from go.c, you need only type the command:

```
make go
```

as shown:

```
hermes% make go
cc   -mc68020  -o go go.c
```

The rule for building a .o file from a .c file is called the .c.o (pronounced "dot-c-dot-o") rule. The rule for building an executable file from a .c file is called the .c (dot-c) rule. An implicit rule is named for the target entry in the default makefile that contains it. The complete set of implicit rules is listed below in this chapter.

Implicit rules eliminate the need to duplicate target entries with frequently used compilation commands, such as those shown above. In most cases, judicious use of implicit rules makes for shorter, more readable makefiles.

Writing a Simple Makefile

The basic format for a makefile target entry is:

f there is no rule for a target entry, nake looks for an implicit rule to ıse.

```
target ... : [ dependency ... ]
             [ command ]
             ...
```

Figure 1-1 *Makefile Target Entry Format*

f the dependency list is terminated vith a semicolon and followed by a :ommand, that command is ncluded in the rule. However, nakefiles tend to read better if you ıvoid this.

In the first line, the target name (or list) is followed by a colon, which is required. This, in turn, is followed by the dependency list if there is one. Several target names separated by white space can precede the colon; this indicates a list of independent targets that are built using the same dependency list and rule.

Command lines in a rule start with a TAB; leading spaces are no substitute as far as make is concerned.

Subsequent lines that start with a TAB are taken as the commands lines that comprise the target's rule. make is awfully fussy about those leading TAB's, SPACE characters *simply won't do.*

Lines that start with a # are treated as comments up until the next (unescaped) NEWLINE, and do not terminate the target entry. The target entry is terminated by the next nonempty line that begins with a character other than TAB or #, or by the end of the file.

A trivial makefile might consist of just one target:

```
test:
        ls test
        touch test
#       'test' is now present
        ls test
```

Figure 1-2 *A Trivial Makefile*

The convention is to use the name `Makefile`, since filenames starting with a capital are listed first by `ls`; this highlights the fact that a makefile is present.

When you run `make` with no arguments, it searches first for a file named `makefile`, or if there is no file by that name, `Makefile`. If either of these files is under `sccs` control, `make` extracts the current version and uses it.

If `make` finds a makefile, it begins the dependency check with the first target entry in that file. Otherwise you must list the targets to build as arguments on the command line. `make` displays each command it runs while building its targets.

```
hermes% make
ls test
test not found
touch test
ls test
test
```

Because the file `test` was not present (and therefore out of date), `make` performed the rule in its target entry. If you run `make` a second time, it issues a message indicating that the target is now up to date:

```
hermes% make
'test' is up to date.
```

and doesn't perform the rule.

`make` invokes a Bourne shell to process a command line if that line contains any shell metacharacters, such as a semicolon (`;`), redirection symbol (<, >, >>, ...) or pipe symbol (`|`), etc. If a shell isn't required to parse the command line, etc. *make* invokes the command directly for better performance.

Line breaks within a rule are significant in that each command line is performed by a separate process or shell.

This means that a rule such as:

```
test:
        cd /tmp
        pwd
```

behaves differently than you might expect, as shown below.

```
hermes% make test
cd /tmp
pwd
/usr/hermes/waite/arcana/minor/pentangles
```

You can use semicolons to specify a sequence of commands to perform in a single shell invocation:

```
test:
        cd /tmp ; pwd
```

Or, you can continue the input line onto the next line in the makefile by escaping the [NEWLINE] with a backslash (\):

```
test:
        cd /tmp ; \
        pwd
```

Here is an example of a simple target entry to compile a C program from a single source file:

This entry performs the same function with respect to go as in the second example of implicit rules shown above; it compiles an executable program from a C source file.

```
go: go.c
        cc -mc68020 -o go go.c
```

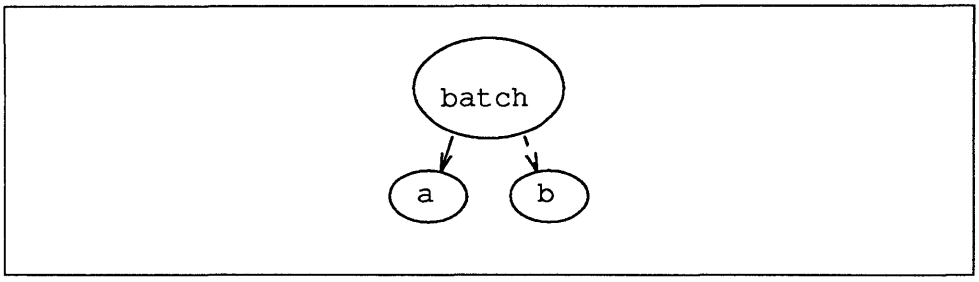
Figure 1-3 *A Simple Target Entry for Compiling A* C *Source File*

Processing Dependencies

Once make begins, it processes targets as it encounters them in its depth-first dependency scan. For example, with the following makefile:

```
batch: a b
        touch batch
b:
        touch b
a:
        touch a
c:
        echo "you won't see me"
```

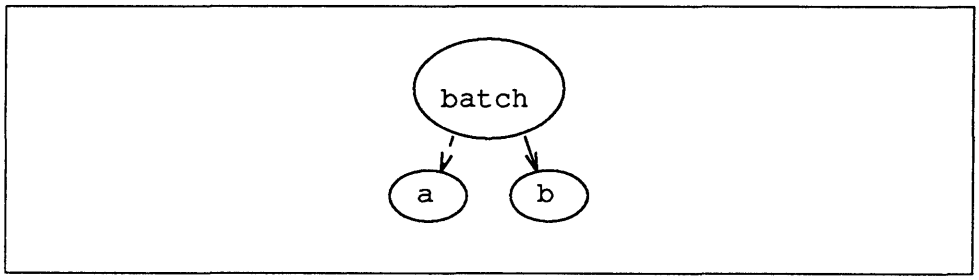
make starts with the target batch. Since batch has some dependencies that haven't been checked yet, namely a and b, make defers checking batch until after it has checked each of them against any dependencies they might have.



Since a has no dependencies, make processes it; if the file is not present make performs its rule.

```
hermes% make
touch a
. . .
```

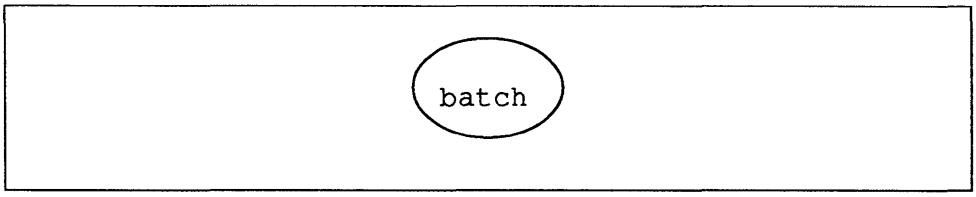
Next, make works its way back up to the parent target batch. Since there is still an unchecked dependency b, make descends to b and checks it.



b also has no dependencies, so make processes it:

```
. . .
touch b
. . .
```

Finally, now that all of the dependencies for batch have been checked and built if needed, make checks it against those dependency files:



Since both a and b were built just now, and are therefore newer than batch, make builds it:

```
. . .
touch batch
```

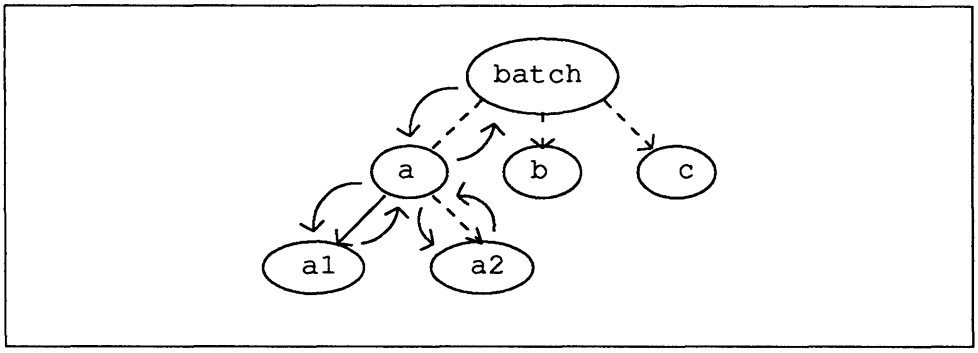
Although there is a target entry for c in the makefile, make does not encounter it while performing its dependency scan. Target entries that aren't encountered in the dependency scan are omitted from processing. You can select a starting target like c by entering it as an argument to the make command:

```
hermes% make c
echo "you won't see me"
you won't see me
```

In the next example, batch is used to group a set of targets.

```
batch:  a b c

a:   a1  a2
        touch a
b:
        touch b
c:
        touch c
a1:
        touch a1
a2:
        touch a2
```

In this case, the targets are checked and processed as shown in the following diagram:



1. make checks batch, for dependencies and notes that there are three, so it defers processing it.
2. make checks a, the first dependency, and notes that it has two dependencies of its own. So, continuing in the same fashion, make:
   - Checks a1, and if necessary, rebuilds it.
   - Checks a2, and rebuilds it if necessary.
   - Determines whether to build a.
3. make checks b and rebuilds it if need be, and then:

4. Checks and rebuilds c if needed.
5. After processing all of these nested dependencies, make checks and processes the topmost target, batch.

Missing Targets and Dependencies

If a target entry contains no rule, make attempts to select an implicit rule to build it. If make cannot find an appropriate rule to apply and there is no sccs file to extract it from, make presumes that the target has an *empty* rule, and continues processing subsequent targets. With this makefile:

```
void:
```

make produces:

```
hermes% make void
hermes%
```

make stops processing and issues an error message if the target was named either on the command line or in a dependency list but it:

- is missing,
- has no target entry,
- no implicit rule can be used to build it, and
- there is no sccs file to extract it from.

The following command produces:

```
hermes% make believe
make: Fatal error: Don't know how to make target 'believe'.
```

On the other hand, if the target entry has no rule, and make encounters the target in a dependency list, it does *not* produce an error, either when processing the dependency, or when processing the target for which it is a dependency. This holds true, even if the dependency file is *absent.*

make *finds* a target entry for the dependency. It *executes* the (null) rule for that dependency without encountering errors. So, thatmakeconcludes the time that the (null) rule is performed. The dependency is considered *newer* than the target, even though no dependency *file* results from performing the rule. In a case such as this, make simply goes on to rebuild the parent target (after processing any remaining dependencies). With this makefile:

You can use a dependency with a null rule to force the target's rule to be executed. The conventional name for such a dependency is FRC.

```
haste: FRC
        echo "haste makes waste"
FRC:
```

make performs the rule for making haste, even if a file by that name is up to date:

```
hermes% touch haste
hermes% make haste
echo "haste makes waste"
haste makes waste
```

Running Commands Silently

You can inhibit the display of a given command line by inserting an @ as the first non-TAB character on that line. For example, the following target:

```
quiet:
        @ echo you only see me once
```

produces:

```
hermes% make quiet
you only see me once
```

If you want to inhibit the display during a particular make run, you can use the -s option. If you want to inhibit the display of all command lines in every run, add the special target .SILENT to your makefile:

Special-function targets begin with a dot (.). Target names that begin with a dot are never used as the starting target, unless specifically requested as an argument on the command line.

```
.SILENT:
quiet:
        echo you only see me once
```

Ignoring a Command's Exit Status

make normally issues an error message and stops when a command returns a nonzero exit code. For example, if you have the target:

```
rmxyz:
        rm xyz
```

and there is no file named xyz, make halts after rm returns its exit status.

```
hermes% ls xyz
xyz not found
hermes% make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1
make: Fatal error: Command failed for target 'rmxyz'
```

If - and @ are the first two such characters, both take effect.

To continue processing regardless of the command's exit code, use a dash character (-) as the first non-TAB character:

```
rmxyz:
        -rm xyz
```

In this case you get a warning message indicating the exit code make received:

```
hermes% make rmxyz
rm xyz
rm: xyz: No such file or directory
*** Error code 1 (ignored)
```

Unless you are testing a makefile, it is usually a bad idea to ignore non-zero error codes on a global basis. Specific commands that return non-zero status can be ignored in certain circumstances. But, in general, a non-zero exit code indicates trouble. It is best for make to stop so that you can diagnose the problem right away.

Although it is generally ill-advised to do so, you can have make ignore error codes entirely within a run with the -i option. You can also have make ignore exit codes when processing a given makefile, by adding the special target .IGNORE to your makefile, although this too should normally be avoided.

```
.IGNORE:
rmxyz:
        rm xyz
```

If you are processing a list of targets, and you want make to continue with the next target on the list, rather than stopping entirely after encountering an non-zero return code, use the -k option.

Automatic Extraction of sccs Files

When source files are named in the dependency list, make treats them just like any other target file. Because the source file is presumed to be present in the directory, there is no need to add an entry for it to the makefile. When a target has no dependencies, but is present in the directory, make assumes that that file is up to date. If, however, a source file is under sccs control, make does some additional checking to assure that the source file is the version most recently checked in. If the file is missing, or if there is a new version has been checked in, make automatically issues an sccs get -G*filename filename* command to extract the most recent version:[2] If, however, the source file is writable by anyone, make does not extract it.

[2] With other versions of make automatic sccs extraction was a feature only of certain implicit rules. Also, unlike earlier versions, make only looks for history (s.) files in the SCCS subdirectory; s. files in the current directory are ignored.

```
hermes% ls SCCS
SCCS/s.go.c
hermes% rm -f go.c
hermes% make go
sccs get go.c -Ggo.c > /dev/null
cc -mc68020 -o go go.c
```

This makes it unnecessary to add sccs commands for extracting current versions of source files; make handles this for you automatically.

Suppressing sccs Extraction

The command for extracting sccs files is specified in the rule for the .SCCS_GET special target in the default makefile. To suppress automatic extraction, simply add an entry for this target, without any rule, to your makefile:

```
#       Suppress sccs extraction.
.SCCS_GET:
```

Passing Parameters: Simple make Macros

make's macro substitution comes in handy when you want to pass parameters to commands lines within a makefile. Suppose that you sometimes wish to compile an optimized version of the program go using cc's -O option. You can lend this sort of flexibility to your makefile by adding a *macro reference*, such as the one below, to the target for go:

```
go: go.c
        cc -mc68020 $(CFLAGS) -o go go.c
```

The macro reference acts as a placeholder for a value that you define, either in the makefile itself, or as an argument to the make command. If you then supply make with a *definition* for the CFLAGS macro, make replaces the macro reference with the value you have defined.

There is a reference to the CFLAGS macro in both the .c and the .c.o implicit rules.

```
hermes% rm go
hermes% make go "CFLAGS= -O"
cc -mc68020 -O -o go go.c
```

If a macro is undefined, make replaces references to it with an empty string:

```
hermes% rm go
hermes% make go
cc -mc68020 -o go go.c
```

You can also include macro definitions in the makefile itself. A typical use is to set CFLAGS to -O so that make produces optimized object code by default, as shown below.

```
CFLAGS= -O
go: go.c
        cc -mc68020 $(CFLAGS) -o go go.c
```

With no arguments, the make command produces:

```
hermes make
cc -mc68020 -O -o go go.c
```

A macro definition supplied as an argument to make overrides all other definitions for that macro found in that make run. For instance, to compile go for debugging with dbx or dbxtool, you can define the value of CFLAGS to be -g in the make command:

```
hermes% rm go
hermes% make CFLAGS=-g
cc -mc68020 -g -o go go.c
```

To compile a profiling version for use with gprof, supply both -O and -pg in the value for CFLAGS:

```
hermes% rm go
hermes% make "CFLAGS= -O -pg"
cc -mc68020 -O -pg -o go go.c
```

A macro reference must include parentheses when the name of the macro is longer than one character. If the macro name is only one character, the parentheses can be omitted. Also, you can use curly braces, { and }, instead of parentheses. For example:

```
S= @ echo now and forever
.SILENT:
when:
        $S
        $(S)
        ${S}
```

are all three equivalent:

```
hermes% make when
now and forever
now and forever
now and forever
```

Command Dependency Checking and .KEEP_STATE

In addition to the normal dependency checking, you can use the special target .KEEP_STATE to activate *command dependency* checking.[3] When activated, make not only checks each target file against its dependency files, it compares each command line in the rule with the corresponding command line it ran the last time it built the target. (This information is stored in a state file in the current directory.) If the command line has changed, make rebuilds the target. So, if .KEEP_STATE were in effect for the previous few examples, you wouldn't have had to type in all those rm go commands.

With the makefile:

```
CFLAGS= -O
.KEEP_STATE:
go: go.c
        cc -mc68020 -o go go.c
```

the following commands work as shown:

```
hermes% make
cc -mc68020 -O -o go go.c
hermes% make CFLAGS=-g
cc -mc68020 -g -o go go.c
hermes% make "CFLAGS= -O -pg"
cc -mc68020 -O -pg -o go go.c
```

This assures you that make compiles a program with the options you want, even if a different variant of the file is present and newer than its dependencies.

The first make run with .KEEP_STATE in effect recompiles all targets. This insures that they have, in fact, been built by the command line reported in the state file.

Suppressing or Forcing Command Dependency Checking for Selected Lines

To inhibit command dependency checking for a given command line, insert a question mark as the first character after the TAB. For instance, without the question mark, this makefile:

```
ARG= redone or not
.KEEP_STATE:
x:
        echo $(ARG) | tee x
```

reprocesses x when you define ARG on the command line, as shown below.

[3] This feature is not available in earlier versions of make.

```
hermes% make x
echo redone or not | tee x
redone or not
hermes% make x "ARG= redone this time"
echo  redone this time | tee x
redone this time
```

Adding a ? as the first character after the TAB suppresses command dependency checking.

```
ARG= is it redone
.KEEP_STATE:
x:
        ? echo $(ARG) | tee x
```

With it, x is not reprocessed as a result of changing ARG, as shown:

```
hermes% make x
echo is it redone | tee x
is it redone
hermes% make x "ARG= still not redone"
'x' is up to date.
```

Command dependency checking is automatically suppressed for lines containing the dynamic macro $?, This macro stands for the list of dependencies that are newer than the current target, and can be expected to differ between any two make runs. (See *Implicit Rules and Dynamic Macros* for more information.) To force make to perform command dependency checking on a line containing this macro, prefix the command line with a ! character (following the TAB).

The State File

When the .KEEP_STATE special target is in effect, make writes out a state file named .make.state, in the current directory. This file lists all targets that have ever been processed while .KEEP_STATE has been in effect, in a format similar to a makefile. In order to assure that this state file is maintained consistently, once you have added the .KEEP_STATE special target to a makefile, we recommend that you leave it in effect.[4]

Hidden Dependencies and .KEEP_STATE

When a source file contains #include directives for interpolating header files, the target depends just as much on those header files as it does on the sources that include them. Because such header files may not be listed explicitly as sources in the compilation command line, they are called *hidden dependencies*. When .KEEP_STATE is in effect, make receives a report from the various compilers and compilation preprocessors indicating which hidden dependency files were interpolated for each target.[5] It adds this information to the dependency list in the

[4] Since this target is ignored in earlier versions of make, it does not introduce any compatibility problems. Other versions simply treat it as a superfluous target that no targets depend on, with an empty rule and no dependencies of its own. Since it starts with a dot, it is not used as the starting target.

[5] Also unavailable with earlier versions of make.

state file. In subsequent runs, these additional dependencies are processed just like regular dependencies. This feature maintains the hidden dependency list for each target automatically; this insures that the dependency list for each target is always accurate and up to date. It also eliminates the need for the complicated schemes found in some earlier makefiles to generate complete dependency lists.

A slight inconvenience can arise the first time `make` processes a target with hidden dependencies, because there is as yet no record of them in the state file. If a header file is missing, and `make` has no record of it, `make` won't know that it needs to extract it from `sccs`, before compiling the target. So, even though there is an `sccs` history file, the current version won't be extracted because it doesn't yet appear in a dependency list or the state file. So, when the C preprocessor attempts to interpolate the header, it won't find it; the compilation fails.

Supposing that an `#include` directive for interpolating the header file `hidden.h` is added to `go.c`, and that the file `hidden.h` is somehow removed before the subsequent `make` run. The results would be:

```
hermes% make go
cc -mc68020 -O -o go go.c
go.c: 2: Can't find include file hidden.h
make: Fatal error: Command failed for target 'go'
. . .
```

The workaround is simple. Just make sure that the new header file is present in the directory before you run `make`. Or, if the compilation should fail (and assuming the header file is under `sccs`), extract it from `sccs` manually:

```
hermes% sccs get hidden.h
1.1
10 lines
hermes% make go
cc -mc68020 -O -o go go.c
```

In future cases, should the header file turn up missing, `make` will know to build or extract it for you, because it will be listed in the state file as a hidden dependency:

```
hermes% rm go hidden.h
hermes% make go
sccs get hidden.h -Ghidden.h > /dev/null
cc -mc68020 -O -o go go.c
```

Displaying Information About a make Run

There is an exception to this however. make executes any command line containing a reference to the MAKE macro (i.e., $(MAKE) or ${MAKE}), regardless of -n. So, it would be a very bad idea to include a line like: "$(MAKE) ; rm -f *" in your makefile.

Running make with the -n option displays the commands make is to perform, without executing them. This comes in handy when verifying that the macros in a makefile are expanded as expected. With the following makefile:

```
CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

program: main.o data.o
        $(LINK.c) -o program main.o data.o
```

make -n displays:

```
hermes% make -n
cc -mc68020 -c main.c
cc -mc68020 -c data.c
cc -mc68020 -o program main.o data.o
```

make has some other options that you can use to keep abreast of what it's doing and why:

Setting an environment variable named MAKEFLAGS can lead to complications, since make adds its value to the list of options. To prevent puzzling surprises, avoid setting this variable.

-d Displays the criteria by which make determines that a target is be out-of-date. Unlike -n, it *does* process targets, as shown below. This options also displays the value imported from the environment (null by default) for the MAKEFLAGS macro, which is described in detail in a later section.

```
hermes% make -d
MAKEFLAGS value: (null)
Building main.o using suffix rule .c.o because it is out of date relative to main.c
cc -mc68020 -c main.c
Building data.o using suffix rule .c.o because it is out of date relative to data.c
cc -mc68020 -c data.c
Building program using explicit rule because it is out of date
cc -mc68020 -o program main.o data.o
```

-dd This option displays all dependencies make checks in vast detail.

-D Displays the text of the makefile as it is read.

-DD Displays the makefile and the default makefile, the state file, and hidden dependency reports for the current make run.

Several -f options indicate the concatenation of the named makefiles.

-f *makefile*
make uses the named *makefile* (instead of makefile or Makefile).

-p Displays the complete set of macro definitions and target entries.

-P Displays the complete dependency tree for each target encountered.

Due to its potentially troublesome side effects, we recommend against using the -t (touch) option for make.

There is an option that can be used to shortcut make processing, the -t option. When run with -t, make does not perform the rule for building a target. Instead it uses touch to alter the modification time. for each target that it encounters in the dependency scan. This often creates more problems than it supposedly solves, and so we recommend that you exercise extreme caution if you do use it. If there is no file corresponding to a target entry touch creates it.

The following is one example of how *not* to use make -t. Suppose you have a target named clean that performed housekeeping in the directory by removing target files produced by make:

```
clean:
    rm program main.o data.o
```

If you give the command:

```
hermes% make -t clean
touch clean
hermes% make clean
'clean' is up to date.
```

you then have to remove the file clean before your housekeeping target can work once again.

For a complete listing of all make options, refer to make(1) in the *Commands Reference Manual.*

1.2. Compiling Programs With make

Compilation Strategies

In previous examples you have seen how to compile a simple C program from a single source file, using both explicit target entries and implicit rules. Most C programs, however, are compiled from several source files. Many include library routines, either from one of the standard system libraries or from a local library. Although it may be easier to recompile and link a single-source program using a single cc command, it is usually more convenient to compile programs with multiple sources in stages—first, by compiling each source file into a separate object (.o) file, and then by linking the object files to form an executable program (an a.out format file). This method requires more disk space, but subsequent (repetitive) recompilations need be performed only on those object files for which the sources have changed. The time saved is usually worth the extra space required, since the remaining, up-to-date, object files are simply relinked as is into a newly produced executable program.

The makefile that follows compiles an executable program from two C source files. In subsequent examples, this makefile will be refined and enhanced to take advantage of make's predefined macros and implicit rules. Subsequent sections describe the mechanics of implicit rules, including how to add new ones of your own.

Then, additional features are introduced that are useful in makefiles for maintaining C object libraries. Later sections expand upon these examples to create sophisticated templates that are easily modified to handle a variety of programs or libraries.[6]

Further examples illustrate template makefiles for more complex operations, such as linking programs with with user-supplied object libraries (from other directories), linking C programs with assembly language routines, and compiling programs from `lex` and `yacc` sources.

A Simple Makefile

This makefile is not very flexible or elegant, but it does the job:

```
#       Simple makefile for compiling a program from
#       two C source files.

.KEEP_STATE:

program: main.o data.o
        cc -mc68020 -o program main.o data.o

main.o: main.c
        cc -mc68020 -O -c main.c

data.o: data.c
        cc -mc68020 -O -c data.c

clean:
        rm program main.o data.o
```

Figure 1-4 *Simple Makefile for Compiling* C *Sources: Everything Explicit*

In this example, the command

```
make
```

produces the object files `main.o` and `data.o`, and the executable file `program`.

Conventions have evolved for the use of certain target names, such as `all`, `clean` (and `install`, among others). There may be other conventions in your organization. In general, it is a good idea to avoid creating files by any such name in your source directories.

The last target, `clean`, removes these files. This is a common addition to simplify housekeeping chores. The name `clean` is a convention for targets that removes derived files.

Using `make`'s Predefined Macros

The next example performs exactly the same function, but demonstrates the use of `make`'s predefined macros for the indicated compilation commands. Using predefined macros eliminates the need to edit makefiles when the underlying compilation environment changes. They also provide access to the `CFLAGS`

[6] Makefiles for programs and libraries written in other compiled languages, such as FORTRAN 77, Pascal, and Modula-2, are analogous.

macro (and other FLAGS macros) for supplying compiler options from the command line. Predefined macros are also used extensively within make's implicit rules. The predefined macros in the following makefile are listed below.[7] They are generally useful for compiling C programs.

Macro names that end in the string FLAGS are used to pass options to a related compiler-command macro. It is good practice to use these macros for consistency and portability. It is also good practice to note the desired default values for the appropriate *FLAGS* macros in the makefile.

The complete list of all predefined macros is shown in Table 1.2, below.

COMPILE.c — The complete cc command line; composed of the values of CC, CFLAGS, CPPFLAGS, and TARGET_ARCH, as follows, along with the -c option.

```
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
```

The root of the macro name, COMPILE, is a convention used to indicate that the macro stands for an entire compilation command line. The .c suffix is a mnemonic device to indicate that the command line applies to .c (C source) files.

LINK.c — The complete cc command line to link object files: its value is similar to that of COMPILE.c, minus the reference to CPPFLAGS and the -c option, and with the addition of a reference to the LDFLAGS macro:

```
LINK.c=$(CC) $(CFLAGS) $(LDFLAGS) $(TARGET_ARCH)
```

CC — The value cc. (You can redefine the value to be the pathname of an alternate C compiler.)

CFLAGS — Options for the cc command; none by default.

CPPFLAGS — Options for cpp; none by default.

LDFLAGS — Options for the link editor, ld; none by default.

TARGET_ARCH — The target-architecture argument to cc used for cross-compiling. Refer to *Cross-Compilation on the Sun Workstation* for details.

[7] Predefined macros are used more extensively than in earlier versions of make. Not all of the predefined macros shown here are available with earlier versions.

```
#       Makefile for compiling two C sources
#       using predefined macros.

CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

program: main.o data.o
        $(LINK.c) -o program main.o data.o

main.o: main.c
        $(COMPILE.c) main.c

data.o: data.c
        $(COMPILE.c) data.c

clean:
        rm program main.o data.o
```

Figure 1-5 *Makefile for Compiling* C *Sources Using Predefined Macros*

Using Implicit Rules to Simplify a Makefile

The command lines for compiling `main.o` and `data.o` from their respective `.c` files are now functionally equivalent to the `.c.o` implicit rule. Since this is so, then for all intents and purposes they are redundant; `make` performs the same compilation whether they appear in the makefile or not. This next version of the makefile takes advantage of `make`'s implicit rules to perform the compilation.

```
#       Makefile for a program from two C sources
#       using implicit rules.

CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

program: main.o data.o
        $(LINK.c) -o program main.o data.o

clean:
        rm program main.o data.o
```

Figure 1-6 *Makefile for Compiling* C *Sources Using Implicit Rules*

A complete list of implicit rules appears in Table 1-1.

As `make` processes the dependencies `main.o` and `data.o`, it finds no target entries for them. So, it checks for an appropriate implicit rule to apply. In this case, `make` selects the `.c.o` rule for building a `.o` file from a dependency file that has the same basename and a `.c` suffix.

make uses the order of appearance in the suffixes list to determine which dependency file and implicit rule to use. For instance, if there were both main.c and main.s files in the directory, make would use the .c.o rule, since .c is ahead of .s in the list.

First, make scans its suffixes list to see whether or any the suffix for the target file appears. In the case of main.o, the string .o appears in the list. Next, make checks for an implicit rule to build it with, and a dependency file to build it from. The dependency file has the same basename as the target, but a different suffix. In this case, while checking the .c.o rule, make finds a dependency file named main.c, so it selects that rule. The target entry for the implicit rule is named for the dependency suffix and the target suffix; the name is composed of the two suffixes, in this case the target name is .c.o; make uses the rule in this entry to build the target.

The suffixes list is a special-function target named .SUFFIXES. The various suffixes are included in the definition for the SUFFIXES macro; the dependency list for .SUFFIXES is given as a reference to this macro:

```
SUFFIXES= .o .c .c~ .s .s~ .S .S~ .ln .f .f~ .F .F~ .l .l~ \
        .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .y .y~ .h .h~ .sh .sh~

.SUFFIXES:  $(SUFFIXES)
```

Figure 1-7 *The Standard Suffixes List*

The following example shows a makefile for compiling a whole set of executable programs, each having just one source file. Each executable is to be built from a source file that has the same basename, and the .c suffix appended. For instance demo_1 is built from demo_1.c.

Like clean, all is a target name used by convention. It builds "all" the targets in its dependency list. Normally, all is the first target; make and make all are usually equivalent.

```
#       Makefile for a set of C programs, one source
#       per program.  The source file names have ".c"
#       appended.

CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

all: demo_1 demo_2 demo_3 demo_4 demo_5
```

In this case, make does not find a suffix match for any of the targets (demo_1 through demo_5). So, it treats each as if it had a null suffix. It then searches for an implicit rule and dependency file with a valid suffix. In the case of demo_2, it would find a file named demo_2.c. Since there is a target entry for a .c(null) rule, namely the .c rule, along with a corresponding .c file make uses the rule in the .c target entry to build demo_2 from demo_2.c.

There is no transitive closure for implicit rules. If you had an implicit rule for building, say, a .y file from a .x file, and another for building a .z file from a .y file, make would not combine their rules to build a .z file from a .x file. You must specify the intermediate steps as targets, as in the next example.

```
hermes% ls mcp.[xyz]
mcp.x
hermes% make mcp.z
Don't know how to make mcp.z.  Stop.
hermes% make mcp.y mcp.z
cp mcp.x mcp.y
cp mcp.y mcp.z
```

Explicit Target Entries vs. Implicit Rules

Whenever you build a target from multiple dependency files, you must provide make with an explicit target entry that contains a rule for doing so. When building a target from a single dependency file, it is often convenient to use an implicit rule.

As the previous examples show, make is happy to compile a single source file into a corresponding object file or executable. However, it has no built-in knowledge whatsoever about how to collate several files into one. For instance, it has no idea of the order in which to link a list of object files into an executable program. Also, make only compiles those object files that it encounters in its dependency scan. It needs a starting point—a target for which each object file in the list (and ultimately, each source file) is a dependency.

So, for a target built from multiple dependency files, make needs an explicit rule that provides a collating order, and a dependency list that accounts for all of its dependency files. On the other hand, if each of those dependency files is built from just one source, you could use an implicit rule to build them.

Implicit Rules and Dynamic Macros

make maintains a set of macros dynamically, on a target-by-target basis. These macros are used quite extensively, especially in the definitions of implicit rules. So, it is important to understand what they mean.

Because they aren't explicitly defined in a makefile, the convention is to document dynamic macros with the $-sign prefix attached (in other words, by showing the macro reference).

They are:

$@ The name of the current target.

$? The list of dependencies newer than the target.

$< The name of the dependency file, as if selected by make for use with an implicit rule.

$* The basename of the current target (the target name stripped of its suffix).

$% For libraries, the name of the member being processed. See *Building Object Libraries,* below, for more information.

Implicit rules make use of these dynamic macros in order to supply the name of a target or dependency file to a command line within the rule itself. For instance, in the .c.o rule, shown in the next example.

The macro OUTPUT_OPTION has an empty value by default. While similar to CFLAGS in function, it is provided as a separate macro, intended for passing in the -o *filename* compiler option, as needed, to force compiler output to a given filename.

```
.c.o:
        $(COMPILE.c) $< $(OUTPUT_OPTION)
```

$< is replaced by the name of the dependency file (in this case the .c file) for the current target.

In the .c rule:

```
.c:
        $(LINK.c) $< -o $@
```

$@ is replaced with the name of the current target.

Because values for the < and * macros depend upon both the order of suffixes in the suffixes list, you may get surprising results when you use them in an explicit target entry. See *Suffix Replacement in Macro References* for a strictly deterministic method for deriving a filename from a related filename.

Dynamic Macro Modifiers

Dynamic macros can be modified by including F and D in the reference. If the target being processed is in the form of a pathname, $(@F) indicates the filename part, while $(@D) indicates the directory part. If there are no / characters in the target name, then $(@D) is assigned the dot character (.) as its value. For example, with the target named /tmp/test, $(@D) has the value /tmp; $(@F) has the value test.

Dynamic Macros and the Dependency List: Delayed References

Dynamic macros are assigned while processing any and all targets. They can be used within the target's rule as is, or in the dependency list by prepending an additional $ character to the reference. A reference beginning with $$ is called a *delayed* reference to a macro. For instance, the entry:

```
x.o y.o z.o: $$@.BAK
        cp $@.BAK $@
```

could be used to copy x.o from a backup copy named x.o.BAK, and so forth for y.o and z.o.

The dependency list is read twice, once while building the dependency list for the target, and again while checking each dependency. make resolves macros it encounters in each pass. Before processing any dependencies, the dynamic macros aren't defined. Unless the references are delayed until the second pass, make resolves them to an empty value. $$ is a reference to the $ predefined macro. This macro, conveniently enough, has the value $, and when make resolves it in the first pass, the string $$* is interpreted as $*. In the second pass, $* has been assigned a value, so make uses it's value.

Adding Implicit Rules

Pattern matching rules, which are described in the next section, are much easier to use. The procedure for adding implicit rules is given here for compatibility with previous versions of make.

Although make supplies you with a number of useful implicit rules, you can add new ones of your own if you wish. Historically, implicit rules are also referred to as "suffix" rules, since with the default set of implicit selection is based on the suffixes of the target and dependency.

Now, however, there is an easier and more general method for building one type of file from another with a common basename. Pattern matching rules,[8] which are described in the next section, provide a method for selecting implicit rules based on prefixes, suffixes, or both.[9] This section describes the traditional method of adding implicit rules. Makefiles that use this method will be compatible with earlier versions of make.

Adding an implicit rule is a two-step process. First, you must add the suffixes of both target and dependency file to the suffixes list by providing them as dependencies to the .SUFFIXES special target. Because dependency lists accumulate, you can add suffixes to the list simply by adding another entry for this target, for example:

```
.SUFFIXES: .ms .tr
```

Second, you must add a target entry for the implicit rule:

```
.ms.tr:
        troff -t -ms $< > $@
```

A makefile with these entries can be used to format document source files containing ms macros (.ms files) into troff output files (.tr files):

```
hermes% make doc.tr
troff -t -ms doc.ms > doc.tr
```

Entries in the suffixes list are contained in the SUFFIXES[10] macro. To insert suffixes at the head of the list, first clear its value by supplying an entry for the .SUFFIXES target that has no dependencies (an exception to the rule that dependency lists accumulate):

```
.SUFFIXES:
```

and then add another entry containing the new suffixes, followed by a reference to the SUFFIXES macro, as shown below.

[8] Not available with earlier versions of make.

[9] The implicit rules provided in the default makefile are written as suffix rules, for compatibility with earlier versions of make. They could just as well have been written as pattern matching rules.

[10] Note that there is no leading dot.

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

Pattern Matching Rules: an Alternative to Implicit (Suffix) Rules

A *pattern matching rule* is similar to an implicit rule in function. Pattern matching rules are easier to write, and more powerful, because you can specify a relationship between a target and a dependency based on prefixes and suffixes both. A pattern matching rule is a target entry of the form:

tp%ts: *dp%ds*
    rule

where *tp* and *ts* are the optional prefix and suffix in the target name, respectively, *dp* and *ds* are the (optional) prefix and suffix in the dependency name, and % is a wild card that stands for a basename common to both.

make checks for pattern matching rules before it checks for implicit rules. Although you *can* use them to override the standard set of implicit rules, it is usually a bad idea to do so.

If there is no rule for building a target, make searches for a pattern matching rule, *before* checking for an implicit (suffix) rule. If it can use a pattern matching rule, it does so.

If the target pattern matches the target name, and there is a dependency file matching the dependency pattern, and if the target is out of date with respect to that dependency file, make rebuilds the target. If the target is up to date with respect to the dependency, make does not rebuild it, and continues processing with the next target in the dependency hierarchy.

If the target entry for a pattern matching rule contains no rule, make processes the target file as if it had an explicit target entry with no rule; it searches for a suffix rule, attempts to extract a version of the target file from sccs, finally it treats the target as having a null rule (flagging the target as having been updated which forces a parent target to be rebuilt).

A pattern matching rule for formatting a troff source file into a troff output file looks like:

```
%.tr: %.ms
        troff -t -ms $< > $@
```

As you can see, this is much easier to write, and much simpler to follow than the equivalent suffix rule from the previous section.

make's Standard Implicit Rules and Predefined Macros

The following tables list the standard set of implicit rules and predefined macros supplied with make.

Table 1-1 make's *Standard Implicit Rules*

Use	*Implicit Rule Name*	*Command Line(s)*
Assembly Files	.s.o	$(COMPILE.s) $< -o $@
	.S.o	$(COMPILE.S) $< -o $@
C *Files*	.c	$(LINK.c) $< -o $@
	.c.ln	$(LINT.c) -i $< $(OUTPUT_OPTION)
	.c.o	$(COMPILE.c) $< $(OUTPUT_OPTION)
FORTRAN 77 Files	.f	$(LINK.f) $< -o $@
	.f.o	$(COMPILE.f) $< $(OUTPUT_OPTION)
	.F	$(LINK.F) $< -o $@
	.F.o	$(COMPILE.F) $< $(OUTPUT_OPTION)
lex *Files*	.l	$(RM) $*.c $(LEX.l) $< > $*.c $(LINK.c) $*.c -o $@ $(RM) $*.c
	.l.c	$(RM) $@ $(LEX.l) $< > $@
	.l.ln	$(RM) $*.c $(LEX.l) $< > $*.c $(LINT.c) -i $*.c -o $@ $(RM) $*.c
	.l.o	$(RM) $*.c $(LEX.l) $< > $*.c $(COMPILE.c) $*.c -o $@ $(RM) $*.c
Modula 2 Files	.mod .mod.o .def.sym	$(COMPILE.mod) -e $@ $< -o $@ $(COMPILE.mod) $< -o $@ $(COMPILE.def) $< -o $@
Pascal Files	.p	$(LINK.p) $< -o $@
	.p.o	$(COMPILE.p) $< $(OUTPUT_OPTION)
Ratfor Files	.r	$(LINK.r) $< -o $@
	.r.o	$(COMPILE.r) $< $(OUTPUT_OPTION)
Shell Scripts	.sh	cp $< $@ chmod +x $@
yacc *Files*	.y	$(YACC.y) $< $(LINK.c) y.tab.c -o $@ $(RM) y.tab.c
	.y.c	$(YACC.y) $< mv y.tab.c $@
	.y.ln	$(YACC.y) $< $(LINT.c) -i y.tab.c -o $@ $(RM) y.tab.c

Table 1-1 make's *Standard Implicit Rules— Continued*

Use	*Implicit Rule Name*	*Command Line(s)*
	.y.o	$(YACC.y) $< $(COMPILE.c) y.tab.c -o $@ $(RM) y.tab.c

Table 1-2 make's *Predefined and Dynamic Macros*

Use	*Macro*	*Default Value*
Assembler Commands	AS	as
	ASFLAGS	
	COMPILE.s	$(AS) $(ASFLAGS) $(TARGET_ARCH)
	COMPILE.S	$(CC) $(ASFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
C Compiler Commands	CC	cc
	CFLAGS	
	CPPFLAGS	
	COMPILE.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $(LDFLAGS)
FORTRAN 77 Compiler Commands	FC	f77
	FFLAGS	
	COMPILE.f	$(FC) $(FFLAGS) $(TARGET_ARCH) -c
	LINK.f	$(FC) $(FFLAGS) $(TARGET_ARCH) $(LDFLAGS)
	COMPILE.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $(LDFLAGS)
Link Editor Command	LD	ld
	LDFLAGS	
lex *Command*	LEX	lex
	LFLAGS	
	LEX.l	$(LEX) $(LFLAGS) -t
lint *Command*	LINT	lint
	LINTFLAGS	
	LINT.c	$(LINT) $(LINTFLAGS) $(CPPFLAGS) $(TARGET_ARCH)
Modula 2 Commands	M2C	m2c
	M2FLAGS	
	MODFLAGS	
	DEFFLAGS	
	COMPILE.def	$(M2C) $(M2FLAGS) $(DEFFLAGS) $(TARGET_ARCH)
	COMPILE.mod	$(M2C) $(M2FLAGS) $(MODFLAGS) $(TARGET_ARCH)
Pascal Compiler Commands	PC	pc
	PFLAGS	
	COMPILE.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $(LDFLAGS)
Ratfor Compilation Commands	RFLAGS	
	COMPILE.r	$(FC) $(FFLAGS) $(RFLAGS) $(TARGET_ARCH) -c
	LINK.r	$(FC) $(FFLAGS) $(RFLAGS) $(TARGET_ARCH) $(LDFLAGS)

Table 1-2 make's *Predefined and Dynamic Macros— Continued*

Use	*Macro*	*Default Value*
rm *Command*	RM	rm -f
yacc *Command*	YACC YFLAGS YACC.y	yacc $(YACC) $(YFLAGS)
Suffixes List	SUFFIXES	.o .c .c~ .s .s~ .S .S~ .ln .f .f~ .F .F~ .l .l~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .y .y~ .h .h~ .sh .sh~

1.3. Building Object Libraries

Libraries, Members and Symbols

An object library is a set of object files contained in an ar library archive.[11]

Various languages make use of object libraries to store compiled functions of general utility, such as those in the C library.

ar reads in a set of one or more files to create a library. Each member contains the text of one file, preceded by a header. This header contains information taken from the file's directory entry when the text is read in, including the modification time. make can treat the library member as a separate entity for dependency checking using this header.

When you compile a program that uses functions from an object library (specifying the proper library either by filename, or with the -l option to cc), selects and links with the library member that contains a needed function or symbol.

You can use ranlib to generate a symbol table for a library of object files. ld uses this table for random access to symbols within the library—to locate and link object files in which functions are defined. Alternatively, you can use lorder and tsort to put members in order within the library. (See lorder(1) for details.) For very large libraries, it is a good idea to do both.

Library Members and Dependency Checking

make recognizes a target or dependency of the form

lib.a (*member* ...)

as a reference to a library member, or a space-separated list of members.[12] For example, the following target entry indicates that the library named librpn.a is built from members named stacks.o and fifos.o. The pattern matching rule indicates that each member depends on a corresponding object file, and that object file is built from its corresponding source file using an implicit rule.

---

[11] See ar(1), ar(5), lorder(1), and ranlib(1) in the *Commands Reference Manual* for details about library archive files.

[12] Earlier versions make recognize this notation. However, only the first item in a parenthesized list of members was processed. In this version of make, all members in a parenthesized list are processed.

```
librpn.a: librpn.a(stacks.o fifos.o)
        ar rv $@ $?
        ranlib $@
        rm -f $?

lib.a(%.o): %.o
```

The `rm` command in the target entry removes the object file. Since there is an exact duplicate of that file contained in the library that the link editor can use, the file is not needed. When you update the source files and run `make`, the outdated object files are rebuilt.[13] When used with library-member notation, the dynamic macro `$?` contains the list of files that are newer than their corresponding members:

```
hermes% make
cc   -mc68020 -c stacks.c
cc   -mc68020 -c fifos.c
ar rv librpn.a stacks.o fifos.o
a - stacks.o
a - fifos.o
ranlib librpn.a
rm -f stacks.o fifos.o
```

Library Member Name Length Limit

The name of an `ar` library member cannot exceed 15 characters. If a filename is longer than that, `ar` truncates the name of its corresponding member to the first 15 characters. If a library depends upon a member whose corresponding filename is too long, `make` attempts to match the name of the member to the first 15 characters of a file in the directory. *make* uses the first filename that matches as the file from which to build the member.

.PRECIOUS: Preserving Libraries Against Removal Due to Interrupts

Normally, if you interrupt `make` in the middle of a target, the target file is removed. For individual files this is a good thing, otherwise incomplete files with brand new modification times might be left in the directory. For libraries, which consist of several members, the story is different. It is often better to leave the library intact, even if one of the members is still out of date. This is especially true for large libraries, especially since a subsequent `make` run will pick up where the previous one left off—by processing the object file or member whose processing was interrupted.

`.PRECIOUS` is a special target that is used to indicate which files should be preserved against removal on interrupts; `make` does not remove targets that are listed as its dependencies. If you add the line:

```
.PRECIOUS: librpn.a
```

[13] This is only true for object files in the current working directory. Object files from other directories that correspond to library members aren't necessarily rebuilt.

to the makefile shown above, run `make`, and interrupt the processing of `librpn.a`, the library is preserved.

The `$%` dynamic macro is provided specifically for use with libraries. When a library member is the target, the member name is assigned to the `$%` macro. For instance, this makefile:

```
libx.a(demo.o):
        echo $%
```

produces the results shown in the next example.

```
hermes% make
echo demo.o
demo.o
```

1.4. Maintaining Programs With `make`

In previous sections you have learned how `make` can help compile simple programs and build simple libraries. The focus of this section is on developing makefiles for more complex compilations. When things get complicated it is often a good idea to put each module into a separate directory of its own. This eliminates confusion about which source files pertain to which programs or libraries. This scheme allows you to create makefiles that operate consistently between various parts of a software project. Subsequent sections describe how to maintain a project that spans several directories as a single entity.

Using Macros for Added Flexibility

You have seen how to use predefined and dynamic macros within rules, and for passing parameters from the command line. `make` also allows you to define your own macros within a makefile. Macros allow you to simplify makefiles and make them more flexible (for use in building other modules within the same project, or between projects; makefiles for this version of `make` are not necessarily portable across versions). With judicious use of macros, you can develop *template* makefiles that can be re-used for similar tasks after only minor edits. This section begins to develop template makefiles for C programs and libraries.

Macro references can appear anywhere in a makefile; they can be used to abbreviate long target lists or expressions, or as shorthand to replace long strings that would otherwise have to be repeated. And, macro references can be nested—they can contain other macro references.[14]

Once you have developed a powerful makefile that works the way you like, chances are that you won't want to make extensive edits in order to re-use it for a similar program or library.

For example, the makefile for compiling a C program that used implicit rules can be made to accommodate other programs. By replacing key words with macros, and by editing the definitions of those macros, it becomes a simple task to alter

[14] Nested references are expanded from innermost to outermost. This is not the case with previous versions of `make`.

the basic makefile for use with yet another program:

```
#       Flexible makefile for a C program.

SOURCES= main.c data.c
OBJECTS= main.o data.o
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=

.KEEP_STATE:

$(PROGRAM): $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS)

clean:
        rm $(PROGRAM) $(OBJECTS)
```

In this case, you need only edit the SOURCES, OBJECTS and PROGRAM macros and you can compile a different program entirely, albeit in the same way.

Although in a simple case like this the changes to the makefile might not seem worth the extra trouble, the added flexibility becomes increasingly important as apply more powerful techniques. With judicious use of macros you can avoid having to puzzle over which specific changes you can or should (or even dare) make to a hard-coded makefile.

Makefiles as Specifications

A makefile performs an important function by documenting what object files, programs, or libraries get built from which sources files, and what compilation options are used by default to build them. Specifying this information with a set of macro definitions at the top of a makefile is a great aid the reader, especially when makefiles are similar in format or at all complicated.

No one should have to scan an entire makefile just to puzzle out what it builds.

Suffix Replacement in Macro References

In the flexible makefile shown above, the value of OBJECTS is a bit redundant. It would be better to derive the names of the object files from the names of the source files. In fact, there are any number of filenames that can be derived from the names of source files, simply by altering their suffix. For this reason, make provides a mechanism for temporarily replacing suffixes of words in a macro's value, when the reference to that macro is of the form:[15]

$ (*macro*: *old-suffix*=*new-suffix*)

This *suffix replacement* macro reference allows you to express the list of object files in terms of the list of sources:

```
OBJECTS= $(SOURCES:.c=.o)
```

[15] Although conventional suffixes start with dots, a suffix may consist of any string of characters.

Using lint With make

We encourage you to lint your C programs for easier debugging and maintenance. lint also checks for C constructs that are not considered portable across machine architectures. It can be a real help in writing portable C programs.

lint, the C program verifier,[16] is an important tool for forestalling the kinds of bugs that are most difficult and tedious to track down. These include uninitialized pointers, parameter mismatches in function calls, and nonportable uses of C constructs. As with the clean target, lint is a target name used by convention; it is usually a good practice to include it in makefiles that operate on C programs. lint produces output files that have been preprocessed through cpp and its own first (parsing) pass. These files characteristically end in the .ln suffix, and can also be derived from the list of sources through suffix replacement:

```
LINTFILES= $(SOURCES:.c=.ln)
```

The lint target entry appears as follows:

```
lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)
```

There is an implicit rule for building each .ln file from its corresponding .c file, so there is no need for target entries for the .ln files. As sources change, the .ln files are updated whenever you run

```
make lint
```

Since the LINT.c predefined macro includes a reference to the LINTFLAGS macro, it is a good idea to specify the lint options to use by default (none in this case). Since lint entails the use of cpp, it is a good idea to use CPPFLAGS, rather than CFLAGS for compilation preprocessing options (such as -I).

Also, when you run make clean you will want to get rid of any .ln files produced by this target. It is a simple enough matter to add another such macro reference to the clean target:

```
clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

With these changes, the new version of the makefile appears as follows.

[16] See *Debugging Tools for the Sun Workstation* for more information about lint.

```
#       Makefile for a C program with an entry for lint.

SOURCES= main.c data.c
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

$(PROGRAM): $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS)

lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Figure 1-8 *Makefile With "Suffix-Replacement" Macro References*

Linking With System-Supplied Libraries

This makefile is easily altered to compile a program that uses system-supplied library packages. The next example shows a makefile that compiles a program that uses the `curses` and `termlib` library packages for screen-oriented cursor motion.

You can also link with a library by specifying its pathname name as an argument to `cc`.

A makefile link with user-supplied libraries appears later on.

```
#   Makefile for a C program with curses and termlib.
SOURCES= main.c data.c
LIBS= -lcurses -ltermlib
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

$(PROGRAM): $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS) $(LIBS)

lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Figure 1-9 *Makefile for a* C *Program With System-Supplied Libraries*

Since the link editor resolves undefined symbols as they are encountered, it is normally a good idea to place library references at the end of the list of files to (compile and) link.

This makefile produces:

```
hermes% make
cc -O  -mc68020 -c main.c
cc -O  -mc68020 -c data.c
cc -O  -mc68020 -o program main.o data.o -lcurses -ltermlib
```

Compiling Programs for Debugging and Profiling

Compiling programs for debugging or profiling introduces a new twist to the procedure, and to the makefile. These variants are produced from the same source code, but are built with different options to the C compiler. The `cc` option to produce object code that is suitable for debugging is `-g`, and it is important to omit the `-O` option in this case. The `cc` options that produce code for profiling are `-O` and `-pg`.

Since the compilation procedure is the same otherwise, you *could* give make a definition for `CFLAGS` on the command line. Since this definition overrides the definition in the makefile, and `.KEEP_STATE` assures any command lines affected by the change are performed, the command:

```
make "CFLAGS= -O -pg"
```

produces the following results.

```
hermes% make "CFLAGS= -O -pg"
cc  -O -pg  -mc68020 -c main.c
cc  -O -pg  -mc68020 -c data.c
cc  -O -pg  -mc68020 -o program main.o data.o -lcurses -ltermlib
```

Of course, you may not want to memorize these options or type a complicated command like this, especially when you can put this information in the makefile. What is needed is a way to tell make how to produce a debugging or profiling variant, and some instructions in the makefile that tell it how. One way to do this might be to add two new target entries, one named `debug`, and the other named `profile`, with the proper compiler options hard-coded into the command line.

A better way would be to add these targets, but rather than hard-coding their rules, include instructions to alter the definition of `CFLAGS` depending upon which target it starts with. Then, by making each one depend on the existing target for `program` `make` could simply make use of its rule, along with the specified options.

Instead of saying "**`make "CFLAGS= -g`**", you could say "**`make debug`**" to compile a variant for debugging. The question is, how do you tell `make` that you want a macro defined one way for one target (and its dependencies), and another way for a different target?

Conditional Macro Definitions

A conditional macro definition is a line of the form:

target-name **:=** *macro* **=** *value*

`make` must know which targets the definition applies to, so you can't use a conditional macro definition to alter a target name.

which assigns the given *value* to the indicated *macro* while `make` is processing the target named *target-name* and its dependencies. The following lines give `CFLAGS` an appropriate value for processing each program variant.

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
```

The following makefile produces your choice of optimized, debugging, or profiling variants of a **C** program, depending on which target you specify (the default is the optimized variant). Command dependency checking guarantees that the program and its object files will be recompiled whenever you switch between variants.

```
#
#       Makefile for a C program with alternate
#               debugging and profiling variants.

SOURCES= main.c data.c
LIBS= -lcurses -ltermlib
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

all debug profile: $(PROGRAM)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

$(PROGRAM): $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS) $(LIBS)

lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Figure 1-10 *Makefile for a* C *Program with Alternate Debugging and Profiling Variants*

Going through the makefile, all of the lines above `.KEEP_STATE` seem familiar. The subsequent target entry specifies three targets, with `all` appearing first

`all` is a conventional target for building "all" final, or "finished" targets. Debugging and profiling variants aren't normally considered part of a finished program.

`all` traditionally appears as the first target in makefiles with alternate starting targets (or those that process a list of targets). It's dependencies are "all" targets that go into the final build, whatever that may be. In this case, the final target is the optimized program variant. This entry also indicates that `debug` and `profile` depend on `program` (the value of `$(PROGRAM)`).

The next two lines contain conditional macro definitions for `CFLAGS`, when it appears in `profile` or `debug`, or their dependencies:

```
debug := CFLAGS= -g
profile := CFLAGS= -pg -O
```

Next comes the familiar target entry that starts with `$(PROGRAM)`. Finally, the remainder of the makefile looks familiar.

With this makefile,

```
make
```

or

```
make all
```

produces:

```
hermes% make
cc -O  -mc68020 -c main.c
cc -O  -mc68020 -c data.c
cc -O  -mc68020 -o program main.o data.o -lcurses -ltermlib
```

```
make debug
```

produces:

```
hermes% make debug
cc -g  -mc68020 -c main.c
cc -g  -mc68020 -c data.c
cc -g  -mc68020 -o program main.o data.o -lcurses -ltermlib
```

and

```
make profile
```

produces:

```
hermes% make profile
cc -pg -O   -mc68020 -c main.c
cc -pg -O   -mc68020 -c data.c
cc -pg -O   -mc68020 -o program main.o data.o -lcurses -ltermlib
```

The next example applies similar techniques to maintaining a C object library.

```
#
#   Makefile for a C library with alternate
#       variants.

SOURCES= calc.c map.c draw.c
LIBRARY= libpkg.a

CFLAGS= -O
CPPFLAGS=
LINTFLAGS=

MEMBERS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

all debug profile: $(LIBRARY)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

.KEEP_STATE:
.PRECIOUS: $(LIBRARY)

$(LIBRARY): $(LIBRARY)($(MEMBERS))
        ar rv $@ $?
        ranlib $@
        rm -f $?

$(LIBRARY)(%.o): %.o

lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

clean:
        rm  -f $(LIBRARY) $(MEMBERS) $(LINTFILES)
```

Figure 1-11 *Makefile for a* C *Library with Alternate Variants*

Maintaining a Directory of Header Files

The makefile for maintaining an `include` directory of header files, is really quite simple. Since header files are maintained as plain text, all that is needed is a target, `all` that lists them all as dependencies. Automatic `sccs` extraction takes care of the rest. If you use a macro for the list of header files, this same list can be used in other target entries that may be added later for project management purposes.

```
#       /usr/include/make/h.mk
#
#       Makefile for maintaining an include directory.

FILES.h= calc.h map.h draw.h

all: $(FILES.h)

clean:
        rm -f $(FILES.h)
```

Compiling and Linking With Your Own Libraries

When preparing your own library packages, it often makes sense to treat each library as a separate entity from programs that use it, as well as the header files used by both. Separating programs, libraries and header files into distinct directories often makes it easier to prepare makefiles for each type of module. And, it clarifies the structure of a software project.

It is not a good idea to have things pop up all over the file system as a result of running make.

A courteous and necessary convention of makefiles is that they only build files in the working directory, or in temporary subdirectories. Unless you are using make specifically to install files into a specific directory on an agreed-upon file system, it is regarded as very poor form for a makefile to produce output in another directory.

Building programs that rely on user-supplied libraries in other directories adds several new wrinkles to the makefile. Up until now, everything needed has been in the directory, or else in one of the standard directories that are presumed to be stable. This is not true for user-supplied libraries that are part of a project under development, especially when their contents are subject to change.

More importantly, since these libraries aren't built automatically (there is no equivalent to automatic sccs extraction for them), there must be an explicit target entry to build them. So, a problem arises (until such time as the library has been completed tested and can be presumed to be stable).

On the one hand, you need to assure the libraries you link with are up to date. On the other hand, you need to observe the convention that a makefile should only maintain files in the local directory. In addition, the makefile should not contain duplicate information that could get out of sync with a makefile in another directory. The whole purpose of make, after all, is to provide consistent, modular processing.

Nested make Commands

The solution is to use a nested make command, running in the directory the library resides in, to rebuild it (according to the target entry in the makefile there).

The MAKE macro, which is set to the value "make" in the default file, overrides the -n option. Any command line in which it is referred to is executed, even though -n may be in effect. Since this macro is used to invoke make, and since the make it invokes inherits -n from the special MAKEFLAGS macro, make can trace a hierarchy of nested make commands with the -n option.

```
#       First cut entry for target in another
#       directory.

../lib/libpkg.a:
        cd ../lib ; $(MAKE) libpkg.a
```

The library is specified with a pathname relative to the current directory. In general, it is better to use relative pathnames. If the project is moved to a new root directory or machine, so long as its structure remains the same relative to that new root directory, all the target entries will still point to the proper files.

Within the nested make command line, the dynamic macro modifiers F and D come in handy, as does the MAKE predefined macro. If the target being processed is in the form of a pathname, $(@F) indicates the filename part, while $(@D) indicates the directory part. If there are no / characters in the target name, then $(@D) is assigned the dot character (.) as its value.

The target entry can be rewritten as:

```
#       Second cut.

../lib/libpkg.a:
        cd $(@D); $(MAKE) $(@F)
```

Forcing A Nested make Command to Run

Because it has no dependencies, this target will only run when the file named `../lib/libpkg.a` is missing. If the file is a library archive protected by `.PRECIOUS`, this could be a rare occurrence. The current `make` invocation neither knows nor cares about what that file depends on, nor should it. It is the nested invocation that decides whether and how to rebuild that file. After all, just because a file is present in the file system doesn't mean that it is up to date. This means that you have to force the nested `make` to run, regardless of the file's presence, by making it depend on a target with a null rule:

```
#       Reliable target entry for a nested make
#       command.

../lib/libpkg.a: FRC
        cd $(@D); $(MAKE) $(@F)
FRC:
```

In this way, `make` reliably `cd`'s to the directory `../lib` and builds `libpkg.a` if necessary, using the entry in the makefile found in that directory (`../lib`):

```
hermes% make ../lib/libpkg.a
cd ../lib; make libpkg.a
make libpkg.a
'libpkg.a' is up to date.
```

These lines are produced by the nested `make` run.

The following makefile uses a nested `make` command to process local libraries that a program depends on.

```
#
#       Makefile for a C program with user-supplied
#       libraries and nested make commands.

SOURCES= main.c data.c
ULIBS= ../lib/libpkg.a
SLIBS= -lcurses -ltermlib
PROGRAM= program

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

all debug profile: $(PROGRAM)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

$(PROGRAM): $(OBJECTS) $(ULIBS)
        $(LINK.c) -o $@ $(OBJECTS) $(ULIBS) $(SLIBS)

$(ULIBS): FRC
        cd $(@D); $(MAKE) $(@F)
FRC:

lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Figure 1-12 *Makefile for* C *Program With User-Supplied Libraries*

When `../lib/libpkg.a` is up to date, this makefile produces:

```
hermes% make
cc -O  -mc68020 -c main.c
cc -O  -mc68020 -c data.c
cd ../lib; make libpkg.a
'libpkg.a' is up to date.
cc -O  -mc68020 -o program main.o data.o ../lib/libpkg.a -lcurses -l termlib
```

The MAKEFLAGS Macro

Do not define MAKEFLAGS in your makefiles.

Like the MAKE macro, MAKEFLAGS is also a special case. As its name suggests, it contains options for make. Its value is composed of the MAKEFLAGS environment variable, if set, in combination with the command-line options that make is invoked with (except for -f). This combination of options takes effect for the current make run.

The value of MAKEFLAGS is always exported, whether set in the environment or not, and the options it contains are passed to any nested make commands (whether invoked by $(MAKE), make or /bin/make). This insures you that nested make commands are always passed the options that the parent make was invoked with. Because MAKEFLAGS is maintained automatically, defining it in the makefile would only be misleading; such a definition has no effect whatsoever on the value exported by MAKEFLAGS.

Macro Definitions and Environment Variables: Passing Parameters to Nested make Commands

With the exception of MAKEFLAGS,[17] make imports variables from the environment and treats them as if they were defined macros. In turn, make propagates those environment variables and their values to commands it invokes, including nested make commands. Macros can also be defined as command line arguments, or a macro can be defined in the default file. This can lead to conflicts when a macro is defined in more than one place. make has precedence rules for evaluating macros with conflicting definitions.

First of all, conditional macro definitions always take effect within the targets (and their dependencies) for which they are defined.

If make is invoked with a macro-definition argument, that definition takes precedence over definitions given either within the makefile, or imported from the environment. (This does not necessarily hold true for nested make commands, however.) Otherwise, if you define (or redefine) a macro within the makefile, the most recent definition applies. The latest definition normally overrides the environment. Lastly, if the macro is defined in the default file and nowhere else, that value is used.

The -e option alters this scheme. With -e, macros defined in the environment override any and all makefile definitions (but not the command line).

With nested make commands, definitions made in the makefile normally override the environment, but only for the makefile in which each definition occurs; the value of the corresponding environment variable is propagated regardless. Command-line definitions override both environment and makefile definitions, but only for the topmost make run. Although values from the command line are propagated to nested make commands, they are overridden both by definitions in the nested makefiles, and by environment variables imported by the nested make commands.

The -e option behaves more consistently. The environment overrides macro definitions made in any makefile, and command-line definitions are always used ahead of definitions in the makefile and the environment. One drawback to -e is that it introduces a situation in which information that is *not contained in the makefile* can be critical to the success or failure of a build.

This is an awful lot to remember, so a good rule of thumb when passing parameters to nested make commands is: supply them as command-line definitions, and use -e. However, before you run make with the -e option, it is important to

[17] and SHELL. The SHELL environment variable is neither imported nor exported in this version of make. See *Commands Reference Manual*, the make reference manual page, Appendix B, for more information about the SHELL macro.

eliminate all extraneous or improperly defined environment variables, since `make -e` will propagate whatever is in the environment to the entire hierarchy of nested `make` commands:

```
make -e CFLAGS=-E
```

Environment variables don't go away when you're done with them (i.e, they stay around to haunt you, especially when you attempt to build something else with `make` later on). One way to avoid lingering environment variables is to invoke `make` within a subshell. When you set environment variables and run `make` in the subshell, their values are isolated within that subshell and any processes it spawns (like the one for `make`):

```
( setenv CFLAGS -E ; make -e )
```

This next example illustrates the difference in parameters between the top `make` run and the nested `make` runs, using the two makefiles shown below.

```
# top.mk

MACRO= "Correct if unexpected."

top:
        @echo "------------------------------ top"
        echo $(MACRO)
        @echo "------------------------------"
        $(MAKE) -f nested.mk

        @echo "------------------------------ clean"
clean:
        rm nested
```

and:

```
# nested.mk

MACRO=nested

nested:
        @echo "------------------------------ nested"
        touch nested
        echo $(MACRO)
        $(MAKE) -f top.mk
        $(MAKE) -f top.mk clean
```

With these makefiles, the command:

```
make -f top.mk MACRO=top
```

produces the results that follow.

```
hermes% make -f top.mk MACRO=top
---------------------------- top
echo top
top
----------------------------
make -f nested.mk
---------------------------- nested
touch nested
echo nested
nested
make -f top.mk
---------------------------- top
echo "Correct, if unexpected."
Correct, if unexpected.
----------------------------
make -f nested.mk
'nested' is up to date.
make -f top.mk clean
---------------------------- clean
rm nested
```

This pair of makefiles can be helpful if you decide to review the various cases yourself.

Table 1-3 *Summary of Macro Assignment Order*

Without -e	*With* -e *in effect*
top-level make *command:*	
conditional definitions	conditional definitions
make command line	make command line
latest makefile definition	environment value
environment value	latest makefile definition
predefined value, if any	predefined value, if any
nested make *commands:*	
conditional definitions	conditional definitions
make command line	make command line
latest makefile definition	parent make cmd. line
environment variable	environment value
predefined value, if any	latest makefile definition
parent make cmd. line	predefined value, if any

Compiling Other Source Files

The following examples illustrate the use of make to maintain C programs that contain assembly routines, and programs produced with lex and yacc.

Compiling and Linking a C Program with Assembly Language Routines

The makefile in the next example maintains a program with C source files linked with assembly language routines.[18] There are two varieties of assembly source files, those that contain cpp preprocessor directives, and those that don't. By convention, assembly source files without preprocessor directives have the .s suffix. Assembly sources that require preprocessing have the .S suffix.

Assembly sources are assembled to form object files in a fashion similar to that used to compile C sources. The object files can then be linked into a C program. make has implicit rules for transforming .s and .S files into object files, so at a minimum, a target entry for a C program with assembly routines need only specify how to link the objects files. You can use the familiar cc command to link object files produced by the assembler:

ASFLAGS passes options for as to the .s.o and .S.o implicit rules.

```
CFLAGS= -O
ASFLAGS= -O

.KEEP_STATE:

driver:  c_driver.o s_routines.o S_routines.o
        cc -o driver c_driver.o s_routines.o S_routines.o
```

The next example shows a more flexible makefile for this sort of compilation.

[18] Refer to the *Assembly Reference Manual* for more information about assembly language source files.

```
#       Makefile for a C program linked with assembly routines.

SOURCES.c= c_driver.c
SOURCES.s= s_routines.s
SOURCES.S= S_routines.S
ULIBS=
SLIBS=
PROGRAM= driver

ASFLAGS=
CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=
OBJECTS= $(SOURCES.c:.c=.o) $(SOURCES.s:.s=.o) $(SOURCES.S:.S=.o)
LINTFILES= $(SOURCES.c:.c=.ln)  # not for assembly sources

.KEEP_STATE:

all debug profile: $(PROGRAM)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

$(PROGRAM): $(OBJECTS) $(ULIBS)
        $(LINK.c) -o $@ $(OBJECTS) $(ULIBS) $(SLIBS)

$(ULIBS): FRC
        cd $(@D); $(MAKE) $(@F)
FRC:

lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Figure 1-13 *Makefile for a C Program with Assembly Routines*

This makefile compiles the executable program `driver` as shown:

```
hermes% make
cc -O  -mc68020 -c c_driver.c
as  -mc68020 -o s_routines.o s_routines.s
cc   -mc68020 -o S_routines.o -c S_routines.S
cc -O  -mc68020 -o driver c_driver.o s_routines.o S_routines.o
```

Note that the `.S` files are processed using the `cc` command, which invokes the C preprocessor `cpp`, and invokes the assembler implicitly.

Compiling lex and yacc Sources

lex and yacc produce C source files as output. Source files for lex end in the suffix .l, while those for yacc end in .y. When used separately, the compilation process for each is similar to that used to produce programs from C sources alone. There are implicit rules for compiling the lex or yacc sources into .c files; from there the files are further processed with the implicit rules for compiling object files from C sources. Typically, however, there is no need to keep the c file, which in this simple case serves as an intermediate file, and so it is typical when compiling a lex or yacc file to use either the .l.o rule, or the .y.o rule, respectively, to produce the object files and remove the .c files. For example, the makefile:

```
CFLAGS= -O
.KEEP_STATE:

all: l_grammar y_compiler

l_grammar: l_grammar.o

y_compiler: y_compiler.o
```

produces:

```
hermes% make
rm -f l_grammar.c
lex  -t l_grammar.l > l_grammar.c
cc -O  -mc68020 -c l_grammar.c -o l_grammar.o
rm -f l_grammar.c
rm -f l_grammar.c
lex  -t l_grammar.l > l_grammar.c
cc -O  -mc68020 l_grammar.c -o l_grammar
rm -f l_grammar.c
yacc  y_compiler.y
cc -O  -mc68020 -c y.tab.c -o y_compiler.o
rm -f y.tab.c
yacc  y_compiler.y
cc -O  -mc68020 y.tab.c -o y_compiler
rm -f y.tab.c
```

Things get to be a bit more complicated when you use lex and yacc in combination. In order for the object files to work together properly, the C code from lex must include a header file produced by yacc. So, it may be necessary to recompile the C source file produced by lex when the yacc source file changes. In this case, it is better to retain the .c (intermediate) files produces by lex, as well as the additional .h file that yacc provides, so as to avoid running lex whenever the yacc source changes.

The following makefile maintains a program built from a lex source, a yacc source, and a C source file.

yacc produces output files named y.tab.c and y.tab.h. If you want the output files to have the same basename as the source file, you must rename them.

```
CFLAGS= -O
.KEEP_STATE:

a2z: c_functions.o l_grammar.o y_compiler.o
        cc -o $@ c_functions.o l_grammar.o y_compiler.o

l_grammar.c:

y_compiler.c + y_compiler.h: y_compiler.y
        yacc -d y_compiler.y
        mv y.tab.c y_compiler.c
        mv y.tab.h y_compiler.h
```

Since there is no transitive closure for implicit rules, you must supply a target entry for `l_grammar.c`. This entry bridges the gap between the `.l.c` implicit rule and the `.c.o` implicit rule, so that the dependency list for `l_grammar.o` extends to `l_grammar.l`. Since there is no rule in the target entry, `l_grammar.c` is built using the `.l.c` implicit rule.

The next target entry describes how to produce the `yacc` intermediate files. Because there is no implicit rule for producing both the header file and the C source file using `yacc -d`, a target entry must be supplied that includes a rule for doing so.

Specifying Target Groups With the + Sign

In the target entry for `y_compiler.c` and `y_compiler.h`, the + sign separating the target names indicates that the entry is for a *target group*.[19] A target group is a set of files, all of which are produced when the rule is performed. Taken as a group, the set of files is what comprises the target. Without the + sign, each item listed would comprise a separate target. With a target group, `make` checks the modification dates separately against each target file, but performs the target's rule only once, if necessary, per `make` run.

The next example shows a makefile for the more general case of a `lex` source, a `yacc` source, and any number of C source files.

[19] Not available with earlier versions of `make`.

```
#       Makefile to a compile C program with lex and yacc sources.

SOURCES.c= c_functions.c
LEXFILE.l= l_grammar.l
YACCFILE.y= y_compiler.y
ULIBS=
SLIBS=
PROGRAM= a2z

LFLAGS=
YFLAGS=
CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

LEXFILE.c= $(LEXFILE.l:.l=.c)
YACCFILE.c= $(YACCFILE.y:.y=.c)
YACCFILE.h= $(YACCFILE.y:.y=.h)
SOURCES= $(SOURCES.c) $(LEXFILE.c) $(YACCFILE.c)
OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)

.KEEP_STATE:

all debug profile: $(PROGRAM)

debug := CFLAGS= -g
profile := CFLAGS= -pg -O

$(PROGRAM): $(OBJECTS) $(ULIBS)
        $(LINK.c) -o $@ $(OBJECTS) $(ULIBS) $(SLIBS)

$(LEXFILE.c): $(YACCFILE.h)

$(YACCFILE.c) + $(YACCFILE.h): $(YACCFILE.y)
        $(YACC.y) -d $(YACCFILE.y)
        mv y.tab.c $(YACCFILE.c)
        mv y.tab.h $(YACCFILE.h)

$(ULIBS): FRC
        cd $(@D); $(MAKE) $(@F)
FRC:

lint: $(LINTFILES)
        $(LINT.c) $(LINTFILES)

clean:
        rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES)
```

Figure 1-14 *Makefile for Compiling C Programs With* `lex` *and* `yacc` *Sources*

```
hermes% make all
cc -O  -mc68020 -c c_functions.c
yacc  -d y_compiler.y
mv y.tab.c y_compiler.c
mv y.tab.h y_compiler.h
rm -f l_grammar.c
lex  -t l_grammar.l > l_grammar.c
cc -O  -mc68020 -c l_grammar.c
cc -O  -mc68020 -c y_compiler.c
cc -O  -mc68020   -o a2z c_functions.o l_grammar.o y_compiler.o
hermes%
```

Maintaining Shell Scripts with `make` and `sccs`

Although a shell script is a plain text file, it must be executable in order to run. Since `sccs` removes execute permission for files under its control and a shell script must have execute permission in order to run, a distinction must be drawn between a shell script and it's "source" file under `sccs` control. `make` has an implicit rule for deriving a script from its "source" file under `sccs`. The suffix for a shell script source file is `.sh`. Even though the contents of the script and the `.sh` file are the same, the script has execute permissions, while the `.sh` file does not. `make`'s implicit rule for scripts "derives" the script from its source file, making a copy of the `.sh` file (extracting it first, if necessary) and changing the mode of the resulting script file to allow execution. For example:

```
hermes% file script.sh
script.sh:      ascii text
hermes% make script
cp script.sh script
chmod +x script
hermes% file script
script:         commands text
```

Running Tests with `make`

Shell scripts often come in handy for running tests, and performing other routine tasks that are either interactive, or don't require `make`'s dependency checking. Test suites, in particular, often entail providing a program with specific, repeatable input that a program might expect to receive from a terminal.

In the case of a library, a set of programs that exercise its various functions may be written in C, and then executed in a specific order, with specific inputs from a script. In the case of a utility program, there may be a set of benchmark programs that exercise and time its functions. In each of these cases, the commands to run each test can be incorporated into a shell script repeatability and easy maintenance.

Once you have developed a test script that suits your needs, including a target to run it is easy. Although `make`'s dependency checking may not be needed within the script itself, you *can* use it to make sure that the program or library is updated before running those tests.

In the following target entry for running tests, `test` depends on the library named as a dependency to `all`. If the library is out of date, `make` rebuilds it and proceeds with the test. This insures that you always test with an up to date version:

```
test: all testscript
        set -x ; testscript > /tmp/test.$$$$

testscript: testscript.sh test_1 test_2 test_3

test_1 test_2 test_3: $$@.c $(LIBRARY)
        $(LINK.c) -o $@ $< $(LIBRARY) $(SLIBS)
```

`test` also depends on `testscript`, which in turn depends on the three test programs. This assures that they too are up to date before `make` initiates the test procedure. `all` is built according to its target entry in the makefile; `testscript` is built using the `.sh` implicit rule; and the test programs are built using the rule in the last target entry, assuming that there is just one source file for each test program. (The `.c` implicit rule doesn't apply to these programs, because they must link with the proper libraries in addition to their respective `.c` files).

Delayed References to a Shell Variable

The string `$$$$`, in the rule for `test` is, in fact, a pair of references to `make`'s `$` macro (each written as `$$`). `make` resolves each such reference into a single `$`, and the command line is passed to the shell as:

```
set -x ; testscript > /usr/tmp/test.$$
```

In this way, the variable reference is delayed from final expansion until it reaches the shell, which interprets it as a reference to `$$`, the value of which is the process number of the shell. This number is appended to the output filename so that the results of each successive test is written to a unique filename with a standard format. The `set -x` command forces the shell to display the command on the terminal. This allows you to see the actual filename containing the test results.

This makefile produces:

```
hermes make
cp testscript.sh testscript
chmod +x testscript
cc   -mc68020 -o test_1 test_1.c
cc   -mc68020 -o test_2 test_2.c
cc   -mc68020 -o test_3 test_3.c
testscript > /tmp/test.$$
+ testscript > /tmp/test.26500
```

A more flexible set of entries for testing a library looks like:

```
TESTSCRIPT= testscript
TESTPROGS= test_1 test_2 test_3

test: all $(TESTSCRIPT)
        set -x ; $(TESTSCRIPT) > /tmp/test.$$$$

$(TESTSCRIPT): $(TESTSCRIPT).sh $(TESTPROGS)

$(TESTPROGS): $$@.c $(LIBRARY)
        $(LINK.c) -o $@ $< $(LIBRARY) $(SLIBS)
```

In the case of a program, testing routines written in C may not be necessary; leaving TESTPROGS undefined will mean the target entry for test programs is omitted from the dependency scan. TESTSCRIPT depends only upon its corresponding .sh file. If there *are* test programs that don't depend on a library (the LIBRARY macro is undefined) this method is still applicable; it is the equivalent of the .c implicit rule. If, there is a test program that depends on the same libraries as the program does, you can either replace references to the LIBRARY macro with references to ULIBS:

```
$(TESTPROGS): $$@.c $(ULIBS)
        $(LINK.c) -o $@ $< $(ULIBS) $(SLIBS)
```

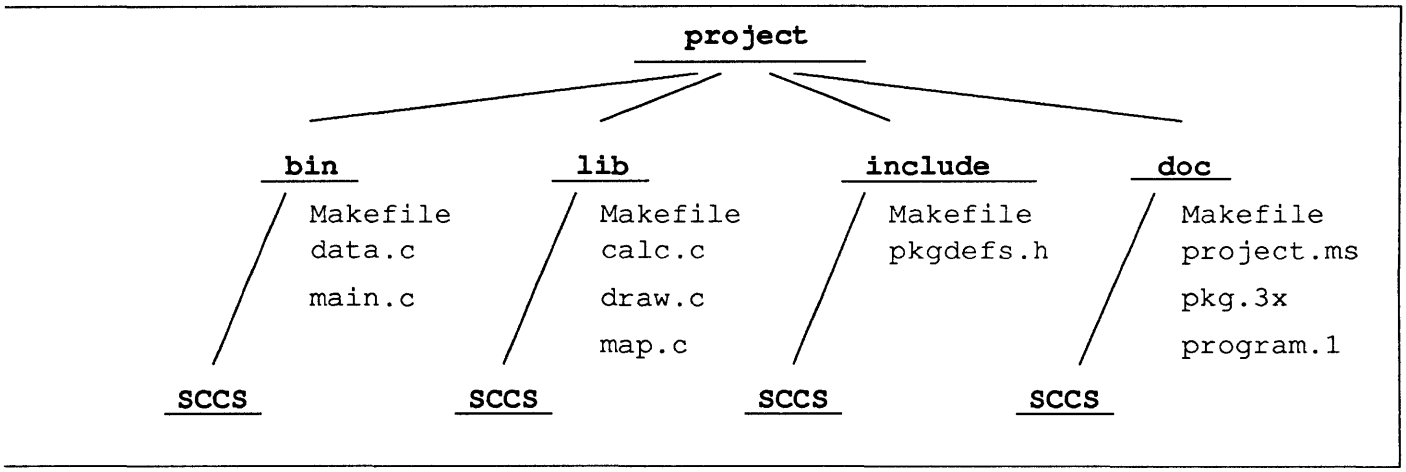
1.5. Maintaining Software Projects

make is especially useful when a software project consists of a system of programs and libraries. By taking advantage of nested make commands, you can use it to maintain object files, executables, and libraries in a whole hierarchy of directories. You can use make in conjunction with sccs, to assure that sources are maintained in a controlled manner, and that programs built from them are consistent. This means that you can provide other programmers with duplicates of the directory hierarchy for simultaneous development and testing if you wish (although there are tradeoffs to consider).

You can use make to build the entire project and install final copies of various modules onto another filesystem for integration and distribution.

Organizing A Project for Ease of Maintenance

As mentioned earlier, one good way to organize a project is to segregate each major piece into its own directory. A project broken out this way usually resides within a single file-system or directory hierarchy. Header files could reside in one subdirectory, libraries in another, and programs in still another. Documentation, such as Reference Pages, may also be kept on hand in another subdirectory. Suppose that a project is composed of one executable program, one library that you supply, a set of header files for the library routines, and some documentation, structured as shown.



The makefiles in each subdirectory can be borrowed from examples in earlier sections, but something more is needed to manage the project as a whole. A carefully structured makefile in the root directory, the *root makefile* for the project, provides target entries for managing the project as a single entity.

As a project grows, the need for consistent, easy-to-use makefiles also grows. Macros and target names should have the same meanings no matter which makefile you are reading. Conditional macro definitions and compilation options for output variants should be consistent across the entire project.

Where feasible, a *template* approach to writing makefiles makes sense. This makes it easy for you keep track of how the project gets built. All you have to do to add a new type of module is to make a new directory for it, copy an appropriate makefile into that directory, and make a few minor edits to change macro values. (Of course, you also need to add the new module to the list of things to build in the root makefile, but that comes later.)

Although a makefile should document exactly what it builds, it does not necessarily have to contain an explanation of every step. After all, the idea is to spend time working on the code, not the makefiles.

Conventions for macro names, such as those for the various source files in the above examples, should be instituted and observed throughout the project. Mnemonic macro names mean that although you may not remember the exact value of the macro, you'll know the type of value it represents (and that's usually more valuable when deciphering a makefile anyway).

Using `include` Makefiles

One method of simplifying makefiles, while providing a consistent compilation environment, is to use `make`'s

`include` *filename*

directive to read in the contents of a named makefile. For instance, there is no need to duplicate the pattern-matching rule for processing `troff` sources in each makefile, when you can include it's target entry. When reading a makefile that contains the `include` directive:

```
SOURCES= main.c data.c
...
clean: $(PROGRAM) $(OBJECTS) $(LINTFILES)
include ../pm.rules.mk
```

make reads in the contents of ../pm.rules.mk, shown here:

```
# pm.rules.mk
#
#       Simple "include" makefile for pattern matching
#       rules.

%.tr: %.ms
        troff -t -ms $< > $@
%.nr: %.ms
        nroff -ms $< > $@
```

While it may seem silly to propagate something like this, keeping the document source for a module's specification in the same directory as the source code makes it easy to find. Having an implicit rule to format the document whenever it is updated makes updating it a whole lot easier.

Although you might not want to rebuild the specification along with the module, and probably don't want its pattern matching rules cluttering up the makefile, ancillary little tidbits like this are handy when you want them.

Installing Finished Programs and Libraries

When a program is ready to be released for outside testing or general use, you can use make to install it. Adding a new target and new macro definition to do so is easy:

```
DESTDIR= /proto/project/bin

install: $(PROGRAM)
        -mkdir $(DESTDIR)
        cp $(PROGRAM) $(DESTDIR)
```

A similar target entry can be used for installing a library under the macro naming scheme used in this manual:

```
DESTDIR= /proto/project/lib

install: $(LIBRARY)
        -mkdir $(DESTDIR)
        cp $(LIBRARY) $(DESTDIR)
```

A list of header files might appear as:

```
DESTDIR= /proto/project/include

install: $(LIST)
        -mkdir $(DESTDIR)
        cp $(LIST) $(DESTDIR)
```

Finally, a list of Reference Manual Pages, which are typically distributed in source form, are installed just like header files (these may comprise a subset of the items in the doc subdirectory).

Building the Entire Project

From time to time it is necessary to take a snapshot of the sources, and the object files that they produce. This can either be done as a checkpoint in the development process, or as an intermediate or final build for release to users. Building an entire project is simply a matter of invoking make successively in each subdirectory to build and install each module.

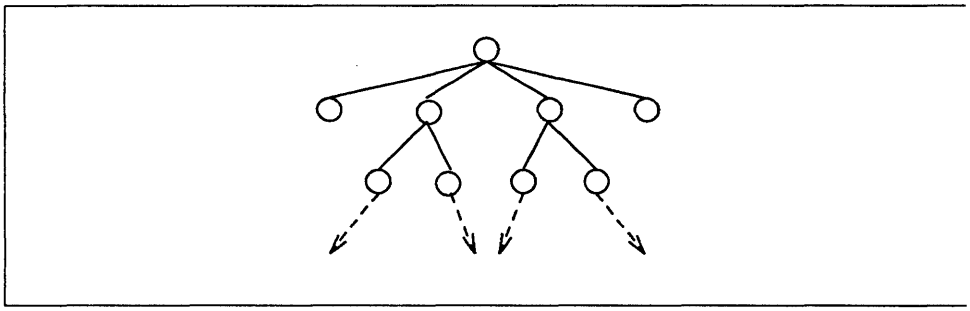
Subsequent examples show how to incorporate these make commands in the root makefile. The root makefile should also allow you to build debugging and profiling variants of the project, clean the directories, and install completed modules. The following simple example uses the shell for construct to loop through the list of subdirectories and invoke a nested make command in each.

```
#       Simple makefile to maintain a project.

DIRS= bin include lib doc

all debug profile lint clean test install: FRC
        for i in $(DIRS) ; do \
                cd $$i ; $(MAKE) $@; \
        done
FRC:
```

The delayed reference, $$i, is used to pass a reference to the index variable for the loop, $i, to the shell. It is important to note that the shell ignores the return status of commands running in the loop. If a make command in one directory fails for some reason, processing continues with the next iteration and the next value for i.

Maintaining Directory Hierarchies With Recursive Makefiles

If you extend your project hierarchy to include more layers:



chances are that not only will the makefile in each intermediate directory have to produce target files, but it will also have to invoke nested `make` commands for its subdirectories. Files in the current directory can sometimes depend on files in subdirectories. Their target entries need to depend on their nested counterparts in the subdirectories.

This means that the nested `make` command for each such target should run before the command in the local directory does. One way to assure that the commands run in the proper order is to make a separate entry for the nested part, and another for the local part. If you add these new targets to the dependency list for the original target, its action will encompass them both.

Targets that encompass equivalent actions in both the local directory and in subdirectories are referred to as *recursive* targets.[20] A makefile with recursive targets is referred to as a *recursive* makefile.

In the case of `all`, the the nested dependency can be named `all.nested`; the local dependency, `all.local`:

```
...
all: all.nested all.local

all.nested:
        for i in $(DIRS) ; do \
                cd $$i ; $(MAKE) all ; \
        done

all.local: $(PROGRAM)
...
```

Note that the nested target invokes `make` with the argument `all`, *not* `all.nested`. The nested `make` must also be recursive, unless it is at the bottom of the hierarchy. Either way, it should be invoked with the same target name as its parent target. In the makefile for a leaf directory (one with no subdirectories to descend into), you can simply give a null definition to the `DIRS` macro. This will halt any further descent by `all.nested`. When the shell begins a

[20] Strictly speaking, any target that calls `make`, with its name as an argument, is recursive. However, here the term is reserved for the narrower case of targets that have both nested and local actions. Targets that only have nested actions are referred to as "nested" targets.

`for` loop with an empty list for the index variable, it performs zero iterations of the loop; the loop terminates without having issued any nested `make` commands; the shell exits gracefully. You can also use one of the earlier makefiles that omits the recursive functions, however, if you add yet another layer of subdirectories later on, you may have to switch makefiles at that time.

Recursive `install` Targets

This same principle can be extended to all of the generic targets. The `install` target, however, is something of a special case. If the destination is a parallel directory hierarchy (such as when you are installing completed source code), the parent directories must be created before the destination subdirectories can be. This often means that the `make install` target in the current directory (which creates the destination directory if needed) must be performed before that in any subdirectory can succeed. So, `install.local` must appear ahead of `install.nested` in the dependency list for `install`.[21]

This next example shows a recursive makefile in a directory with a C program and subdirectories.

[21] If the local target depends on files within a subdirectory, this may force `make` to descend into that subdirectory twice during a `make install` run.

```
#       Recursive makefile for a C program and subdirectories.
#       Also includes test and install targets.

SOURCES= main.c data.c
ULIBS= ../lib/libpkg.a
SLIBS= -lcurses -ltermlib
PROGRAM= program

DIRS= sun2 sun3
TESTSCRIPT= testscript
TESTPROGS= test_1 test_2 test_3
DESTDIR= /proto/project/bin

CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=

OBJECTS= $(SOURCES:.c=.o)
LINTFILES= $(SOURCES:.c=.ln)
TARGETS.nested= all.nested debug.nested profile.nested \
        lint.nested clean.nested test.nested install.nested

.KEEP_STATE:

debug := CFLAGS= -g
profile := CFLAGS= -pg -O
debug.local := CFLAGS= -g       # as in: "make debug.local"
profile.local := CFLAGS= -pg -O #       "make profile.local"

#       Recursive targets:

all debug profile lint clean test: $$@.nested $$@.local
install: $$@.local $$@.nested

#       Nested targets:

$(TARGETS.nested):
        for i in $(DIRS) ; do \
                cd $$i ; $(MAKE) $(@:.nested=) ; \
        done

#       Local target entries:

all.local debug.local profile.local: $(PROGRAM)

$(PROGRAM): $(OBJECTS) $(ULIBS)
    $(LINK.c) -o $@ $(OBJECTS) $(ULIBS) $(SLIBS)

$(ULIBS): FRC
    cd $(@D); $(MAKE) $(@F)
FRC:

lint.local: $(LINTFILES)
    $(LINT.c) $(LINTFILES)

clean.local:
    rm -f $(PROGRAM) $(OBJECTS) $(LINTFILES) $(TESTSCRIPT) $(TESTPROGS)
```

```
test.local: all $(TESTSCRIPT)
        set -x ; $(TESTSCRIPT) > /tmp/test.$$$$
$(TESTSCRIPT): $(TESTSCRIPT).sh $(TESTPROGS)
$(TESTPROGS): $$@.c $(ULIBS)
        $(LINK.c) -o $@ $< $(ULIBS) $(SLIBS)

install.local: $(PROGRAM)
        -mkdir $(DESTDIR)
        -cp $(PROGRAM) $(DESTDIR)
```

Figure 1-15 *Recursive Makefile for Building a* C *Program and Subdirectories*

Notice that you can still use `make` to build a local target, simply by appending the `.local` suffix to the target name that you're used to. The command "`make all.local`" does exactly what you'd expect. However, we recommend against making a habit of this practice, especially where local targets rely on modules in nested targets. If the files in the subdirectories are up to date, it doesn't take very long for `make` to check them. If they *aren't* up to date, and you've built the local target, there is a strong possibility that the local target file will be inconsistent with those lower-level files, at least until it is `clean`'ed and remade.

Maintaining A Large Library as a Hierarchy of Subsidiaries

When maintaining a very large library, it is sometimes easier to break it up into smaller, subsidiary libraries, and use `make` to combine them into a complete package. Although you cannot combine libraries directly with `ar`, you can extract the member files from each subsidiary library, and then archive those files in another step:

```
hermes% ar xv libx.a
x - x1.o
x - x2.o
x - x3.o
hermes% ar xv liby.a
x - y1.o
x - y2.o
hermes% ar rv libz.a *.o
 a - x1.o
 a - x2.o
 a - x3.o
 a - y1.o
 a - y2.o
 ar: creating libz.a
hermes% rm *.o
```

A subsidiary library is maintained using a makefile in its own directory, along with the (object) files it is built from. The makefile for the complete library typically makes a symbolic link to each subsidiary archive, extracts their contents into a temporary subdirectory, and archives the resulting files to form the complete package.

In general, use of shell filename wildcards is considered to be bad form in a makefile. If you *do* use it, you need to take steps to insure that it excludes spurious files, perhaps by isolating affected files in a temporary subdirectory.

The next example updates the subsidiary libraries, creates a temporary directory in which to extracted the files, and extracts them. It uses the * (shell) wild card within that temporary directory to generate the collated list of files. While filename substitutions are generally frowned upon, this use of the wild card is acceptable because the directory is created afresh whenever the target is built. This guarantees that it will contain only files extracted during the *current* `make` run.

The example relies on a naming convention for directories. The name of the directory is taken from the basename of the library it contains. For instance, if `libx.a` is a subsidiary library, the directory that contains it is named `libx`. It makes use of suffix replacements in dynamic-macro references to derive the directory name for each specific subdirectory. (You can verify yourself that this is necessary.)

It uses a shell command substitution to collate the object files into proper sequence for linking (using `lorder` and `tsort`) as it archives them into the package. Finally, it removes the temporary directory and its contents.

```
#       Simple makefile for collating a library from
#       subsidiaries.

LIBRARY= libz.a
LIBS= libx.a liby.a

ARFLAGS=
CFLAGS= -O
CPPFLAGS=

.KEEP_STATE:
.PRECIOUS: libz.a

all: $(LIBRARY)

$(LIBRARY): $(LIBS)
    -rm -rf tmp
    -mkdir tmp
    set -x ; for i in $(LIBS) ; do \
        ( cd tmp ; ar x ../$$i ) ; \
    done
    ( cd tmp ; rm -f _ _.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
    -ranlib $@
    -rm -rf tmp $(LIBS)

$(LIBS): FRC
    -cd $(@:.a=) ; $(MAKE) $@
    -ln -s $(@:.a=)/$@ $@
FRC:
```

For the sake of clarity, this example omits support for alternate variants, as well as the targets for `clean`, `install`, and `test` (`lint` does not apply since the source files are in the subdirectories). This material is added in later examples.

The `rm -f _ _.SYMDEF` command embedded in the collating line prevents a symbol table in a subsidiary (produced by running `ranlib` on that library) from

being archived in this library.

Since the nested make commands build the subsidiary libraries before the currently library is processed, it is a simple matter to extend this makefile to account for libraries built from both subsidiaries and object files in the current directory. You need only add the list of object files to the dependency list for the library, and a command to copy them into the temporary subdirectory for collation with object files extracted from subsidiary libraries.

```
#       Simple makefile for collating a library from
#       subsidiaries and local object files.

LIBRARY= libz.a
LIBS= libx.a liby.a
SOURCES= map.o calc.o draw.o
ULIBS= $(LIBRARY)

ARFLAGS=
CFLAGS= -O
CPPFLAGS=

OBJECTS= $(SOURCES.c=.o)

.KEEP_STATE:
.PRECIOUS: libz.a

all: $(LIBRARY)

$(LIBRARY): $(LIBS) $(OBJECTS)
    -rm -rf tmp
    -mkdir tmp
        -cp $(OBJECTS) tmp
    set -x ; for i in $(LIBS) ; do \
        ( cd tmp ; ar x ../$$i ) ; \
    done
    ( cd tmp ; rm -f __.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
    -ranlib $@
    -rm -rf tmp $(LIBS)

$(LIBS): FRC
    -cd $(@:.a=) ; $(MAKE) $@
    -ln -s $(@:.a=)/$@ $@
FRC:
```

The next example includes support for debugging and profiling variants, along with recursive targets for clean, lint, test, and install.

```
#       Makefile for collating a library from local object files and
#       subsidiary libraries.  Supports alternate variants, and maintains
#       subdirectories recursively.

LIBRARY= libz.a
LIBS= libx.a liby.a
SOURCES= map.c calc.c draw.c
ULIBS= $(LIBRARY)
SLIBS= -lcurses -ltermlib

DIRS= $(LIBS:.a=)
TESTSCRIPT= testscript
TESTPROGS= test_1 test_2 test_3
DESTDIR= /proto/project/lib

ARFLAGS=
CFLAGS= -O
CPPFLAGS=
LDFLAGS=
LINTFLAGS=
TARGETS= all

OBJECTS= $(SOURCES.c:.c=.o)
LINTFILES= $(SOURCES.c:.c=.ln)
TARGETS.nested= lint.nested clean.nested test.nested \
        install.nested

.KEEP_STATE:
.PRECIOUS: libz.a

all profile debug: $(LIBRARY)

debug := CFLAGS= -g
profile := CFLAGS= -O -pg
debug := TARGET= debug
profile := TARGET= profile

$(LIBRARY): $(LIBS) $(OBJECTS)
    -rm -rf tmp
    -mkdir tmp
        -cp $(OBJECTS) tmp
    set -x ; for i in $(LIBS) ; do \
        ( cd tmp ; ar x ../$$i ) ; \
    done
    ( cd tmp ; rm -f __.SYMDEF ; ar cr ../$@ `lorder * | tsort` )
    -ranlib $@
    -rm -rf tmp $(LIBS)

$(LIBS): FRC
    -cd $(@:.a=) ; $(MAKE) $(TARGET)
    -ln -s $(@:.a=)/$@ $@
FRC:

#       Recursive targets:

lint clean test: $$@.nested $$@.local
install: $$@.local $$@.nested
```

```
#       Nested targets:

$(TARGETS.nested):
        for i in $(DIRS) ; do \
                cd $$i ; $(MAKE) $(@:.nested=) ; \
        done

#       Local target entries:

lint.local: $(LINTFILES)
    $(LINT.c) $(LINTFILES)

clean.local:
    rm -f $(LIBRARY) $(OBJECTS) $(LINTFILES) $(TESTSCRIPT) $(TESTPROGS)

test.local: all $(TESTSCRIPT)
        set -x ; $(TESTSCRIPT) > /tmp/test.$$$$
$(TESTSCRIPT): $(TESTSCRIPT).sh $(TESTPROGS)
$(TESTPROGS): $$@.c $(ULIBS)
        $(LINK.c) -o $@ $< $(ULIBS) $(SLIBS)

install.local: $(LIBRARY)
        -mkdir $(DESTDIR)
        -cp $(PROGRAM) $(DESTDIR)
```

Figure 1-16 *Makefile for a Hierarchy of Subsidiary Libraries with Variants*

In Conclusion

make has evolved into a powerful and flexible tool for consistently processing files that stand in a hierarchical relationship to one another. The methods and examples shown in this manual are intended to provide you with an exposure to the kinds of problems that lend themselves to solution with make. There is a large body of folklore about make; strong and varied opinions about its "best" use abound. This manual does not make the claim that any one approach or example is necessarily the best available. Compromises between clarity and functionality were made in many of the examples.

Also, there is considerable opinion both pro and against makefiles that use macros extensively. Some experts prefer to tailor makefiles for specific situations. Others prefer that all makefiles look the same and work the same way.

This manual takes the latter approach. The examples are intended to be useful, just as they are, in a wide variety of not-too-complicated settings. As procedures become more complicated, so do the makefiles that implement them. The trick is to know which approach will yield a reasonable makefile that works in a given situation. The examples are intended to give you a flavor for common situations, and some fairly straightforward methods to simplify them using make.

If a template approach is used in a project from the outset, chances are that custom makefiles that evolve from the templates will be more familiar, and therefore easier to understand, to integrate, to maintain, and more importantly, to *re-use.* After all, the less time you spend tinkering with the makefiles, the more time you have to develop your program or project.

A

make Enhancements Summary

make Enhancements Summary 69

A.1. New Features 69
Default Makefile 69
The State File .make.state 69
Hidden Dependency Checking 69
Command Dependency Checking 69
Automatic sccs Extraction 70
Tilde Rules Superceded 70
sccs History Files 70
Pattern Matching Rules: Convenient Implicit Rules 70
New Options 71
Support for Modula-2 71
Naming Scheme for Predefined Macros 71
New Special-Purpose Targets 71
New Implicit Rule for lint 72
Macro Processing Changes 72
Macros: Definition, Substitution, and Substring Replacement 72
Improved *ar* Library Support 72
Lists of Members 72
Handling of ar's Member-Name Length Limitation 73
Target Groups 73
A.2. Incompatibilities With Previous versions of make 73
New Meaning for -d Option 73
Dynamic Macros 73

A

make Enhancements Summary

A.1. New Features

Default Makefile

make's implicit rules and macro definitions are no longer hard-coded within the program itself. They are now contained in the default makefile /usr/include/make/default.mk. make reads this file automatically, unless there is a file in the local directory named default.mk. When you use a local default.mk file, you must add an include /usr/include/make/default.mk directive to get the standard implicit rules and predefined macros.

The State File make.state

make also reads from a state file, .make.state in the directory. When the special-function target .KEEP_STATE is used in the makefile, make writes out a cumulative report for each target containing a list of hidden dependencies (as reported by compilation processors such as cpp), and the most recent rule used to build each target. The state file is very similar in format to an ordinary makefile.

Hidden Dependency Checking

When activated by the presence of the .KEEP_STATE target, make uses information reported from cpp, f77, make, pc and other compilation commands, and performs a dependency check against any header files (or in some cases, libraries) that are incorporated into the target file. These "hidden" dependency files do not appear in the dependency list, and often do not reside in the local directory.

Command Dependency Checking

When .KEEP_STATE is in effect, if any command line used to build a target should changes between make runs, perhaps by editing the makefile, or supplying a command-line definition for (a macro like) CFLAGS, the target is treated as if it were out of date; make rebuilds it (even if it is newer that the files it depends on).

Automatic sccs Extraction

Tilde Rules Superceded

This version of make automatically runs sccs get, as appropriate, when there is no rule to build a target file. A tilde appended to a suffix in the suffixes list indicates that sccs extraction is appropriate for files having that suffix. There are no longer special versions of implicit rules that include commands to extract current versions of sccs files.

To inhibit or alter the procedure for automatic extraction of the current sccs version, redefine the .SCCS_GET special-function target. An empty rule for this target inhibits automatic extraction entirely.

sccs History Files

make no longer searches the current directory for sccs history (s.) files. These files must now reside in an SCCS subdirectory.

Pattern Matching Rules: Convenient Implicit Rules

Pattern matching rules have been added to simplify the process of adding new implicit rules of your own design. A target entry of the form:

tp%*ts* : *dp*%*ds*
    rule

defines a pattern matching rule for building a target from a a related dependency file. *tp* is the target's prefix; *ts*, its suffix. *dp* is the dependency's prefix; *ds*, its suffix. The % symbol is a wild card that matches a contiguous string of zero or more characters appearing in both the target and the dependency filename. For example, the following target entry defines a pattern matching rule for building a troff output file, ending in .tr from a file that uses the -ms macro package ending in .ms:

```
%.tr: %.ms
        troff -t -ms $< > $@
```

With this entry in the makefile, the command:

```
make doc.tr
```

produces:

```
hermes% make doc.tr
troff -t -ms doc.ms > doc.tr
```

Using that same entry, if there is a file named doc2.ms the command:

```
make doc2.tr
```

produces:

```
hermes% make doc2.tr
troff -t -ms doc2.ms > doc2.tr
```

An explicit target entry overrides any pattern matching rule that might apply to a target. Pattern matching rules, in turn, normally override implicit rules. An exception to this is when the pattern matching rule has no commands in the rule

portion of its target entry. In this case, make continues the search for a rule to build the target, and using as its dependency the file that matched the (dependency) pattern.

New Options

There are a number of new options:

-d Display dependency-check results for each target processed. Displays all dependencies that are newer, or indicates that the target was built as the result of a command dependency.

-dd The same function as -d had in earlier versions of make. Displays a great deal of output about all details of the make run, including internal states, etc.

-D Display the text of the makefile.

-DD Display the text of the makefile, and of the default makefile being used.

-p Print macro definitions and target entries.

-P Report on dependency checks without rebuilding targets.

Support for Modula-2

This version of make contains predefined macros and implicit rules for compiling Modula-2 sources.

Naming Scheme for Predefined Macros

The naming scheme for predefined macros has been rationalized, and the implicit rules have been rewritten to reflect the new scheme. The macros and implicit rules are upward compatible with existing makefiles.

For example, there is now a macro called SUFFIXES, that contains the default entries for the suffixes list; the target entry for the default suffixes list looks like:

```
.SUFFIXES: $(SUFFIXES)
```

If you want to insert new suffixes at the head of the list, you can do so quite simply as follows:

```
.SUFFIXES:
.SUFFIXES: .ms .tr $(SUFFIXES)
```

Other examples include the macros for standard compilations commands:

LINK.c Standard cc command line for producing executable files.
COMPILE.c Standard cc command line for producing object files.

New Special-Purpose Targets

The .KEEP_STATE target should not be removed once it has been used in a make run.

.KEEP_STATE When included in a makefile, this target enables hidden dependency and command dependency checking. In addition, make updates the state file .make.state after each run.

.INIT and .DONE
These targets can be used to supply commands to perform at the beginning and end, respectively, of each make run.

`.SCCS_GET` This target contains the rule for extracting current versions from `sccs` history files.

New Implicit Rule for `lint`

Implicit rules have been added to support incremental verification with `lint`.

Macro Processing Changes

A macro's value can now be of virtually any length.

Macros: Definition, Substitution, and Substring Replacement

New Append Operator: +=

This operator appends a SPACE, followed by a word or words, onto the existing value of the macro.

Conditional Macro Definitions: :=

This operator indicates a conditional (targetwise) macro definition. A makefile entry of the form:

target := *macro* = *value*

indicates that *macro* takes the indicated *value* while processing *target* and its dependencies.

Substring Replacement Precedence

Substring replacement now takes place following expansion of the macro being referred to. Previous versions of `make` applied the substitution first, with results that were counterintuitive.

Nested Macro References

`make` now expands inner references before parsing the outer reference. So, a nested reference as in this example:

```
CFLAGS-g = -I../include
OPTION = -g
$(CFLAGS$(OPTION))
```

now yields the value `-I../include`, rather than a null value, as it would have in previous versions.

Cross-Compilation Macros

The predefined macros `HOST_ARCH` and `TARGET_ARCH` are available for use in cross-compilations. The `HOST_ARCH` macro is automatically set to the type of processor in your workstation.

Improved *ar* Library Support

Lists of Members

`make` automatically updates an *ar* library member from a file having the same name as the member. Also, `make` now supports lists of members as dependency names of the form:

lib.a: *lib.a* (*member member* ...)

Handling of ar's Member-Name Length Limitation

make now copes with the 15-character member-name length limitation in ar. It now recognizes a member name that matches the first 15 characters of a filename as the member corresponding to the file.

Target Groups

It is now possible to specify that a rule produces a set of target files. A + sign between target names in the target entry indicates that the named targets comprise a group. The target group's rule is performed once, at most, in a make invocation.

A.2. Incompatibilities With Previous versions of make

New Meaning for -d Option

The –d option now reports the reason why a target is considered out of date.

Dynamic Macros

Although the dynamic macros < and * were documented being assigned only for implicit rules and the .DEFAULT target, in some cases they actually were assigned for explicit target entries. The assignment action is now documented properly.

The actual value assigned to each of these macros is derived by the same procedure used within implicit rules (this hasn't changed). This can lead to unexpected results when they are used in explicit target entries.

Even if you supply explicit dependencies, make doesn't use them to derive values for these macros. Instead, it searches for an appropriate implicit rule and dependency file. For instance, if you have the explicit target entry:

```
test: test.f
        echo $<
```

and the files: test.c and test.f, you might expect that $< would be assigned the value test.f. This is *not* the case. It is assigned test.**c**, because .c is ahead of .f in the suffixes list:

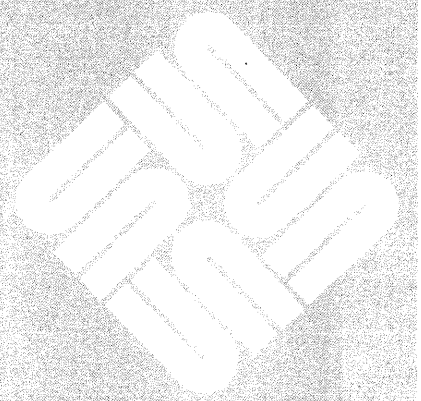
```
hermes% make test
echo test.c
test.c
```

For explicit entries, we recommend a strictly deterministic method for deriving a dependency name using macro references and suffix replacements. For example, you could use: $@.f instead of $< to derive the dependency name. To derive the basename of a .o target file, you could use the suffix replacement macro reference: $(@:.o=) instead of $*.

B

`make` Manual Page

`make` Manual Page 77



NAME

make – maintain, update, and regenerate groups of programs

SYNOPSIS

make [**–f** *makefile*] ... [**–d**] [**–dd**] [**–D**] [**–DD**] [**–e**] [**–i**] [**–k**] [**–n**] [**–p**] [**–P**] [**–q**] [**–r**] [**–s**] [**–S**] [**–t**] [*target* ...] [*macro*=*value* ...]

DESCRIPTION

make executes a list of shell commands associated with each *target*, typically to create or update a file of the same name. *makefile* contains entries that describe how to bring a target up to date with respect to others on which it depends. These prerequisite targets are called *dependencies* . Since each dependency is a target, it may have dependencies of its own.

Targets, dependencies, and sub-dependencies comprise a tree structure that *make* traces when deciding whether or not to rebuild a *target*. *make* recursively checks each *target* against its dependencies, beginning with the first target entry in *makefile* if none is supplied on the command line. If, after processing its all of its dependencies, a target file is found either to be missing, or to be older than any of its dependency files, *make* rebuilds it. Optionally with this version of *make*, a target can be treated as out-of-date when the commands used to generate it have changed.

To build a given target, *make* executes the list of commands, called a *rule*. This rule may be listed explicitly in the target's makefile entry, or it may be supplied implicitly by *make*.

If no *makefile* is specified with a **–f** option, *make* attempts to:

- read a file named *makefile*. Or, if there is an *sccs*(1) history file for it, *make* attempts to extract and read the most recent version of that file.
- read, or extract and read, the current version of a file named *Makefile*.

If no *target* is specified on the command line, *make* uses the first target defined in *makefile*.

If a *target* has no *makefile* entry, or if its entry has no rule, *make* attempts to derive a rule by each of the following methods, in turn, until a suitable rule is found. (Each method is described under USAGE below.)

- Pattern matching rules.
- Implicit rules, read in from a user-supplied makefile.
- Standard implicit rules (also known as suffix rules), typically read in from the file */usr/include/make/default.mk*.
- *sccs*(1) extraction. *make* extracts the most recent version from the *sccs* history file (if any). See the description of the .SCCS_GET: special-function target for details.
- the rule from the .DEFAULT: target entry, if there is such an entry in the makefile.

If there is no makefile entry for a target, if no rule can be derived for building it, and if no file by that name is present, *make* issues an error message and stops.

OPTIONS

–f *makefile*

Use the description file *makefile*. A – as the *makefile* argument denotes the standard input. The contents of *makefile*, when present, override the standard set of implicit rules and predefined macros. When more than one **–f** *makefile* argument pair appears, *make* uses the concatenation of those files, in order of appearance.

–d When a target is rebuilt, show the criteria by which *make* found it to be out-of-date. *make* displays any and all dependencies that are newer. In addition, *make* displays options read in from the MAKEFLAGS environment variable.

–dd Display the dependency check and processing in vast detail.

–D Show the text of the makefile.

–DD Display the text of the makefile, the default file, the state file, and all hidden-dependency reports.

–e Environment variables override assignments within makefiles.

–i Ignore error codes returned by commands. Equivalent to the special-function target **.IGNORE:**.

–k When a nonzero error status is returned by a command, abandon work on the current target, but continue with other branches that do not depend on that target.

–n No execution mode. Print commands, but do not execute them. Even lines beginning with an **@** are printed. However, if a command line contains a reference to the **$(MAKE)** macro, that line is always executed (see the discussion of **MAKEFLAGS** in *Reading Makefiles and the Environment*).

–p Print out the complete set of macro definitions and target descriptions.

–P Report dependencies recursively to show the entire dependency hierarchy, without rebuilding any targets.

–q Question mode. *make* returns a zero or nonzero status code depending on whether or not the target file is up to date.

–r Do not read in the default file.

–s Silent mode. Do not print command lines before executing them. Equivalent to the special-function target **.SILENT:**.

–S Undo the effect of the **–k** option. Stop processing when a non-zero exit status is returned by a command.

–t Touch the target files (bringing them up to date) rather than performing their rules. *This can be dangerous when files are maintained by more than one person.* When the **.KEEP_STATE:** target appears in the makefile, this option updates the state file just as if the rules had been performed.

macro=value

Macro definition. This definition remains fixed for the *make* invocation. It overrides any regular definition for the specified macro within the makefile itself, or in the environment. However, this definition can still be overridden by conditional macro assignments and delayed macro assignments in target entries.

USAGE

Refer to *Doing More With UNIX: Beginner's Guide*, and *Make* in *Programming Utilities for the Sun Workstation* for tutorial information about *make*.

Reading Makefiles and the Environment

When *make* first starts, it reads the **MAKEFLAGS** environment variable to obtain a list of options. Then it reads the command line for additional options that also take effect.

Next, *make* reads in a default makefile that typically contains predefined macro definitions, target entries for implicit rules, and additional rules, such as the rule for extracting *sccs*(1) files. If present, *make* uses the file *default.mk* in the current directory; otherwise it reads the file */usr/include/make/default.mk*, which contains the standard definitions and rules. Use the directive **include /usr/include/make/default.mk.** in your local *default.mk* file to include them.

Next, *make* imports variables from the environment (unless the **–e** option is in effect), treating them as defined macros. Because *make* uses the most recent definition it encounters, a macro definition in the makefile normally overrides an environment variable of the same name. When **–e** is in effect, however, environment variables are read in *after* all makefiles have been read. In that case, the environment variable takes precedence over any makefile definition.

Next, *make* reads the state file, *.make.state* in the local directory if it exists, and then any makefiles you specify with **–f**, or one of *makefile* or *Makefile* as described above.

Finally, (after reading the environment if **–e** is in effect), *make* reads in any macro definitions from the command line. These override macro definitions in the makefile and the environment both. But, if there is a definition for the macro in a makefile used by a nested *make* command, that definition takes effect for the nested *make*, unless you use the **–e** option. With **–e**, the nested *make* also uses the value set on the

command line.

make exports its environment variables to each command or shell that it invokes. It does not export macros defined in the makefile. If an environment variable is set, and a macro with the same name is defined on the command line, *make* exports its value as defined on the command line. Unless –e is in effect, macro definitions within the makefile take precedence over those imported from the environment.

The macros **MAKEFLAGS, MAKE** and **SHELL** are special cases. See *Special-Purpose Macros* below, for details.

Makefile Target Entries

A target entry has the following format:

target ... [: | ::] [*dependency*] ... [; *command*] ...
[*command*]
...

The first line contains the name of a target (or a space-separated list of target names), terminated with a colon or double colon. This may be followed by a *dependency*, or a dependency list that *make* checks in order. The dependency list may be terminated with a semicolon (;), which in turn can be followed by a Bourne shell command. Subsequent lines in the target entry begin with a TAB, and contain Bourne shell commands. These commands comprise a rule for building the target.

Shell commands may be continued across input lines by escaping the NEWLINE with a backslash (\). The continuing line must also start with a TAB.

To rebuild a target, *make* expands macros, strips off initial TABs and either executes the command directly (if it contains no shell metacharacters), or passes each command line to a Bourne shell for execution.

The first line that does not begin with a TAB or # begins another target or macro definition.

Makefile Special Characters

Global

Start a comment. The comment ends at the next NEWLINE. If the # follows the TAB in a command line, that line is passed to the shell (which also treats # as the start of a comment).

include *filename*
If the word **include** appears as the first seven letters of a line and is followed by a SPACE or TAB, the string that follows is taken as a filename to interpolate at that line. **include** files can be nested to a depth of no more than about 16.

Targets and Dependencies

: Target list terminator. Words following the colon are added to the dependency list for the target or targets. If a target is named in more than one colon-terminated target entry, the dependencies for all its entries are added to form that target's complete dependency list.

:: Target terminator for alternate dependencies. When used in place of a colon (:) the double-colon allows a target to be checked and updated with respect to alternate dependency lists. When the target is out-of-date with respect to dependencies listed in one entry, it is built according to the rule for that entry. When out-of-date with respect to dependencies in an alternate entry, it is built according the rule in that alternate entry. Implicit rules do not apply to double-colon targets; you must supply a rule for each entry. If no dependencies are specified, the rule is always performed.

target [+ *target*...] **:**
Target group. The rule in the target entry builds all the indicated targets as a group. It is normally performed only once per *make* run, but is checked for command dependencies every time a target in the group is encountered in the dependency scan.

% Pattern matching rule wild card character. Like the * shell wild card, % matches any string of zero or more characters occurring in both a target and the name of a dependency file. See *Pattern Matching Rules*, below for details.

Macros

= Macro definition. The word to the left of this character is the macro name; words to the right comprise its value. Leading white space between the = and the first word of the value is ignored. A word break following the = is implied. Trailing white space is included in the value.

$ Macro reference. The following character, or the parenthesized or bracketed string, is interpreted as a macro reference: *make* expands the reference (including the $) by replacing it with the macro's value.

( )
{ } Macro-name delimiters. A parenthesized or bracketed word appended to a $ is taken as the name of the macro being referred to. Without the delimiters, *make* recognizes only the first character as the macro name.

$$ A reference to the dollar-sign macro, the value of which is the character $. Used to pass variable expressions beginning with $ to the shell, to refer to environment variables which are expanded by the shell, or to delay processing of dynamic macros within the dependency list of a target, until that target is actually processed.

+= When used in place of =, appends a string to a macro definition (must be surrounded by white space, unlike =).

:= Conditional macro assignment. When preceded by a list of targets with explicit target entries, the macro definition that follows takes effect when processing only those targets, and their dependencies.

Rules

– *make* ignores any nonzero error code returned by a command line for which the first non-TAB character is a –. This character is not passed to the shell as part of the command line. *make* normally terminates when a command returns nonzero status, unless the –**i** or –**k** options, or the .IGNORE: special-function target is in effect.

@ If the first non-TAB character is a @, *make* does not print the command line before executing it. This character is not passed to the shell.

? Escape command-dependency checking. Command lines starting with this character are not subject to command dependency checking.

! Force command-dependency checking. Command-dependency checking is applied to command lines for which it would otherwise be suppressed. This checking is normally suppressed for lines that contain references to the ? dynamic macro (for example, $?).

When any combination of –, @, ?, or ! appear as the first characters after the TAB, all apply. None are passed to the shell.

Special-Function Targets

When incorporated in a makefile, the following target names perform special-functions:

.DEFAULT:
If it has an entry in the makefile, the rule for this target is used to process a target when there is no other entry for it, no rule for building it, and no *sccs*(1) history file from which to extract a current version. *make* ignores any dependencies for this target.

.DONE: If defined in the makefile, *make* processes this target and its dependencies after all other targets are built.

.IGNORE:
Ignore errors. When this target appears in the makefile, *make* ignores non-zero error codes returned from commands.

.INIT: If defined in the makefile, this target and its dependencies are built before any other targets are processed.

.KEEP_STATE:
If this target appears in the makefile, *make* updates the state file, *.make.state*, in the current directory. This target also activates command dependencies, and hidden dependency checks.

.MAKE_VERSION:
A target-entry of the form:

.MAKE_VERSION: VERSION–*number*

enables version checking. If the version of *make* differs from the version indicated, *make* issues a warning message.

.PRECIOUS:
List of files not to delete. *make* does not remove any of the files listed as dependencies for this target when interrupted. *make* normally removes the current target when it receives an interrupt.

.SCCS_GET:
This target contains the rule for extracting the current version of an *sccs*(1) file from its history file. To suppress automatic extraction, add an entry for this target, with an empty rule. to your makefile.

.SILENT:
Run silently. When this target appears in the makefile, *make* does not echo commands before executing them.

.SUFFIXES:
The suffixes list for selecting implicit rules (see *The Suffixes List*).

Command Dependencies
When the **.KEEP_STATE:** target appears in the makefile, *make* checks the command for building a target against the state file, *.make.state*. If the command has changed since the last *make* run, *make* rebuilds the target.

Hidden Dependencies
When the **.KEEP_STATE:** target appears in the makefile, *make* reads reports from *cpp*(1) and other compilation processors for any "hidden" files, such as **#include** files. If the target is out of date with respect to any of these files, *make* rebuilds it.

Macros
Entries of the form

macro=*value*

define macros. *macro* is the name of the macro, and *value*, which consists of all characters up to a comment character or unescaped NEWLINE, is the value.

Subsequent references to the macro, of the forms: $(*name*) or ${*name*} are replaced by *value*. The parentheses or brackets can be omitted in a reference to a macro with a single-character name.

Macro definitions can contain references to other macros, in which case nested references are expanded first.

Suffix Replacement Macro References
Substitutions within macros can be made as follows:

$(*name*:*str1*=*str2*)

where *str1* is either a suffix, or a word to be replaced in the macro definition, and *str2* is the replacement suffix or word. Words in a macro value are separated by SPACE, TAB, and escaped NEWLINE characters.

Appending to a Macro
Words can be appended to macro values as follows:

macro += *word* ...

The space preceding the + is required. *make* inserts a leading space between the previous value and the first appended word.

Special-Purpose Macros

When the MAKEFLAGS variable is present in the environment, *make* takes options (except for **–f**) from it, in combination with any flags entered on the command line. *make* retains this combined value as the MAKEFLAGS macro, and exports it automatically to each command or shell it invokes.

Note, however that flags passed with MAKEFLAGS are only displayed when the **–d**, or **–dd** options are in effect.

The MAKE macro is another special case. It has the value "make" by default, and temporarily overrides the **–n** option for any line in which it is referred to. This allows nested invocations of *make* written as:

$(MAKE) ...

to run recursively, with the **–n** flag in effect for all commands but *make*. This lets you use **make –n** to test an entire hierarchy of makefiles.

For compatibility with the 4.2 BSD *make*, the MFLAGS macro is set from the MAKEFLAGS variable by prepending a "–". MFLAGS is not exported automatically.

The SHELL macro, when set to a single-word value such as **/bin/csh**, indicates the name of an alternate shell to use. Note, however, that *make* executes commands containing no shell metacharacters directly. Builtin commands, such as **dirs** in the C-Shell, are not recognized unless the command line includes a metacharacter (for instance, a semicolon). This macro is neither imported from, nor exported to the environment, regardless of **–e**. To be sure it is set properly, you must define this macro within every makefile that requires it.

The **KEEP_STATE** environment variable, has the same effect as the **.KEEP_STATE:** special-function target, enabling command dependencies, hidden dependencies and writing of the state file.

Predefined Macros

make supplies the macros shown in the table that follows for compilers and their options, host architectures, and other commands.

Table of Predefined Macros		
Use	*Macro*	*Default Value*
Assembler Commands	AS	as
	ASFLAGS	
	COMPILE.s	$(AS) $(ASFLAGS) $(TARGET_ARCH)
	COMPILE.S	$(CC) $(ASFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
C Compiler Commands	CC	cc
	CFLAGS	
	CPPFLAGS	
	COMPILE.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $(LDFLAGS)
FORTRAN 77 Compiler Commands	FC	f77
	FFLAGS	
	COMPILE.f	$(FC) $(FFLAGS) $(TARGET_ARCH) -c
	LINK.f	$(FC) $(FFLAGS) $(TARGET_ARCH) $(LDFLAGS)
	COMPILE.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $(LDFLAGS)
Link Editor Command	LD	ld
	LDFLAGS	
lex *Command*	LEX	lex
	LFLAGS	
	LEX.l	$(LEX) $(LFLAGS) -t
lint *Command*	LINT	lint
	LINTFLAGS	
	LINT.c	$(LINT) $(LINTFLAGS) $(CPPFLAGS) $(TARGET_ARCH)
Modula 2 Commands	M2C	m2c
	M2FLAGS	
	MODFLAGS	
	DEFFLAGS	
	COMPILE.def	$(M2C) $(M2FLAGS) $(DEFFLAGS) $(TARGET_ARCH)
	COMPILE.mod	$(M2C) $(M2FLAGS) $(MODFLAGS) $(TARGET_ARCH)
Pascal Compiler Commands	PC	pc
	PFLAGS	
	COMPILE.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
	LINK.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $(LDFLAGS)
Ratfor Compilation Commands	RFLAGS	
	COMPILE.r	$(FC) $(FFLAGS) $(RFLAGS) $(TARGET_ARCH) -c
	LINK.r	$(FC) $(FFLAGS) $(RFLAGS) $(TARGET_ARCH) $(LDFLAGS)
rm *Command*	RM	rm -f
yacc *Command*	YACC	yacc
	YFLAGS	
	YACC.y	$(YACC) $(YFLAGS)
Suffixes List	SUFFIXES	.o .c .c~ .s .s~ .S .S~ .ln .f .f~ .F .F~ .l .l~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .y .y~ .h .h~ .sh .sh~

Unless these macros are read in as environment variables, their values are not exported by *make*. If you run *make* with any of these set in the environment, it is a good idea to add commentary to the makefile to indicate what value each is expected to take.

If **-r** is in effect, *make* does not supply these macro definitions.

Dynamic Macros

There are several dynamically maintained macros that are useful as abbreviations within rules. They are shown here as references; it is best not to define them explicitly.

$* The basename of the current target, derived as if selected for use with an implicit rule. In the case of pattern matching rules, the value is the string matched by the % .

$< The name of a dependency file, derived as if selected for use with an implicit rule.

$@ The name of the current target.

$? The list of dependencies that are newer than the target. Command-dependency checking is automatically suppressed for lines that contain this macro, just as if the command had been prefixed with a ?. See the description of ?, under *Makefile Special Characters*, above. You can force this check with the ! command-line prefix.

$% The name of the library member being processed. See *Library Maintenance* for more information.

To refer to a dynamic macro within a dependency list, precede the reference with an additional $ character (for example, $$<). Because *make* assigns $< and $* as it would for implicit rules (according to the suffixes list and the directory contents), they may be unreliable when used within explicit target entries.

All of these macros but $? can be modified to apply either to the filename part, or the directory part of the strings they stand for, by adding an upper case **F** or **D**, respectively (and enclosing the resulting name in parentheses or braces). Thus, $(@D) refers to the directory part of the string $@; if there is no directory part, . is assigned. $(@F) refers to the filename part.

Pattern Matching Rules

A target entry of the form:

Tp%Ts: *Dp%Ds*
rule

where *Tp* is a target prefix, *Ts* is a target suffix, *Dp* is a dependency prefix, and *Ds* is a dependency suffix (any of which may be null) forms a *target-dependency* pattern. When *make* encounters a target for which there is no rule, it attempts to match that target name against the target pattern. A match occurs when the target has the indicated prefix and/or suffix, in which case *make* looks for a dependency file that matches the dependency pattern and has the same root (basename) as the target. When such a file is found, *make* uses the rule in the target entry for the pattern matching rule to build the target from the dependency file. These rules typically make use of the @ and < dynamic macros as placeholders for the target and dependency names, respectively. The dynamic macro * is set to the string matched by the % wild card.

Implicit Rules

When a target has no explicit target entry and no pattern matching rule applies, *make* checks the target name to see if it contains a suffix in the suffixes list. If so, checks for an implicit rule, and a dependency file (with same basename as the target, but a different suffix) from which to build the target. The implicit rule has a target entry with a name composed of the dependency suffix and target suffix. *make* uses the rule in that target entry to build the desired target from the selected dependency file. Unlike pattern matching rules, different prefixes between a target and a dependency are not recognized. Implicit rules are also referred to as *suffix* rules.

An implicit rule is a target of the form:

DsTs:
rule

where *Ts* is the suffix of the target, *Ds* is the suffix of the dependency file, and *rule* is the implicit rule for building such a target from such a dependency file. Both *Ds* and *Ts* must appear in the suffixes list.

An implicit rule with only one suffix describes how to build a target having a null (or no) suffix, from a dependency having the indicated suffix. For instance, the .c rule describes how to build the executable *file* from a C source file.

Table of Standard Implicit Rules		
Use	*Implicit Rule Name*	*Command Line*
Assembly Files	.s.o	$(COMPILE.s) $< -o $@
	.S.o	$(COMPILE.S) $< -o $@
C *Files*	.c	$(LINK.c) $< -o $@
	.c.ln	$(LINT.c) -i $< $(OUTPUT_OPTION)
	.c.o	$(COMPILE.c) $< $(OUTPUT_OPTION)
FORTRAN 77 Files	.f	$(LINK.f) $< -o $@
	.f.o	$(COMPILE.f) $< $(OUTPUT_OPTION)
	.F	$(LINK.F) $< -o $@
	.F.o	$(COMPILE.F) $< $(OUTPUT_OPTION)
lex *Files*	.l	$(RM) $*.c $(LEX.l) $< > $*.c $(LINK.c) $*.c -o $@ $(RM) $*.c
	.l.c	$(RM) $@ $(LEX.l) $< > $@
	.l.ln	$(RM) $*.c $(LEX.l) $< > $*.c $(LINT.c) -i $*.c -o $@ $(RM) $*.c
	.l.o	$(RM) $*.c $(LEX.l) $< > $*.c $(COMPILE.c) $*.c -o $@ $(RM) $*.c
Modula 2 Files	.mod .mod.o .def.sym	$(COMPILE.mod) -e $@ $< -o $@ $(COMPILE.mod) $< -o $@ $(COMPILE.def) $< -o $@
Pascal Files	.p	$(LINK.p) $< -o $@
	.p.o	$(COMPILE.p) $< $(OUTPUT_OPTION)
Ratfor Files	.r	$(LINK.r) $< -o $@
	.r.o	$(COMPILE.r) $< $(OUTPUT_OPTION)
Shell Scripts	.sh	cp $< $@ chmod +x $@
yacc *Files*	.y	$(YACC.y) $< $(LINK.c) y.tab.c -o $@ $(RM) y.tab.c
	.y.c	$(YACC.y) $< mv y.tab.c $@
	.y.ln	$(YACC.y) $< $(LINT.c) -i y.tab.c -o $@ $(RM) y.tab.c
	.y.o	$(YACC.y) $< $(COMPILE.c) y.tab.c -o $@ $(RM) y.tab.c

make reads in the standard set of implicit rules from the file */usr/include/make/default.mk*, unless **–r** is in effect, or there is a *default.mk* file in the local directory that does not **include** it.

The Suffixes List

The suffixes list is given as the list of dependencies for the **.SUFFIXES:** special-function target. The default list is contained in the **SUFFIXES** macro (See *Table of Predefined Macros* for the standard list of suffixes). You can define additional **.SUFFIXES:** targets; a **.SUFFIXES** target with no dependencies clears the list of suffixes. Order is significant within the list; *make* selects a rule that corresponds to the target's suffix and

the first dependency-file suffix found in the list. To place suffixes at the head of the list, clear the list and replace it with the new suffixes, followed by the default list:

.SUFFIXES:
.SUFFIXES: *suffixes* **$(SUFFIXES)**

A tilde (~) indicates that if a dependency file with the indicated suffix (minus the ~) is under *sccs*(1), its most recent version should be extracted, if necessary, before the target is processed.

Library Maintenance

A target name of the form:

lib(*member* ...)

refers to a member, or a space-separated list of members, in an *ar*(1) library.

The dependency of the library member on the corresponding file must be given as an explicit entry in the makefile. This can be handled by a pattern matching rule of the form:

lib(*%.s*): *%.s*

where *.s* is the suffix of the member; this suffix is typically **.o** for object libraries.

A target name of the form

lib((*symbol*))

refers to the member of a randomized object library (see *ranlib*(1)) that defines the entry point named *symbol*.

Command Execution

Command lines are executed one at a time, *each by its own process or shell*. Shell commands, notably *cd*, are ineffectual across an unescaped NEWLINE in the makefile. A line is printed (after macro expansion) just before being executed. This is suppressed if it starts with a **@**, if there is a **.SILENT:** entry in the makefile, or if *make* is run with the **–s** option. Although the **–n** option specifies printing without execution, lines containing the macro **$(MAKE)** are executed regardless, and lines containing the **@** special character are printed. The **–t** (touch) option updates the modification date of a file without executing any rules. This can be dangerous when sources are maintained by more than one person.

To use the Bourne shell **if** control structure for branching, use a command line of the form:

if *expression* ; \
then *command* ; \
command ; \
...
elif *expression* ; \
...
else *command* ; \
fi

Although composed of several input lines, the escaped NEWLINEs insure that *make* treats them all as one command line.

To use the Bourne shell **for** control structure for loops, use a command line of the form:

for *var* in *list* ; do \
command ; \
...
done

To write shell variables, use double dollar-signs ($$). This escapes expansion of the dollar-sign by *make*.

Signals

INT and QUIT signals received from the keyboard cause *make* to halt, and to remove the target file being processed unless that target is in the dependency list for **.PRECIOUS:**.

EXAMPLES

This makefile says that **pgm** depends on two files **a.o** and **b.o**, and that they in turn depend on their

corresponding source files (**a.c** and **b.c**) along with a common file **incl.h**:

```
pgm: a.o b.o
        cc a.o b.o -o $@

a.o: incl.h a.c
        cc -c a.c

b.o: incl.h b.c
        cc -c b.c
```

The following makefile uses implicit rules to express the same dependencies:

```
pgm: a.o b.o
        cc a.o b.o -o pgm

a.o b.o: incl.h
```

FILES

[mM]akefile	Current version(s) of *make* description file.
SCCS/s.[mM]akefile	*sccs*(1) history files for the above makefile(s).
default.mk	Default file for user-defined targets, macros, and implicit rules.
/usr/include/make/default.mk	Makefile for standard implicit rules and macros (not read if *default.mk* is).
.make.state	The state file in the local directory.

DIAGNOSTICS

make returns a exit status of 1 when it halts as a result of an error. Otherwise it returns and exit status of 0.

Don't know how to make *target*. Stop.
: There is no makefile entry for *target*, and none of *make*'s implicit rules apply (there is no dependency file with a suffix in the suffixes list, or the target's suffix is not in the list).

*** *target* removed.
: *make* was interrupted while building *target*. Rather than leaving a partially-completed version that is newer than its dependencies, make removes the file named *target*.

*** *target* not removed.
: *make* was interrupted while building *target* and *target* was not present in the directory.

*** *target* could not be removed, ...
: *make* was interrupted while building *target*, which was not removed for the indicated reason.

Read of include file '*file*' failed
: The makefile indicated in an **include** directive was not found, or was inaccessible.

Loop detected when expanding macro value '*macro*'
: A reference to the macro being defined was found in the definition.

Could not write state file '*file*'
: You used the **.KEEP_STATE:** target, but do not have write permission on the state file.

SEE ALSO

cc(1), ar(1), cd(1), get(1), lex(1), ranlib(1), sh(1), sccs(1)

SunPro `make` *User's Guide*

Doing More with UNIX: Beginner's Guide

BUGS

Some commands return nonzero status inappropriately; use **-i** to overcome the difficulty.

Filenames with the characters =, :, or @, don't work.

You cannot build **file.o** from **lib(file.o)**.

Options supplied by **MAKEFLAGS** should be reported for nested *make* commands. Use the **–d** option to find out what options the nested command picks up from **MAKEFLAGS**.

This version of *make* is incompatible in certain respects with previous versions:

The **–d** option output is much briefer in this version. **–dd** now produces the equivalent voluminous output.

make attempts to derive values for the dynamic macros **$*** and **$<**, while processing explicit targets. It uses the same method as for implicit rules; in some cases this can lead either to unexpected values, or to an empty value being assigned. (Actually, this was true for earlier versions as well, even though the documentation stated otherwise.)

Make no longer searches the current directory for *sccs* history files.

Suffix replacement in macro references is now applied after the macro is expanded.

There is no guarantee that makefiles created for this version of *make* will work with earlier versions.

If there is no *default.mk* file in the current directory, and the file */usr/include/make/default.mk* is missing, *make* stops before processing any targets. To force *make* to run anyway, create an empty *default.mk* file in the current directory.

Once a dependency is made, *make* assumes the dependency file is present for the remainder of the run. If a rule subsequently removes that file and future targets depend on its existence, unexpected errors may result.

Index

Special Characters

! force command dependency checking, 15
$ macro reference indicator, 13
$$ delayed references to macros, 25
$% dynamic macro: library member, 24
$* dynamic macro: derived target basename, 24
$< dynamic macro: derived dependency, 24
$? dynamic macro: newer dependencies, 24
$@ dynamic macro: current target, 24
% pattern matching wild card, 27
( ) macro reference delimiters, 13
- ignore exit status command prefix, 11
: target entry name list terminator, 5, 33
:= conditional macro definition indicator, 37
= macro definition indicator, 13, 33
? suppress command dependency checking, 15
@ execute silently command prefix, 11
# comment indicator, 5
\ escape NEWLINE for command continuation, 6
TAB leading character in rule entry, 5
SPACE not suitable as leading character in rule entry, 5
NEWLINE command and comment terminator, 5
{ } macro reference delimiters, 14

A

added flexibility
and user-defined macros, 32
all, convention for "all" final targets, 23, 38
all.local — example convention — local subtarget for recursive all, 58
all.nested — example convention — nested subtarget for recursive all, 58
automatic extraction, 12
.SCCS_GET special target, 13
suppressing, 13

B

building an entire project, 57

C

.c — suffix convention — predefined macros for processing .c files, 21
CC — macro for cc command name, 21
CFLAGS predefined macro, 13, 21
check builds, 57
clean, convention for housekeeping, 20
closing remarks about make, 65
command dependency checking, 15
force or suppress, 15
commands
and predefined macros, 20
checked against state file, 15
ignored exit status, 11
nested make commands, 41
one shell (or process) per line in rule, 6
performed by a shell vs. directly executed, 6
silent execution, 11
comment
indicator, #, 5
terminator, NEWLINE, 5
compilation
and system-supplied libraries, 35
and user-supplied libraries, 41
complex compilations and make, 32
debugging object files, 14
debugging programs and object files, 36
example for compiling program from lex, yacc and C sources, 51
example makefile for compiling alternate library variants, 40
example makefile for compiling alternate program variants, 38
example of linking a C program with assembly routines, 48
example of linking with system-supplied libraries, 36
example of linking with user-supplied libraries, 43
optimized object files, 13
profiling object files, 14
profiling programs and object files, 36
program variants, 37
COMPILE — basename convention — predefined macros for compiling object files, 21
COMPILE.c — macro for compiling .c files into object files, 21
compiler options
-g, 14, 36
-I and the CPPFLAGS macro, 34
-l, 35
-O, 13
-pg, 14, 36
complex compilations and make, 32
conditional macro definitions, 37
conditional macro definitions — example, 37
conventions
.local — example convention — suffix of subtarget, 58
.nested — example convention — suffix of subtarget, 58

conventions, *continued*
clean — housekeeping target, 20
all — list of final targets, 23
all.local — example convention — local subtarget for recursive all, 58
all.nested — example convention — nested subtarget for recursive all, 58
.c — suffix of predefined macros for processing .c files, 21
COMPILE — basename of predefined macros for compiling object files, 21
debug — example convention — target for debugging variant, 38
DIRS — example convention — user-defined macro for list of subdirectories, 57
FILES — example convention — basename of user-defined macros for misc. files, 40
FLAGS — suffix convention — predefined macros for command arguments, 21
FRC — dummy target name, 10
.h — example convention — suffix of user-defined macros for header files, 40
install — target for installing finished programs or libraries, 56
LIBRARY — example convention — user-defined macros for object library, 40
LINK — basename of predefined macros for compiling and linking executable programs, 21
lint — example convention — target for lint, 34
LINTFILES — example convention — derived macro for lint intermediate files, 34, 35
makefiles only maintain targets in the directory they reside in, 41
naming conventions for directory of library — basename of the archive (.a) file, 62
needed increasingly as project grows, 55
OBJECTS — example convention — user-defined macros for object files, 33
profile — example convention — target for profiling variant, 38
PROGRAM — example convention — user-defined macros for object files, 33
SLIBS — example convention — user-defined macro for system-supplied libraries, 43
SOURCES — example convention — user-defined macros for source files, 33
test — example convention — target for testing, 53
TESTPROGS — example convention — macro for test programs, 54
TESTSCRIPT — example convention — macro for test script, 54
ULIBS — example convention — user-defined macro for user-supplied libraries, 43
CPPFLAGS — macro for cpp options, 21
curses and termlib libraries, linking with, 35

D

D dynamic macro modifier: directory part, 25, 41
-d option, display dependency check, 18
-dd option, display entire dependency scan, 18
debugging and profiling
library variants, 40
program variants, 38
dependencies
and conditional macro definitions, 37
and the suffixes list, 23
as distinguished from dependency files, 4
as prerequisites, 4
as targets, 4
command dependency checking, 15
hidden dependencies — missing file startup problem, 17
hidden dependency checking, 16
introduced, 4
missing entries or rules, 10
scanning and processing of, 7
with null rules, 10
directories and makefiles (warning to maintain targets in same directory), 41
directory hierarchies and recursive makefiles, 58
DIRS — example convention — user-defined macro for list of subdirectories, 57
dynamic macros
$% — library member, 24
$* — derived target basename, 24
$< — derived dependency, 24
$? — newer dependencies, 24
$@ — current target, 24
and implicit rules, 24
and references delayed with $$, 25
and the dependency list, 25
introduced, 24
value modifiers D and F, 25

E

-e option
environment overrides makefile macro definitions, 44
makefiles for testing macro definition precedence, 45
errors
Don't know how to make *'target'*. Stop., 10
examples
compiling alternate library variants, 40
compiling alternate program variants, 38
compiling program from lex, yacc and C sources, 51
conditional macro definitions, 37
implicit rules and makefiles, 22
linking a C program with assembly routines, 48
linking with system-supplied libraries, 36
linking with user-supplied libraries, 43
pattern matching rules, 27
predefined macro usage, 22
recursive makefile for C programs and subdirectories, 61
recursive makefile for hierarchical C library and subdirectories, 65
shell loop construct in makefile, 57
simple example for hierarchical library, 62
simple recursive target entry, 58
simple root makefile for project, 57
simple target entry for library, 30
suffix replacement, 34
testing with make, 53
exit status, ignored, 11
extraction from sccs, automatic, 12

F

F dynamic macro modifier: filename part, 25, 41
-f option, read named makefile, 18
FLAGS — suffix convention — predefined macros for command arguments, 21
forcing execution of a target's rule: dummy dependencies, 10
FRC — convention for dummy target, 10, 42

H

header files as hidden dependencies, 16
header files, maintaining a directory of, 40
hidden dependencies, startup problem when file is missing, 17
hidden dependency checking, 16
hierarchical libraries, 61

I

-i option, ignore exit status throughout, 11
.IGNORE — special target, 11
ignored exit status of commands, 11
implicit rules
- adding suffix rules, 26
- and dynamic macros, 24
- and macro references, 13
- and the suffixes list, 23
- basic use, 4
- .c — suffix rule, 5
- .c.ln — suffix rule, 34
- .c.o — suffix rule, 5
- example of use with makefiles, 22
- introduced, 4
- lint, 34
- .l.o — suffix rule, 49
- no transitive closure, 23
- .sh — suffix rule, 52
- table of, 27
- usage within makefiles, 22
- vs. explicit target entries, 24
- .y.o — suffix rule, 49

include *makefile* — make directive, 55
#include files as hidden dependencies, 16, 40
installing finished programs and libraries, 56

K

.KEEP_STATE — special target, 15

L

LDFLAGS: macro for ld options, 21
libraries
- 15-character member name length limit and make, 31
- and ar, ranlib, lorder and tsort, 30
- curses and termlib, 35
- description of, 30
- example of a simple target entry, 30
- hierarchical, 61
- installing finished, 56
- linking with system-supplied, 35
- linking with user-supplied, 41
- require explicit target entry, 41
- special notation for members, 30
- testing with make, 52

LIBRARY — example convention — user-defined macro for object library, 40
line breaks in rules, 6
LINK — basename convention — predefined macros for compiling and linking executable programs, 21
LINK.c — macro for compiling and linking .c files into executable programs, 21
LINT — basename convention — lint command, 34
lint
- and make, 34
- target entry for, 34

LINT.c — macro for lint, 34
LINTFILES — example convention — derived macro for lint intermediate files, 34
loop, shell for construct, in a makefile, 57

M

macros
- $% dynamic macro: library member, 24
- $* dynamic macro: derived target basename, 24
- $< dynamic macro: derived dependency, 24
- $? dynamic macro: newer dependencies, 24
- $@ dynamic macro: current target, 24
- and added flexibility, 32
- CC — macro for cc command name, 21
- CFLAGS, 13, 21
- COMPILE.c — macro for compiling .c files into object files, 21
- conditional macro definitions, 37
- CPPFLAGS — macro for cpp options, 21
- definitions in makefiles, 13
- definitions on command line, 13
- definitions: makefiles for testing definition precedence, 45
- definitions: order of precedence for nested make commands, 46
- delayed references, 25
- DIRS — example convention — user-defined macro for list of subdirectories, 57
- dynamic, 24
- FILES.h — example convention — user-defined macro for list of .h files, 40
- LDFLAGS — macro for ld options, 21
- LINK.c — macro for compiling and linking .c files into executable programs, 21
- LINT.c — macro for lint, 34
- LINTFILES — example convention — derived macro for lint intermediate files, 35
- MAKE — make command — overrides -n option, 18, 41
- MAKEFLAGS — make command options (and macro peculiarities), 18
- OUTPUT_OPTION — macro for -o *filename* compiler option, 25
- passing command-line parameters, 13
- passing parameters to nested make commands, 44
- predefined for commands, 20
- reference delimiters: ( ) or { }, 14
- references in implicit rules, 13
- references in makefile, 13
- SHELL environment variable — neither imported nor exported (footnote), 44
- SLIBS — example convention — user-defined macro for system-supplied libraries, 43
- suffix replacement macro references, 33

macros, *continued*
SUFFIXES — macro for suffix-list entries, 23
table of predefined and dynamic, 29
TARGET_ARCH — macro for cross-compilation target architecture, 21
ULIBS — user-defined macro for user-supplied libraries, 43
undefined values: set to empty string, 13
usage of predefined, 20
various uses for, 32
maintaining software projects, organization issues, 54
MAKE — predefined macro — make command (overrides -n option), 18, 41
make
and lint, 34
assumes static source files, 4
vs. sccs, 3
vs. shell scripts, 3
make command
-t (touch) option (warning against use), 19
command line options to display information, 18
passing command-line parameters, 13
passing parameters to nested make commands, 44
.make.state the state file, 16
makefiles
and implicit rules, 22
and macro definitions, 13
and macro references, 13
and protecting shell filename wild cards, 62
as recipes, 4
as specifications for compilation procedures, 33
display while processing, 18
example for compiling lex and yacc sources, 51
example of compiling alternate library variants, 40
example of compiling alternate program variants, 38
example of linking with assembly routines, 48
example of linking with system-supplied libraries, 36
example of linking with user-supplied libraries, 43
example with conditional macro definitions for alternate variants, 37
example with suffix-replacement macro references, 35
-f option, read named makefile, 18
leading TABs in rule entry, 5
nested make commands vs. recursive targets, 58
only maintain targets in current directory, 41
recursive, and directory hierarchies, 58
root makefile for project hierarchy, 55
shell, 57
simple target entry for a library, 30
the state file, .make.state, 16
with suffix replacement macro references, 34
writing a simple, 5
MAKEFLAGS
always exported, 43
introduced, 18
predefined macro (warning against defining in makefile), 43
value taken from environment value and command line options, 43
members, library
name length limit of 15 characters and make, 31
special notation, 30
metacharacters (shell) in rules, 6
multiple commands on one line in rule, 6

N

nested make commands
introduced, 41
simple target entry, 42

O

OBJECTS — example convention — derived macro for object files, 33
options
-d — display dependency check, 18
-dd — display entire dependency scan, 18
-e — environment overrides makefile macro definitions, 44
-f — read named makefile, 18
-i — ignore exit status throughout, 11
-p — display macro definitions and targets, 18
-s — execute commands silently, 11
-t — touch all targets (warning against use), 19
OUTPUT_OPTION — macro for -o *filename* compiler option, 25

P

-p option, display macro definitions and targets, 18
parameters
makefiles for testing macro definition precedence, 45
order of precedence for macro definitions, nested make commands, 46
passing to nested make commands, 44
simple passing of, 13
pattern matching rules
%: wild card character, 27
adding, 27
and null rules, 27
example, 27
override implicit (suffix) rules, 27
.PRECIOUS — special target, 31
predefined macros
and commands, 20
CFLAGS, 13
COMPILE.c — macro for compiling .c files into object files, 21
CPPFLAGS — macro for cpp options, 21
example, 22
LDFLAGS — macro for ld options, 21
LINK.c — macro for compiling and linking .c files into executable programs, 21
LINT.c — macro for lint, 34
MAKE — overrides -n option, 18
MAKEFLAGS — make command options (and macro peculiarities), 18
OUTPUT_OPTION — macro for -o *filename* compiler option, 25
SUFFIXES — macro for suffix-list entries, 23
table of, 29
TARGET_ARCH — macro for cross-compilation target architecture, 21
usage, 20
prefixes
! — force command dependency check, 15
- — ignored exit status, 11
? — suppress command dependency check, 15
@ — silent command line, 11
preserve target file (library) against interrupts, 31

PROGRAM — example convention — user-defined macro for executable program, 33
programs
  installing finished, 56
  simple entry for a C program, 7
  testing with make, 52
projects
  and nested make commands, 57

R

recursive makefiles and directory hierarchies, 58
recursive targets, as distinct from nested make commands, 58
recursive targets, nested before local subtargets (except install), 59
root makefile of project hierarchy, 55
rules
  and line breaks, 6
  and shell metacharacters, 6
  .c — suffix rule, 5
  .c.o — suffix rule, 5
  forcing execution: dummy dependencies, 10
  ignoring command's exit status, 11
  leading TABs in makefile entry, 5
  .l.o — suffix rule, 49
  multiple commands per line, 6
  null rules in target entries, 10
  one shell (or process) per command line, 6
  pattern matching rules override implicit (suffix) rules, 27
  running commands based on changes in, 15
  running commands silently, 11
  .sh — suffix rule, 52
  table of implicit rules, 27
  target entries for implicit (suffix) rules, 26
  target entries for pattern matching, 27
  .y.o — suffix rule, 49
running tests with make, 52

S

-s option, execute commands silently, 11
sccs
  automatic extraction, 12
  extraction command: sccs get -G*filename filename*, 12
  .SCCS_GET — special target, 13
  suppressing automatic extraction, 13
  vs. make, 3
.SCCS_GET special target, 13
SHELL environment variable — neither imported nor exported (footnote), 44
shell commands
  for, 57
  set -x, 53
shell metacharacters
  in rules, 6
  protecting filename wild cards in makefiles, 62
shell scripts
  maintained with makefP and sccs, 52
  vs. make, 3
shell variables, delayed references with $$, 53
silent execution of commands, 11
.SILENT — special target, 11
SLIBS — example convention — user-defined macro for system-supplied libraries, 43
source files must be static, 4
SOURCES — example convention — user-defined macro for source files, 33
special targets
  .IGNORE, 11
  .KEEP_STATE, 15
  .PRECIOUS, 31
  .SCCS_GET, 13
  .SILENT, 11
  .SUFFIXES, 23
(the) state file, .make.state, 16
suffix replacement
  example in makefile, 34
  in macro references, 33
suffix rules
  adding, 26
  and the suffixes list, 23
  .c — suffix rule, 5
  .c.ln — suffix rule, 34
  .c.o — suffix rule, 5
  introduced, 4
  .l.o — suffix rule, 49
  .sh — suffix rule, 52
  table of, 27
  .y.o — suffix rule, 49
SUFFIXES — macro for suffix-list entries, 23
suffixes
  (null) — executable (a.out format) file, 5
  .c — C source file, 5
  .h — C header file, 40
  .l — lex source file, 49
  .ln — lint intermediate file, 34
  .mk — makefiles, 55
  .o — object file, 5
  .s — assembly source file, 47
  .sh — shell script source under sccs, 52
  .y — yacc source file, 49
suffixes list, the, 23
.SUFFIXES — special target, 23
suppressing automatic sccs extraction, 13

T

-t option, touch all targets (warning against use), 19
table of implicit (suffix) rules, 27
table of predefined macros, 29
TARGET_ARCH — macro for cross-compilation target architecture, 21
targets
  clean — housekeeping target, 20
  all — list of final targets, 23
  all.local — example convention — local subtarget for recursive all, 58
  all.nested — example convention — nested subtarget for recursive all, 58
  and command dependencies, 15
  and conditional macro definitions, 37
  and dependency checking, 7
  and hidden dependencies, 16
  and the suffixes list, 23
  as distinguished from target files, 4
  debug — example convention — target for debugging vari-

ant,
targets, *continued*
38
dependencies with null rules, 10
entries for implicit (suffix) rules, 26
entries for pattern matching rules, 27
for grouping a list of dependencies, 9
`.IGNORE`, 11
in other directories — nested `make` commands, 42
`install` — target for installing finished programs or libraries, 56
introduced, 4
`.KEEP_STATE`, 15
libraries require explicit entry, 41
`lint` — example convention — target for `lint`, 34
missing (null) rules, 10
missing dependencies, 10
missing target entries, 10
nested `make` command, 42
not in the dependency tree, 8
omitted from processing, 8
`profile` — example convention — target for profiling variant, 38
`.SCCS_GET`, 13
`.SILENT`, 11
simple entry for a library, 30
`.SUFFIXES`, 23
`-t` (touch) option (warning against use), 19
target entry for `lint`, 34
target entry format, 5
when to use explicit entries vs. implicit rules, 24
`termlib` and `curses` libraries, linking with, 35
test scripts and programs, 52
transitive closure, none for implicit rules, 23

U

`ULIBS` — example convention — user-defined macro for user-supplied libraries, 43

V

variant object files and programs from the same sources, 37

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 287815

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Sun House
31-41 Pembroke Broadway
Camberley
Surrey GU15 3XD
England
0276 62111
TLX 859017

Australia: 61-2-436-4699
Canada: 416 477-6745
France: (1) 46 30 23 24
Germany: (089) 95094-0
Japan: (03) 221-7021
The Netherlands: 02155 24888
UK: 0276 62111

Europe, Middle East, and Africa, call European Headquarters:
0276 62111

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales