PMake — A Tutorial

Adam de Boor

Berkeley Softworks
2150 Shattuck Ave, Penthouse
Berkeley, CA 94704
adam@bsw.uu.net
...!uunet!bsw!adam

1. Introduction

PMake is a program for creating other programs, or anything else you can think of for it to do. The basic idea behind PMake is that, for any given system, be it a program or a document or whatever, there will be some files that depend on the state of other files (on when they were last modified). PMake takes these dependencies, which you must specify, and uses them to build whatever it is you want it to build.

PMake is almost fully-compatible with Make, with which you may already be familiar. PMake's most important feature is its ability to run several different jobs at once, making the creation of systems considerably faster. It also has a great deal more functionality than Make. Throughout the text, whenever something is mentioned that is an important difference between PMake and Make (i.e. something that will cause a makefile to fail if you don't do something about it), or is simply important, it will be flagged with a little sign in the left margin, like this:

NOTE

This tutorial is divided into three main sections corresponding to basic, intermediate and advanced PMake usage. If you already know Make well, you will only need to skim chapter 2 (there are some aspects of PMake that I consider basic to its use that didn't exist in Make). Things in chapter 3 make life much easier, while those in chapter 4 are strictly for those who know what they are doing. Chapter 5 has definitions for the jargon I use and chapter 6 contains possible solutions to the problems presented throughout the tutorial.

2. The Basics of PMake

PMake takes as input a file that tells a) which files depend on which other files to be complete and b) what to do about files that are "out-of-date." This file is known as a "makefile" and is usually kept in the top-most directory of the system to be built. While you can call the makefile anything you want, PMake will look for `Makefile` and `makefile` (in that order) in the current directory if you don't tell it otherwise. To specify a different makefile, use the **–f** flag (e.g. "`pmake -f program.mk`").

A makefile has four different types of lines in it:

- File dependency specifications
- Creation commands
- Variable assignments
- Comments, include statements and conditional directives

Any line may be continued over multiple lines by ending it with a backslash. The backslash, following newline and any initial whitespace on the following line are compressed into a single space before the input line is examined by PMake.

2.1. Dependency Lines

As mentioned in the introduction, in any system, there are dependencies between the files that make up the system. For instance, in a program made up of several C source files and one header file, the C files will need to be re-compiled should the header file be changed. For a document of several chapters and one macro file, the chapters will need to be reprocessed if any of the macros changes. These are dependencies and are specified by means of dependency lines in the makefile.

On a dependency line, there are targets and sources, separated by a one- or two-character operator. The targets "depend" on the sources and are usually created from them. Any number of targets and sources may be specified on a dependency line. All the targets in the line are made to depend on all the sources. Targets and sources need not be actual files, but every source must be either an actual file or another target in the makefile. If you run out of room, use a backslash at the end of the line to continue onto the next one.

Any file may be a target and any file may be a source, but the relationship between the two (or however many) is determined by the "operator" that separates them. Three types of operators exist: one specifies that the datedness of a target is determined by the state of its sources, while another specifies other files (the sources) that need to be dealt with before the target can be re-created. The third operator is very similar to the first, with the additional condition that the target is out-of-date if it has no sources. These operations are represented by the colon, the exclamation point and the double-colon, respectively, and are mutually exclusive. Their exact semantics are as follows:

: If a colon is used, a target on the line is considered to be "out-of-date" (and in need of creation) if

- any of the sources has been modified more recently than the target, or
- the target doesn't exist.

Under this operation, steps will be taken to re-create the target only if it is found to be out-of-date by using these two rules.

! If an exclamation point is used, the target will always be re-created, but this will not happen until all of its sources have been examined and re-created, if necessary.

:: If a double-colon is used, a target is out-of-date if:

- any of the sources has been modified more recently than the target, or
- the target doesn't exist, or
- the target has no sources.

If the target is out-of-date according to these rules, it will be re-created. This operator also does something else to the targets, but I'll go into that in the next section ("Shell Commands").

Enough words, now for an example. Take that C program I mentioned earlier. Say there are three C files (`a.c`, `b.c` and `c.c`) each of which includes the file `defs.h`. The dependencies between the files could then be expressed as follows:

```
program         : a.o b.o c.o
a.o b.o c.o     : defs.h
a.o             : a.c
b.o             : b.c
c.o             : c.c
```

You may be wondering at this point, where `a.o`, `b.o` and `c.o` came in and why *they* depend on `defs.h` and the C files don't. The reason is quite simple: `program` cannot be made by linking together .c files — it must be made from .o files. Likewise, if you change `defs.h`, it isn't the .c files that need to be re-created, it's the .o files. If you think of dependencies in these terms — which files (targets) need to be created from which files (sources) — you should have no problems.

An important thing to notice about the above example, is that all the .o files appear as targets on more than one line. This is perfectly all right: the target is made to depend on all the sources mentioned on all the dependency lines. E.g. `a.o` depends on both `defs.h` and `a.c`.

NOTE The order of the dependency lines in the makefile is important: the first target on the first dependency line in the makefile will be the one that gets made if you don't say otherwise. That's why `program` comes first in the example makefile, above.

Both targets and sources may contain the standard C-Shell wildcard characters (`{`, `}`, `*`, `?`, `[`, and `]`), but the non-curly-brace ones may only appear in the final component (the file portion) of the target or source. The characters mean the following things:

{} These enclose a comma-separated list of options and cause the pattern to be expanded once for each element of the list. Each expansion contains a different element. For example, `src/{whiffle,beep,fish}.c` expands to the three words `src/whiffle.c`, `src/beep.c`, and `src/fish.c`. These braces may be nested and, unlike the other wildcard characters, the resulting words need not be actual files. All other wildcard characters are expanded using the files that exist when PMake is started.

***** This matches zero or more characters of any sort. `src/*.c` will expand to the same three words as above as long as `src` contains those three files (and no other files that end in `.c`).

? Matches any single character.

[] This is known as a character class and contains either a list of single characters, or a series of character ranges (`a-z`, for example means all characters between a and z), or both. It matches any single character contained in the list. E.g. `[A-Za-z]` will match all letters, while `[0123456789]` will match all numbers.

2.2. Shell Commands

''Isn't that nice,'' you say to yourself, ''but how are files actually 're-created,' as he likes to spell it?'' The re-creation is accomplished by commands you place in the makefile. These commands are passed to the Bourne shell (better known as ''/bin/sh'') to be executed and are expected to do what's necessary to update the target file (PMake doesn't actually check to see if the target was created. It just assumes it's there).

Shell commands in a makefile look a lot like shell commands you would type at a terminal, with one important exception: each command in a makefile *must* be preceded by at least one tab.

Each target has associated with it a shell script made up of one or more of these shell commands. The creation script for a target should immediately follow the dependency line for that target. While any given target may appear on more than one dependency line, only one of these dependency lines may be followed by a creation script, unless the '::' operator was used on the dependency line.

NOTE If the double-colon was used, each dependency line for the target may be followed by a shell script. That script will only be executed if the target on the associated dependency line is out-of-date with respect to the sources on that line, according to the rules I gave earlier. I'll give you a good example of this later on.

To expand on the earlier makefile, you might add commands as follows:

```
program         : a.o b.o c.o
        cc a.o b.o c.o -o program
a.o b.o c.o     : defs.h
a.o             : a.c
        cc -c a.c
b.o             : b.c
        cc -c b.c
c.o             : c.c
        cc -c c.c
```

Something you should remember when writing a makefile is, the commands will be executed if the *target* on the dependency line is out-of-date, not the sources. In this example, the command ''`cc -c a.c`'' will be executed if `a.o` is out-of-date. Because of the ':' operator, this means that should `a.c` *or* `defs.h` have been modified more recently than `a.o`, the command will be executed (`a.o` will be considered out-of-date).

Remember how I said the only difference between a makefile shell command and a regular shell command was the leading tab? I lied. There is another way in which makefile commands differ from regular ones. The first two characters after the initial whitespace are treated specially. If they are any combination of '@' and '−', they cause PMake to do different things.

In most cases, shell commands are printed before they're actually executed. This is to keep you informed of what's going on. If an '@' appears, however, this echoing is suppressed. In the case of an `echo` command, say "`echo Linking index`," it would be rather silly to see

```
echo Linking index
Linking index
```

so PMake allows you to place an '@' before the command ("`@echo Linking index`") to prevent the command from being printed.

The other special character is the '−'. In case you didn't know, shell commands finish with a certain "exit status." This status is made available by the operating system to whatever program invoked the command. Normally this status will be 0 if everything went ok and non-zero if something went wrong. For this reason, PMake will consider an error to have occurred if one of the shells it invokes returns a non-zero status. When it detects an error, PMake's usual action is to abort whatever it's doing and exit with a non-zero status itself (any other targets that were being created will continue being made, but nothing new will be started. PMake will exit after the last job finishes). This behavior can be altered, however, by placing a '−' at the front of a command ("`−mv index index.old`"), certain command-line arguments, or doing other things, to be detailed later. In such a case, the non-zero status is simply ignored and PMake keeps chugging along.

NOTE

Because all the commands are given to a single shell to execute, such things as setting shell variables, changing directories, etc., last beyond the command in which they are found. This also allows shell compound commands (like `for` loops) to be entered in a natural manner. Since this could cause problems for some makefiles that depend on each command being executed by a single shell, PMake has a **−B** flag (it stands for backwards-compatible) that forces each command to be given to a separate shell. It also does several other things, all of which I discourage since they are now old-fashioned. . . .

NOTE

A target's shell script is fed to the shell on its (the shell's) input stream. This means that any commands, such as `ci` that need to get input from the terminal won't work right — they'll get the shell's input, something they probably won't find to their liking. A simple way around this is to give a command like this:

```
ci $(SRCS) < /dev/tty
```

This would force the program's input to come from the terminal. If you can't do this for some reason, your only other alternative is to use PMake in its fullest compatibility mode. See **Compatibility** in chapter 4.

2.3. Variables

PMake, like Make before it, has the ability to save text in variables to be recalled later at your convenience. Variables in PMake are used much like variables in the shell and, by tradition, consist of all upper-case letters (you don't *have* to use all upper-case letters. In fact there's nothing to stop you from calling a variable `@^&$%$`. Just tradition). Variables are assigned-to using lines of the form

```
VARIABLE = value
```

appended-to by

```
VARIABLE += value
```

conditionally assigned-to (if the variable isn't already defined) by

```
VARIABLE ?= value
```

and assigned-to with expansion (i.e. the value is expanded (see below) before being assigned to the variable—useful for placing a value at the beginning of a variable, or other things) by

```
VARIABLE := value
```

Any whitespace before *value* is stripped off. When appending, a space is placed between the old value and the stuff being appended.

The final way a variable may be assigned to is using

```
VARIABLE != shell-command
```

In this case, *shell-command* has all its variables expanded (see below) and is passed off to a shell to execute. The output of the shell is then placed in the variable. Any newlines (other than the final one) are replaced by spaces before the assignment is made. This is typically used to find the current directory via a line like:

```
CWD             != pwd
```

Note: this is intended to be used to execute commands that produce small amounts of output (e.g. "pwd"). The implementation is less than intelligent and will likely freeze if you execute something that produces thousands of bytes of output (8 Kb is the limit on many UNIX systems).

The value of a variable may be retrieved by enclosing the variable name in parentheses or curly braces and preceeding the whole thing with a dollar sign.

For example, to set the variable CFLAGS to the string "`-I/sprite/src/lib/libc -O`," you would place a line

```
CFLAGS = -I/sprite/src/lib/libc -O
```

in the makefile and use the word `$(CFLAGS)` wherever you would like the string
NOTE `-I/sprite/src/lib/libc -O` to appear. This is called variable expansion.

Unlike Make, PMake will not expand a variable unless it knows the variable exists. E.g. if you have a `${i}` in a shell command and you have not assigned a value to the variable `i` (the empty string is considered a value, by the way), where Make would have substituted the empty string, PMake will leave the `${i}` alone. To keep PMake from substituting for a variable it knows, precede the dollar sign with another dollar sign. (e.g. to pass `${HOME}` to the shell, use `$${HOME}`). This causes PMake, in effect, to expand the `$` macro, which expands to a single `$`. For compatibility, Make's style of variable expansion will be used if you invoke PMake with any of the compatibility flags (**–V**, **–B** or **–M**. The **–V** flag alters just the variable expansion).

There are two different times at which variable expansion occurs: When parsing a dependency line, the expansion occurs immediately upon reading the line. If any variable used on a dependency line is undefined, PMake will print a message and exit. Variables in shell commands are expanded when the command is executed. Variables used inside another variable are expanded whenever the outer variable is expanded (the expansion of an inner variable has no effect on the outer variable. I.e. if the outer variable is used on a dependency line and in a shell command, and the inner variable changes value between when the dependency line is read and the shell command is executed, two different values will be substituted for the outer variable).

Variables come in four flavors, though they are all expanded the same and all look about the same. They are (in order of expanding scope):

- Local variables.
- Command-line variables.
- Global variables.
- Environment variables.

The classification of variables doesn't matter much, except that the classes are searched from the top (local) to the bottom (environment) when looking up a variable. The first one found wins.

2.3.1. Local Variables

Each target can have as many as seven local variables. These are variables that are only ‘‘visible’’ within that target’s shell script and contain such things as the target’s name, all of its sources (from all its dependency lines), those sources that were out-of-date, etc. Four local variables are defined for all targets. They are:

.TARGET
: The name of the target.

.OODATE
: The list of the sources for the target that were considered out-of-date. The order in the list is not guaranteed to be the same as the order in which the dependencies were given.

.ALLSRC
: The list of all sources for this target in the order in which they were given.

.PREFIX
: The target without its suffix and without any leading path. E.g. for the target `../../lib/compat/fsRead.c`, this variable would contain `fsRead`.

Three other local variables are set only for certain targets under special circumstances. These are the ‘‘.IMPSRC,’’ ‘‘.ARCHIVE,’’ and ‘‘.MEMBER’’ variables. When they are set and how they are used is described later.

Four of these variables may be used in sources as well as in shell scripts. These are ‘‘.TARGET’’, ‘‘.PREFIX’’, ‘‘.ARCHIVE’’ and ‘‘.MEMBER’’. The variables in the sources are expanded once for each target on the dependency line, providing what is known as a ‘‘dynamic source,’’ allowing you to specify several dependency lines at once. For example,

```
$(OBJS)          : $(.PREFIX).c
```

will create a dependency between each object file and its corresponding C source file.

2.3.2. Command-line Variables

Command-line variables are set when PMake is first invoked by giving a variable assignment as one of the arguments. For example,

```
pmake "CFLAGS = -I/sprite/src/lib/libc -O"
```

would make `CFLAGS` be a command-line variable with the given value. Any assignments to `CFLAGS` in the makefile will have no effect, because once it is set, there is (almost) nothing you can do to change a command-line variable (the search order, you see). Command-line variables may be set using any of the four assignment operators, though only = and `?=` behave as you would expect them to, mostly because assignments to command-line variables are performed before the makefile is read, thus the values set in the makefile are unavailable at the time. += is the same as =, because the old value of the variable is sought only in the scope in which the assignment is taking place (for reasons of efficiency that I won’t get into here). `:=` and `?=` will work if the only variables used are in the environment. `!=` is sort of pointless to use from the command line, since the same effect can no doubt be accomplished using the shell’s own command substitution mechanisms (backquotes and all that).

2.3.3. Global Variables

Global variables are those set or appended-to in the makefile. There are two classes of global variables: those you set and those PMake sets. As I said before, the ones you set can have any name you want them to have, except they may not contain a colon or an exclamation point. The variables PMake sets (almost) always begin with a period and always contain upper-case letters, only. The variables are as follows:

.PMAKE
: The name by which PMake was invoked is stored in this variable. For compatibility, the name is also stored in the MAKE variable.

.MAKEFLAGS
All the relevant flags with which PMake was invoked. This does not include such things as **−f** or variable assignments. Again for compatibility, this value is stored in the MFLAGS variable as well.

Two other variables, ".INCLUDES" and ".LIBS," are covered in the section on special targets in chapter 3.

Global variables may be deleted using lines of the form:

`#undef` *variable*

The '#' must be the first character on the line. Note that this may only be done on global variables.

2.3.4. Environment Variables

Environment variables are passed by the shell that invoked PMake and are given by PMake to each shell it invokes. They are expanded like any other variable, but they cannot be altered in any way.

One special environment variable, `PMAKE`, is examined by PMake for command-line flags, variable assignments, etc., it should always use. This variable is examined before the actual arguments to PMake are. In addition, all flags given to PMake, either through the `PMAKE` variable or on the command line, are placed in this environment variable and exported to each shell PMake executes. Thus recursive invocations of PMake automatically receive the same flags as the top-most one.

Using all these variables, you can compress the sample makefile even more:

```
OBJS            = a.o b.o c.o
program         : $(OBJS)
        cc $(.ALLSRC) -o $(.TARGET)
$(OBJS)         : defs.h
a.o             : a.c
        cc -c a.c
b.o             : b.c
        cc -c b.c
c.o             : c.c
        cc -c c.c
```

2.4. Comments

Comments in a makefile start with a '#' character and extend to the end of the line. They may appear anywhere you want them, except in a shell command (though the shell will treat it as a comment, too). If, for some reason, you need to use the '#' in a variable or on a dependency line, put a backslash in front of it. PMake will compress the two into a single '#' (Note: this isn't true if PMake is operating in full-compatibility mode).

NOTE

2.5. Parallelism

PMake was specifically designed to re-create several targets at once, when possible. You do not have to do anything special to cause this to happen (unless PMake was configured to not act in parallel, in which case you will have to make use of the **−L** and **−J** flags (see below)), but you do have to be careful at times.

There are several problems you are likely to encounter. One is that some makefiles (and programs) are written in such a way that it is impossible for two targets to be made at once. The program `xstr`, for example, always modifies the files `strings` and `x.c`. There is no way to change it. Thus you cannot run two of them at once without something being trashed. Similarly, if you have commands in the makefile that always send output to the same file, you will not be able to make more than one target at once unless you change the file you use. You can, for instance, add a `$$$$` to the end of the file name to tack on the process ID of the shell executing the command (each `$$` expands to a single `$`, thus giving you the shell variable `$$`). Since only one shell is used for all the commands, you'll get the same file name for each command in the script.

The other problem comes from improperly-specified dependencies that worked in Make because of its sequential, depth-first way of examining them. While I don't want to go into depth on how PMake works (look in chapter 4 if you're interested), I will warn you that files in two different "levels" of the dependency tree may be examined in a different order in PMake than they were in Make. For example, given the makefile

```
a               : b c
b               : d
```

PMake will examine the targets in the order `c`, `d`, `b`, `a`. If the makefile's author expected PMake to abort before making `c` if an error occurred while making `b`, or if `b` needed to exist before `c` was made, s/he will be sorely disappointed. The dependencies are incomplete, since in both these cases, `c` would depend on `b`. So watch out.

Another problem you may face is that, while PMake is set up to handle the output from multiple jobs in a graceful fashion, the same is not so for input. It has no way to regulate input to different jobs, so if you use the redirection from `/dev/tty` I mentioned earlier, you must be careful not to run two of the jobs at once.

2.6. Writing and Debugging a Makefile

Now you know most of what's in a makefile, what do you do next? There are two choices: (1) use one of the uncommonly-available makefile generators or (2) write your own makefile (I leave out the third choice of ignoring PMake and doing everything by hand as being beyond the bounds of common sense).

When faced with the writing of a makefile, it is usually best to start from first principles: just what *are* you trying to do? What do you want the makefile finally to produce?

To begin with a somewhat traditional example, let's say you need to write a makefile to create a program, `expr`, that takes standard infix expressions and converts them to prefix form (for no readily apparent reason). You've got three source files, in C, that make up the program: `main.c`, `parse.c`, and `output.c`. Harking back to my pithy advice about dependency lines, you write the first line of the file:

```
expr            : main.o parse.o output.o
```

because you remember `expr` is made from `.o` files, not `.c` files. Similarly for the `.o` files you produce the lines:

```
main.o          : main.c
parse.o         : parse.c
output.o        : output.c
main.o parse.o output.o : defs.h
```

Great. You've now got the dependencies specified. What you need now is commands. These commands, remember, must produce the target on the dependency line, usually by using the sources you've listed. You remember about local variables? Good, so it should come to you as no surprise when you write

```
expr            : main.o parse.o output.o
        cc -o $(.TARGET) $(.ALLSRC)
```

Why use the variables? If your program grows to produce postfix expressions too (which, of course, requires a name change or two), it is one fewer place you have to change the file. You cannot do this for the object files, however, because they depend on their corresponding source files *and* `defs.h`, thus if you said

```
    cc -c $(.ALLSRC)
```

you'd get (for `main.o`):

```
    cc -c main.c defs.h
```

which is wrong. So you round out the makefile with these lines:

```
main.o          : main.c
        cc -c main.c
parse.o         : parse.c
        cc -c parse.c
output.o        : output.c
        cc -c output.c
```

The makefile is now complete and will, in fact, create the program you want it to without unnecessary compilations or excessive typing on your part. There are two things wrong with it, however (aside from it being altogether too long, something I'll address in chapter 3):

1) The string "`main.o parse.o output.o`" is repeated twice, necessitating two changes when you add postfix (you were planning on that, weren't you?). This is in direct violation of de Boor's First Rule of writing makefiles:

   Anything that needs to be written more than once should be placed in a variable.

   I cannot emphasize this enough as being very important to the maintenance of a makefile and its program.

2) There is no way to alter the way compilations are performed short of editing the makefile and making the change in all places. This is evil and violates de Boor's Second Rule, which follows directly from the first:

   Any flags or programs used inside a makefile should be placed in a variable so they may be changed, temporarily or permanently, with the greatest ease.

The makefile should more properly read:

```
OBJS            = main.o parse.o output.o
expr            : $(OBJS)
        $(CC) $(CFLAGS) -o $(.TARGET) $(.ALLSRC)
main.o          : main.c
        $(CC) $(CFLAGS) -c main.c
parse.o         : parse.c
        $(CC) $(CFLAGS) -c parse.c
output.o        : output.c
        $(CC) $(CFLAGS) -c output.c
$(OBJS)         : defs.h
```

Alternatively, if you like the idea of dynamic sources mentioned in section 2.3.1, you could write it like this:

```
OBJS            = main.o parse.o output.o
expr            : $(OBJS)
        $(CC) $(CFLAGS) -o $(.TARGET) $(.ALLSRC)
$(OBJS)         : $(.PREFIX).c defs.h
        $(CC) $(CFLAGS) -c $(.PREFIX).c
```

These two rules and examples lead to de Boor's First Corollary:

Variables are your friends.

Once you've written the makefile comes the sometimes-difficult task of making sure the darn thing works. Your most helpful tool to make sure the makefile is at least syntactically correct is the **−n** flag, which allows you to see if PMake will choke on the makefile. The second thing the **−n** flag lets you do is see what PMake would do without it actually doing it, thus you can make sure the right commands would be executed were you to give PMake its head.

When you find your makefile isn't behaving as you hoped, the first question that comes to mind (after "What time is it, anyway?") is "Why not?" In answering this, two flags will serve you well: "`-d m`" and "`-p 2`." The first causes PMake to tell you as it examines each target in the makefile and indicate why it is deciding whatever it is deciding. You can then use the information printed for other targets to see where you went wrong. The “-p 2” flag makes PMake print out its internal state when it is done, allowing you to see that you forgot to make that one chapter depend on that file of macros you just got a new version of. The output from “-p 2” is intended to resemble closely a real makefile, but with additional information provided and with variables expanded in those commands PMake actually printed or executed.

Something to be especially careful about is circular dependencies. E.g.

```
a       : b
b       : c d
d       : a
```

In this case, because of how PMake works, `c` is the only thing PMake will examine, because `d` and `a` will effectively fall off the edge of the universe, making it impossible to examine `b` (or them, for that matter). PMake will tell you (if run in its normal mode) all the targets involved in any cycle it looked at (i.e. if you have two cycles in the graph (naughty, naughty), but only try to make a target in one of them, PMake will only tell you about that one. You'll have to try to make the other to find the second cycle). When run as Make, it will only print the first target in the cycle.

2.7. Invoking PMake

PMake comes with a wide variety of flags to choose from. They may appear in any order, interspersed with command-line variable assignments and targets to create. The flags are as follows:

–d *what*

This causes PMake to spew out debugging information that may prove useful to you. If you can't figure out why PMake is doing what it's doing, you might try using this flag. The *what* parameter is a string of single characters that tell PMake what aspects you are interested in. Most of what I describe will make little sense to you, unless you've dealt with Make before. Just remember where this table is and come back to it as you read on. The characters and the information they produce are as follows:

- a Archive searching and caching.
- c Conditional evaluation.
- d The searching and caching of directories.
- j Various snippets of information related to the running of the multiple shells. Not particularly interesting.
- m The making of each target: what target is being examined; when it was last modified; whether it is out-of-date; etc.
- p Makefile parsing.
- r Remote execution.
- s The application of suffix-transformation rules. (See chapter 3)
- t The maintenance of the list of targets.
- v Variable assignment.

Of these all, the `m` and `s` letters will be most useful to you. If the **–d** is the final argument or the argument from which it would get these key letters (see below for a note about which argument would be used) begins with a **–**, all of these debugging flags will be set, resulting in massive amounts of output.

–f *makefile*

Specify a makefile to read different from the standard makefiles (`Makefile` or `makefile`). If *makefile* is “–”, PMake uses the standard input. This is useful for making quick and dirty makefiles. . .

–h Prints out a summary of the various flags PMake accepts. It can also be used to find out what level of concurrency was compiled into the version of PMake you are using (look at **–J** and **–L**) and various other information on how PMake was configured.

–i If you give this flag, PMake will ignore non-zero status returned by any of its shells. It's like placing a '–' before all the commands in the makefile.

–k This is similar to **–i** in that it allows PMake to continue when it sees an error, but unlike **–i**, where PMake continues blithely as if nothing went wrong, **–k** causes it to recognize the error and only continue work on those things that don't depend on the target, either directly or indirectly (through depending on something that depends on it), whose creation returned the error. The 'k' is for "keep going"...

–l PMake has the ability to lock a directory against other people executing it in the same directory (by means of a file called "LOCK.make" that it creates and checks for in the directory). This is a Good Thing because two people doing the same thing in the same place can be disastrous for the final product (too many cooks and all that). Whether this locking is the default is up to your system administrator. If locking is on, **–l** will turn it off, and vice versa. Note that this locking will not prevent *you* from invoking PMake twice in the same place — if you own the lock file, PMake will warn you about it but continue to execute.

–n This flag tells PMake not to execute the commands needed to update the out-of-date targets in the makefile. Rather, PMake will simply print the commands it would have executed and exit. This is particularly useful for checking the correctness of a makefile. If PMake doesn't do what you expect it to, it's a good chance the makefile is wrong.

–p *number*
This causes PMake to print its input in a reasonable form, though not necessarily one that would make immediate sense to anyone but me. The *number* is a bitwise-or of 1 and 2 where 1 means it should print the input before doing any processing and 2 says it should print it after everything has been re-created. Thus `-p 3` would print it twice—once before processing and once after (you might find the difference between the two interesting). This is mostly useful to me, but you may find it informative in some bizarre circumstances.

–q If you give PMake this flag, it will not try to re-create anything. It will just see if anything is out-of-date and exit non-zero if so.

–r When PMake starts up, it reads a default makefile that tells it what sort of system it's on and gives it some idea of what to do if you don't tell it anything. I'll tell you about it in chapter 3. If you give this flag, PMake won't read the default makefile.

–s This causes PMake to not print commands before they're executed. It is the equivalent of putting an '@' before every command in the makefile.

–t Rather than try to re-create a target, PMake will simply "touch" it so as to make it appear up-to-date. If the target didn't exist before, it will when PMake finishes, but if the target did exist, it will appear to have been updated.

–v This is a mixed-compatibility flag intended to mimic the System V version of Make. It is the same as giving **–B**, and **–V** as well as turning off directory locking. Targets can still be created in parallel, however. This is the mode PMake will enter if it is invoked either as "`smake`" or "`vmake`".

–x This tells PMake it's ok to export jobs to other machines, if they're available. It is used when running in Make mode, as exporting in this mode tends to make things run slower than if the commands were just executed locally.

–B Forces PMake to be as backwards-compatible with Make as possible while still being itself. This includes:

- Executing one shell per shell command
- Expanding anything that looks even vaguely like a variable, with the empty string replacing any variable PMake doesn't know.
- Refusing to allow you to escape a '#' with a backslash.
- Permitting undefined variables on dependency lines and conditionals (see below). Normally this causes PMake to abort.

–C This nullifies any and all compatibility mode flags you may have given or implied up to the time the **–C** is encountered. It is useful mostly in a makefile that you wrote for PMake to avoid bad things happening when someone runs PMake as "`make`" or has things set in the environment that tell it to be compatible. **–C** is *not* placed in the PMAKE environment variable or the .MAKEFLAGS or MFLAGS global variables.

–D *variable*
Allows you to define a variable to have "1" as its value. The variable is a global variable, not a command-line variable. This is useful mostly for people who are used to the C compiler arguments and those using conditionals, which I'll get into in section 4.3

–I *directory*
Tells PMake another place to search for included makefiles. Yet another thing to be explained in chapter 3 (section 3.2, to be precise).

–J *number*
Gives the absolute maximum number of targets to create at once on both local and remote machines.

–L *number*
This specifies the maximum number of targets to create on the local machine at once. This may be 0, though you should be wary of doing this, as PMake may hang until a remote machine becomes available, if one is not available when it is started.

–M This is the flag that provides absolute, complete, full compatibility with Make. It still allows you to use all but a few of the features of PMake, but it is non-parallel. This is the mode PMake enters if you call it "make."

–P When creating targets in parallel, several shells are executing at once, each wanting to write its own two cent's-worth to the screen. This output must be captured by PMake in some way in order to prevent the screen from being filled with garbage even more indecipherable than you usually see. PMake has two ways of doing this, one of which provides for much cleaner output and a clear separation between the output of different jobs, the other of which provides a more immediate response so one can tell what is really happpening. The former is done by notifying you when the creation of a target starts, capturing the output and transferring it to the screen all at once when the job finishes. The latter is done by catching the output of the shell (and its children) and buffering it until an entire line is received, then printing that line preceded by an indication of which job produced the output. Since I prefer this second method, it is the one used by default. The first method will be used if you give the **–P** flag to PMake.

–V As mentioned before, the **–V** flag tells PMake to use Make's style of expanding variables, substituting the empty string for any variable it doesn't know.

–W There are several times when PMake will print a message at you that is only a warning, i.e. it can continue to work in spite of your having done something silly (such as forgotten a leading tab for a shell command). Sometimes you are well aware of silly things you have done and would like PMake to stop bothering you. This flag tells it to shut up about anything non-fatal.

–X This flag causes PMake to not attempt to export any jobs to another machine.

Several flags may follow a single '–'. Those flags that require arguments take them from successive parameters. E.g.

```
pmake -fDnI server.mk DEBUG /chip2/X/server/include
```

will cause PMake to read server.mk as the input makefile, define the variable DEBUG as a global variable and look for included makefiles in the directory /chip2/X/server/include.

2.8. Summary

A makefile is made of four types of lines:

- Dependency lines
- Creation commands
- Variable assignments
- Comments, include statements and conditional directives

A dependency line is a list of one or more targets, an operator (‘:’, ‘::’, or ‘!’), and a list of zero or more sources. Sources may contain wildcards and certain local variables.

A creation command is a regular shell command preceded by a tab. In addition, if the first two characters after the tab (and other whitespace) are a combination of ‘@’ or ‘−’, PMake will cause the command to not be printed (if the character is ‘@’) or errors from it to be ignored (if ‘−’). A blank line, dependency line or variable assignment terminates a creation script. There may be only one creation script for each target with a ‘:’ or ‘!’ operator.

Variables are places to store text. They may be unconditionally assigned-to using the ‘=’ operator, appended-to using the ‘+=’ operator, conditionally (if the variable is undefined) assigned-to with the ‘?=’ operator, and assigned-to with variable expansion with the ‘:=’ operator. The output of a shell command may be assigned to a variable using the ‘!=’ operator. Variables may be expanded (their value inserted) by enclosing their name in parentheses or curly braces, prceeded by a dollar sign. A dollar sign may be escaped with another dollar sign. Variables are not expanded if PMake doesn’t know about them. There are seven local variables: .TARGET, .ALLSRC, .OODATE, .PREFIX, .IMPSRC, .ARCHIVE, and .MEMBER. Four of them (.TARGET, .PREFIX, .ARCHIVE, and .MEMBER) may be used to specify “dynamic sources.” Variables are good. Know them. Love them. Live them.

Debugging of makefiles is best accomplished using the **−n**, **−d m**, and **−p 2** flags.

2.9. Exercises

TBA

3. Short-cuts and Other Nice Things

Based on what I’ve told you so far, you may have gotten the impression that PMake is just a way of storing away commands and making sure you don’t forget to compile something. Good. That’s just what it is. However, the ways I’ve described have been inelegant, at best, and painful, at worst. This chapter contains things that make the writing of makefiles easier and the makefiles themselves shorter and easier to modify (and, occasionally, simpler). In this chapter, I assume you are somewhat more familiar with Sprite (or UNIX, if that’s what you’re using) than I did in chapter 2, just so you’re on your toes. So without further ado...

3.1. Transformation Rules

As you know, a file’s name consists of two parts: a base name, which gives some hint as to the contents of the file, and a suffix, which usually indicates the format of the file. Over the years, as UNIX® has developed, naming conventions, with regard to suffixes, have also developed that have become almost as incontrovertible as Law. E.g. a file ending in .c is assumed to contain C source code; one with a .o suffix is assumed to be a compiled, relocatable object file that may be linked into any program; a file with a .ms suffix is usually a text file to be processed by Troff with the −ms macro package, and so on. One of the best aspects of both Make and PMake comes from their understanding of how the suffix of a file pertains to its contents and their ability to do things with a file based soley on its suffix. This ability comes from something known as a transformation rule. A transformation rule specifies how to change a file with one suffix into a file with another suffix.

A transformation rule looks much like a dependency line, except the target is made of two known suffixes stuck together. Suffixes are made known to PMake by placing them as sources on a dependency line whose target is the special target .SUFFIXES. E.g.

```
.SUFFIXES        : .o .c
.c.o             :
        $(CC) $(CFLAGS) -c $(.IMPSRC)
```

The creation script attached to the target is used to transform a file with the first suffix (in this case, .c) into a file with the second suffix (here, .o). In addition, the target inherits whatever attributes have been applied to the transformation rule. The simple rule given above says that to transform a C source file into an object file, you compile it using cc with the −c flag. This rule is taken straight from the system makefile. Many transformation rules (and suffixes) are defined there, and I refer you to it for more examples (type “pmake

–h” to find out where it is).

There are several things to note about the transformation rule given above:

1) The `.IMPSRC` variable. This variable is set to the “implied source” (the file from which the target is being created; the one with the first suffix), which, in this case, is the .c file.

2) The `CFLAGS` variable. Almost all of the transformation rules in the system makefile are set up using variables that you can alter in your makefile to tailor the rule to your needs. In this case, if you want all your C files to be compiled with the **–g** flag, to provide information for `dbx`, you would set the `CFLAGS` variable to contain `-g` (“`CFLAGS = -g`”) and PMake would take care of the rest.

To give you a quick example, the makefile in 2.3.4 could be changed to this:

```
OBJS            = a.o b.o c.o
program         : $(OBJS)
        $(CC) -o $(.TARGET) $(.ALLSRC)
$(OBJS)         : defs.h
```

The transformation rule I gave above takes the place of the 6 lines[1]

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

Now you may be wondering about the dependency between the `.o` and `.c` files — it's not mentioned anywhere in the new makefile. This is because it isn't needed: one of the effects of applying a transformation rule is the target comes to depend on the implied source. That's why it's called the implied *source*.

For a more detailed example. Say you have a makefile like this:

```
a.out           : a.o b.o
        $(CC) $(.ALLSRC)
```

and a directory set up like this:

```
total 4
-rw-rw-r--  1 deboor         34 Sep  7 00:43 Makefile
-rw-rw-r--  1 deboor        119 Oct  3 19:39 a.c
-rw-rw-r--  1 deboor        201 Sep  7 00:43 a.o
-rw-rw-r--  1 deboor         69 Sep  7 00:43 b.c
```

While just typing “`pmake`” will do the right thing, it's much more informative to type “`pmake -d s`”. This will show you what PMake is up to as it processes the files. In this case, PMake prints the following:

[1] This is also somewhat cleaner, I think, than the dynamic source solution presented in 2.6

```
Suff_FindDeps (a.out)
    using existing source a.o
    applying .o -> .out to "a.o"
Suff_FindDeps (a.o)
    trying a.c...got it
    applying .c -> .o to "a.c"
Suff_FindDeps (b.o)
    trying b.c...got it
    applying .c -> .o to "b.c"
Suff_FindDeps (a.c)
    trying a.y...not there
    trying a.l...not there
    trying a.c,v...not there
    trying a.y,v...not there
    trying a.l,v...not there
Suff_FindDeps (b.c)
    trying b.y...not there
    trying b.l...not there
    trying b.c,v...not there
    trying b.y,v...not there
    trying b.l,v...not there
--- a.o ---
cc  -c a.c
--- b.o ---
cc  -c b.c
--- a.out ---
cc a.o b.o
```

`Suff_FindDeps` is the name of a function in PMake that is called to check for implied sources for a target using transformation rules. The transformations it tries are, naturally enough, limited to the ones that have been defined (a transformation may be defined multiple times, by the way, but only the most recent one will be used). You will notice, however, that there is a definite order to the suffixes that are tried. This order is set by the relative positions of the suffixes on the `.SUFFIXES` line — the earlier a suffix appears, the earlier it is checked as the source of a transformation. Once a suffix has been defined, the only way to change its position in the pecking order is to remove all the suffixes (by having a `.SUFFIXES` dependency line with no sources) and redefine them in the order you want. (Previously-defined transformation rules will be automatically redefined as the suffixes they involve are re-entered.)

Another way to affect the search order is to make the dependency explicit. In the above example, `a.out` depends on `a.o` and `b.o`. Since a transformation exists from `.o` to `.out`, PMake uses that, as indicated by the "`using existing source a.o`" message.

The search for a transformation starts from the suffix of the target and continues through all the defined transformations, in the order dictated by the suffix ranking, until an existing file with the same base (the target name minus the suffix and any leading directories) is found. At that point, one or more transformation rules will have been found to change the one existing file into the target.

For example, ignoring what's in the system makefile for now, say you have a makefile like this:

```
.SUFFIXES       : .out .o .c .y .l
.l.c            :
        lex $(.IMPSRC)
        mv lex.yy.c $(.TARGET)
.y.c            :
        yacc $(.IMPSRC)
        mv y.tab.c $(.TARGET)
.c.o            :
        cc -c $(.IMPSRC)
.o.out          :
        cc -o $(.TARGET) $(.IMPSRC)
```

and the single file `jive.l`. If you were to type "`pmake -rd ms jive.out`," you would get the following output for `jive.out`:

```
Suff_FindDeps (jive.out)
    trying jive.o...not there
    trying jive.c...not there
    trying jive.y...not there
    trying jive.l...got it
    applying .l -> .c to "jive.l"
    applying .c -> .o to "jive.c"
    applying .o -> .out to "jive.o"
```

and this is why: PMake starts with the target `jive.out`, figures out its suffix (`.out`) and looks for things it can transform to a `.out` file. In this case, it only finds `.o`, so it looks for the file `jive.o`. It fails to find it, so it looks for transformations into a `.o` file. Again it has only one choice: `.c`. So it looks for `jive.c` and, as you know, fails to find it. At this point it has two choices: it can create the `.c` file from either a `.y` file or a `.l` file. Since `.y` came first on the `.SUFFIXES` line, it checks for `jive.y` first, but can't find it, so it looks for `jive.l` and, lo and behold, there it is. At this point, it has defined a transformation path as follows: `.l` → `.c` → `.o` → `.out` and applies the transformation rules accordingly. For completeness, and to give you a better idea of what PMake actually did with this three-step transformation, this is what PMake printed for the rest of the process:

```
Suff_FindDeps (jive.o)
    using existing source jive.c
    applying .c -> .o to "jive.c"
Suff_FindDeps (jive.c)
    using existing source jive.l
    applying .l -> .c to "jive.l"
Suff_FindDeps (jive.l)
Examining jive.l...modified 17:16:01 Oct 4, 1987...up-to-date
Examining jive.c...non-existent...out-of-date
--- jive.c ---
lex jive.l
. . . meaningless lex output deleted . . .
mv lex.yy.c jive.c
Examining jive.o...non-existent...out-of-date
--- jive.o ---
cc -c jive.c
Examining jive.out...non-existent...out-of-date
--- jive.out ---
cc -o jive.out jive.o
```

One final question remains: what does PMake do with targets that have no known suffix? PMake simply pretends it actually has a known suffix and searches for transformations accordingly. The suffix it chooses is the source for the `.NULL` target mentioned later. In the system makefile, `.out` is chosen as the "null suffix'' because most people use PMake to create programs. You are, however, free and welcome to change it to a suffix of your own choosing. The null suffix is ignored, however, when PMake is in compatibility mode (see chapter 4).

3.2. Including Other Makefiles

Just as for programs, it is often useful to extract certain parts of a makefile into another file and just include it in other makefiles somehow. Many compilers allow you say something like

```
#include "defs.h"
```

to include the contents of `defs.h` in the source file. PMake allows you to do the same thing for makefiles, with the added ability to use variables in the filenames. An include directive in a makefile looks either like this:

```
#include <file>
```

or this

```
#include "file"
```

The difference between the two is where PMake searches for the file: the first way, PMake will look for the file only in the system makefile directory (to find out what that directory is, give PMake the **–h** flag). For files in double-quotes, the search is more complex:

1) The directory of the makefile that's including the file.
2) The current directory (the one in which you invoked PMake).
3) The directories given by you using **–I** flags, in the order in which you gave them.
4) Directories given by `.PATH` dependency lines (see chapter 4).
5) The system makefile directory.

in that order.

You are free to use PMake variables in the filename—PMake will expand them before searching for the file. You must specify the searching method with either angle brackets or double-quotes *outside* of a variable expansion. I.e. the following

```
SYSTEM   = <command.mk>

#include $(SYSTEM)
```

won't work.

3.3. Saving Commands

There may come a time when you will want to save certain commands to be executed when everything else is done. For instance: you're making several different libraries at one time and you want to create the members in parallel. Problem is, `ranlib` is another one of those programs that can't be run more than once in the same directory at the same time (each one creates a file called `__.SYMDEF` into which it stuffs information for the linker to use. Two of them running at once will overwrite each other's file and the result will be garbage for both parties). You might want a way to save the ranlib commands til the end so they can be run one after the other, thus keeping them from trashing each other's file. PMake allows you to do this by inserting an ellipsis ("...") as a command between commands to be run at once and those to be run later.

So for the `ranlib` case above, you might do this:

```
lib1.a          : $(LIB1OBJS)
        rm -f $(.TARGET)
        ar cr $(.TARGET) $(.ALLSRC)
        ...
        ranlib $(.TARGET)

lib2.a          : $(LIB2OBJS)
        rm -f $(.TARGET)
        ar cr $(.TARGET) $(.ALLSRC)
        ...
        ranlib $(.TARGET)
```

This would save both

```
ranlib $(.TARGET)
```

commands until the end, when they would run one after the other (using the correct value for the `.TARGET` variable, of course).

Commands saved in this manner are only executed if PMake manages to re-create everything without an error.

3.4. Target Attributes

PMake allows you to give attributes to targets by means of special sources. Like everything else PMake uses, these sources begin with a period and are made up of all upper-case letters. There are various reasons for using them, and I will try to give examples for most of them. Others you'll have to find uses for yourself. Think of it as "an exercise for the reader." By placing one (or more) of these as a source on a dependency line, you are "marking the target(s) with that attribute." That's just the way I phrase it, so you know.

Any attributes given as sources for a transformation rule are applied to the target of the transformation rule when the rule is applied.

.DONTCARE
: If a target is marked with this attribute and PMake can't figure out how to create it, it will ignore this fact and assume the file isn't really needed or actually exists and PMake just can't find it. This may prove wrong, but the error will be noted later on, not when PMake tries to create the target so marked. This attribute also prevents PMake from attempting to touch the target if it is given the **–t** flag.

.EXEC
: This attribute causes its shell script to be executed while having no effect on targets that depend on it. This makes the target into a sort of subroutine. An example. Say you have some LISP files that need to be compiled and loaded into a LISP process. To do this, you echo LISP commands into a file and execute a LISP with this file as its input when everything's done. Say also that you have to load other files from another system before you can compile your files and further, that you don't want to go through the loading and dumping unless one of *your* files has changed. Your makefile might look a little bit like this (remember, this is an educational example, and don't worry about the `COMPILE` rule, all will soon become clear, grasshopper):

```
system          : init a.fasl b.fasl c.fasl
        for i in $(.ALLSRC);
        do
                echo -n '(load "' >> input
                echo -n ${i} >> input
                echo '")' >> input
        done
        echo '(dump "$(.TARGET)")' >> input
        lisp < input

a.fasl          : a.l init COMPILE
b.fasl          : b.l init COMPILE
c.fasl          : c.l init COMPILE
COMPILE         : .USE
        echo '(compile "$(.ALLSRC)")' >> input
init            : .EXEC
        echo '(load-system)' > input
```

`.EXEC` sources, don't appear in the local variables of targets that depend on them (nor are they touched if PMake is given the **−t** flag). Note that all the rules, not just that for `system`, include `init` as a source. This is because none of the other targets can be made until `init` has been made, thus they depend on it.

.EXPORT This is used to mark those targets whose creation should be sent to another machine if at all possible. This may be used by some exportation schemes if the exportation is expensive. You should ask your system administrator if it is necessary.

.EXPORTSAME

Tells the export system that the job should be exported to a machine of the same architecture as the current one. Certain operations (e.g. running text through `nroff`) can be performed the same on any architecture (CPU and operating system type), while others (e.g. compiling a program with `cc`) must be performed on a machine with the same architecture. Not all export systems will support this attribute.

.IGNORE Giving a target the `.IGNORE` attribute causes PMake to ignore errors from any of the target's commands, as if they all had '−' before them.

.INVISIBLE This allows you to specify one target as a source for another without the one affecting the other's local variables. Useful if, say, you have a makefile that creates two programs, one of which is used to create the other, so it must exist before the other is created. You could say

```
prog1           : $(PROG1OBJS) prog2 MAKEINSTALL
prog2           : $(PROG2OBJS) .INVISIBLE MAKEINSTALL
```

where `MAKEINSTALL` is some complex .USE rule (see below) that depends on the `.ALLSRC` variable containing the right things. Without the `.INVISIBLE` attribute for `prog2`, the `MAKEINSTALL` rule couldn't be applied. This is not as useful as it should be, and the semantics may change (or the whole thing go away) in the not-too-distant future.

.JOIN This is another way to avoid performing some operations in parallel while permitting everything else to be done so. Specifically it forces the target's shell script to be executed only if one or more of the sources was out-of-date. In addition, the target's name, in both its `.TARGET` variable and all the local variables of any target that depends on it, is replaced by the value of its `.ALLSRC` variable. As an example, suppose you have a program that has four libraries that compile in the same directory along with, and at the same time as, the program. You again have the problem with `ranlib` that I mentioned earlier, only this time it's more severe: you can't just put the ranlib off to the end since the program will need those libraries before it can be re-created. You can do something like this:

```
program         : $(OBJS) libraries
        cc -o $(.TARGET) $(.ALLSRC)

libraries       : lib1.a lib2.a lib3.a lib4.a .JOIN
        ranlib $(.OODATE)
```

In this case, PMake will re-create the `$(OBJS)` as necessary, along with `lib1.a`, `lib2.a`, `lib3.a` and `lib4.a`. It will then execute `ranlib` on any library that was changed and set `program`'s `.ALLSRC` variable to contain what's in `$(OBJS)` followed by "`lib1.a lib2.a lib3.a lib4.a`." In case you're wondering, it's called `.JOIN` because it joins together different threads of the "input graph" at the target marked with the attribute. Another aspect of the .JOIN attribute is it keeps the target from being created if the **−t** flag was given.

.MAKE The `.MAKE` attribute marks its target as being a recursive invocation of PMake. This forces PMake to execute the script associated with the target (if it's out-of-date) even if you gave the **−n** or **−t** flag. By doing this, you can start at the top of a system and type

```
pmake -n
```

and have it descend the directory tree (if your makefiles are set up correctly), printing what it would have executed if you hadn't included the **−n** flag.

.NOEXPORT

If possible, PMake will attempt to export the creation of all targets to another machine (this depends on how PMake was configured). Sometimes, the creation is so simple, it is pointless to send it to another machine. If you give the target the `.NOEXPORT` attribute, it will be run locally, even if you've given PMake the **−L 0** flag.

.NOTMAIN Normally, if you do not specify a target to make in any other way, PMake will take the first target on the first dependency line of a makefile as the target to create. That target is known as the "Main Target" and is labeled as such if you print the dependencies out using the **−p** flag. Giving a target this attribute tells PMake that the target is definitely *not* the Main Target. This allows you to place targets in an included makefile and have PMake create something else by default.

.PRECIOUS When PMake is interrupted (you type control-C at the keyboard), it will attempt to clean up after itself by removing any half-made targets. If a target has the `.PRECIOUS` attribute, however, PMake will leave it alone. An additional side effect of the '::' operator is to mark the targets as `.PRECIOUS`.

.SILENT Marking a target with this attribute keeps its commands from being printed when they're executed, just as if they had an '@' in front of them.

.USE By giving a target this attribute, you turn it into PMake's equivalent of a macro. When the target is used as a source for another target, the other target acquires the commands, sources and attributes (except `.USE`) of the source. If the target already has commands, the `.USE` target's commands are added to the end. If more than one .USE-marked source is given to a target, the rules are applied sequentially.

The typical .USE rule (as I call them) will use the sources of the target to which it is applied (as stored in the `.ALLSRC` variable for the target) as its "arguments," if you will. For example, you probably noticed that the commands for creating `lib1.a` and `lib2.a` in the example in section 3.3 were exactly the same. You can use the `.USE` attribute to eliminate the repetition, like so:

```
lib1.a          : $(LIB1OBJS) MAKELIB
lib2.a          : $(LIB2OBJS) MAKELIB

MAKELIB         : .USE
        rm -f $(.TARGET)
        ar cr $(.TARGET) $(.ALLSRC)
        ...
        ranlib $(.TARGET)
```

Several system makefiles (not to be confused with The System Makefile) make use of these .USE rules to make your life easier (they're in the default, system makefile directory...take a look). Note that the .USE rule source itself (`MAKELIB`) does not appear in any of the targets's local variables. There is no limit to the number of times I could use the `MAKELIB` rule. If there were more libraries, I could continue with "`lib3.a : $(LIB3OBJS) MAKELIB`" and so on and so forth.

3.5. Special Targets

As there were in Make, so there are certain targets that have special meaning to PMake. When you use one on a dependency line, it is the only target that may appear on the left-hand-side of the operator. As for the attributes and variables, all the special targets begin with a period and consist of upper-case letters only. I won't describe them all in detail because some of them are rather complex and I'll describe them in more detail than you'll want in chapter 4. The targets are as follows:

.BEGIN Any commands attached to this target are executed before anything else is done. You can use it for any initialization that needs doing.

.DEFAULT
This is sort of a .USE rule for any target (that was used only as a source) that PMake can't figure out any other way to create. It's only "sort of" a .USE rule because only the shell script attached to the `.DEFAULT` target is used. The `.IMPSRC` variable of a target that inherits `.DEFAULT`'s commands is set to the target's own name.

.END This serves a function similar to `.BEGIN`, in that commands attached to it are executed once everything has been re-created (so long as no errors occurred). It also serves the extra function of being a place on which PMake can hang commands you put off to the end. Thus the script for this target will be executed before any of the commands you save with the "`...`".

.EXPORT The sources for this target are passed to the exportation system compiled into PMake. Some systems will use these sources to configure themselves. You should ask your system administrator about this.

.IGNORE This target marks each of its sources with the `.IGNORE` attribute. If you don't give it any sources, then it is like giving the **–i** flag when you invoke PMake — errors are ignored for all commands.

.INCLUDES
The sources for this target are taken to be suffixes that indicate a file that can be included in a program source file. The suffix must have already been declared with `.SUFFIXES` (see below). Any suffix so marked will have the directories on its search path (see `.PATH`, below) placed in the `.INCLUDES` variable, each preceded by a **–I** flag. This variable can then be used as an argument for the compiler in the normal fashion. The `.h` suffix is already marked in this way in the system makefile. E.g. if you have

```
.SUFFIXES       : .bitmap
.PATH.bitmap    : /usr/local/X/lib/bitmaps
.INCLUDES       : .bitmap
```

PMake will place "`-I/usr/local/X/lib/bitmaps`" in the `.INCLUDES` variable and you can then say

```
cc $(.INCLUDES) -c xprogram.c
```

(Note: the `.INCLUDES` variable is not actually filled in until the entire makefile has been read.)

.INTERRUPT
: When PMake is interrupted, it will execute the commands in the script for this target, if it exists.

.LIBS
: This does for libraries what `.INCLUDES` does for include files, except the flag used is **–L**, as required by those linkers that allow you to tell them where to find libraries. The variable used is `.LIBS`. Be forewarned that PMake may not have been compiled to do this if the linker on your system doesn't accept the **–L** flag, though the `.LIBS` variable will always be defined once the makefile has been read.

.MAIN
: If you didn't give a target (or targets) to create when you invoked PMake, it will take the sources of this target as the targets to create.

.MAKEFLAGS
: This target provides a way for you to always specify flags for PMake when the makefile is used. The flags are just as they would be typed to the shell (except you can't use shell variables unless they're in the environment), though the **–f** and **–r** flags have no effect.

.NULL
: This allows you to specify what suffix PMake should pretend a file has if, in fact, it has no known suffix. Only one suffix may be so designated. The last source on the dependency line is the suffix that is used (you should, however, only give one suffix. . .).

.PATH
: If you give sources for this target, PMake will take them as directories in which to search for files it cannot find in the current directory. If you give no sources, it will clear out any directories added to the search path before. Since the effects of this all get very complex, I'll leave it til chapter four to give you a complete explanation.

.PATH*suffix*
: This does a similar thing to `.PATH`, but it does it only for files with the given suffix. The suffix must have been defined already. Look at **Search Paths** (section 4.1) for more information.

.PRECIOUS
: Similar to `.IGNORE`, this gives the `.PRECIOUS` attribute to each source on the dependency line, unless there are no sources, in which case the `.PRECIOUS` attribute is given to every target in the file.

.RECURSIVE
: This target applies the `.MAKE` attribute to all its sources. It does nothing if you don't give it any sources.

.SHELL
: PMake is not constrained to only using the Bourne shell to execute the commands you put in the makefile. You can tell it some other shell to use with this target. Check out **A Shell is a Shell is a Shell** (section 4.4) for more information.

.SILENT
: When you use `.SILENT` as a target, it applies the `.SILENT` attribute to each of its sources. If there are no sources on the dependency line, then it is as if you gave PMake the **–s** flag and no commands will be echoed.

.SUFFIXES
: This is used to give new file suffixes for PMake to handle. Each source is a suffix PMake should recognize. If you give a `.SUFFIXES` dependency line with no sources, PMake will forget about all the suffixes it knew (this also nukes the null suffix). For those targets that need to have suffixes defined, this is how you do it.

In addition to these targets, a line of the form

attribute **:** *sources*

applies the *attribute* to all the targets listed as *sources*.

3.6. Modifying Variable Expansion

Variables need not always be expanded verbatim. PMake defines several modifiers that may be applied to a variable's value before it is expanded. You apply a modifier by placing it after the variable name with a colon between the two, like so:

`${`*VARIABLE*`:`*modifier*`}`

Each modifier is a single character followed by something specific to the modifier itself. You may apply as many modifiers as you want — each one is applied to the result of the previous and is separated from the previous by another colon.

There are seven ways to modify a variable's expansion, most of which come from the C shell variable modification characters:

M*pattern*
: This is used to select only those words (a word is a series of characters that are neither spaces nor tabs) that match the given *pattern*. The pattern is a wildcard pattern like that used by the shell, where `*` means 0 or more characters of any sort; `?` is any single character; `[abcd]` matches any single character that is either 'a', 'b', 'c' or 'd' (there may be any number of characters between the brackets); `[0-9]` matches any single character that is between '0' and '9' (i.e. any digit. This form may be freely mixed with the other bracket form), and '\' is used to escape any of the characters '*', '?', '[' or ':', leaving them as regular characters to match themselves in a word. For example, the system makefile `<makedepend.mk>` uses "`$(CFLAGS:M-[ID]*)`" to extract all the `-I` and `-D` flags that would be passed to the C compiler. This allows it to properly locate include files and generate the correct dependencies.

N*pattern*
: This is identical to `:M` except it substitutes all words that don't match the given pattern.

S/*search-string*/*replacement-string*/[g]
: Causes the first occurrence of *search-string* in the variable to be replaced by *replacement-string*, unless the `g` flag is given at the end, in which case all occurences of the string are replaced. The substitution is performed on each word in the variable in turn. If *search-string* begins with a `^`, the string must match starting at the beginning of the word. If *search-string* ends with a `$`, the string must match to the end of the word (these two may be combined to force an exact match). If a backslash preceeds these two characters, however, they lose their special meaning. Variable expansion also occurs in the normal fashion inside both the *search-string* and the *replacement-string*, **except** that a backslash is used to prevent the expansion of a `$`, not another dollar sign, as is usual. Note that *search-string* is just a string, not a pattern, so none of the usual regular-expression/wildcard characters have any special meaning save `^` and `$`. In the replacement string, the `&` character is replaced by the *search-string* unless it is preceded by a backslash. You are allowed to use any character except colon or exclamation point to separate the two strings. This so-called delimiter character may be placed in either string by preceeding it with a backslash.

T
: Replaces each word in the variable expansion by its last component (its "tail"). For example, given

```
OBJS = ../lib/a.o b /usr/lib/libm.a
TAILS = $(OBJS:T)
```

the variable `TAILS` would expand to "`a.o b libm.a`."

H
: This is similar to `:T`, except that every word is replaced by everything but the tail (the "head"). Using the same definition of `OBJS`, the string "`$(OBJS:H)`" would expand to "`../lib /usr/lib`." Note that the final slash on the heads is removed and anything without a head is replaced by the empty string.

E
: `:E` replaces each word by its suffix ("extension"). So "`$(OBJS:E)`" would give you "`.o .a`."

R This replaces each word by everything but the suffix (the ‘‘root’’ of the word). ‘‘`$(OBJS:R)`’’ expands to ‘‘ `../lib/a b /usr/lib/libm`.’’

In addition, the System V style of substitution is also supported. This looks like:

`$(`*VARIABLE*`:`*search-string=replacement*`)`

It must be the last modifier in the chain. The search is anchored at the end of each word, so only suffixes or whole words may be replaced.

3.7. More on Debugging

3.8. More Exercises

(3.1) You’ve got a set programs, each of which is created from its own assembly-language source file (suffix `.asm`). Each program can be assembled into two versions, one with error-checking code assembled in and one without. You could assemble them into files with different suffixes (`.eobj` and `.obj`, for instance), but your linker only understands files that end in `.obj`. To top it all off, the final executables *must* have the suffix `.exe`. How can you still use transformation rules to make your life easier (Hint: assume the error-checking versions have `ec` tacked onto their prefix)?

(3.2) Assume, for a moment or two, you want to perform a sort of ‘‘indirection’’ by placing the name of a variable into another one, then you want to get the value of the first by expanding the second somehow. Unfortunately, PMake doesn’t allow constructs like

```
$($(FOO))
```

What do you do? Hint: no further variable expansion is performed after modifiers are applied, thus if you cause a $ to occur in the expansion, that’s what will be in the result.

4. PMake for Gods

This chapter is devoted to those facilities in PMake that allow you to do a great deal in a makefile with very little work, as well as do some things you couldn’t do in Make without a great deal of work (and perhaps the use of other programs). The problem with these features, is they must be handled with care, or you will end up with a mess.

Once more, I assume a greater familiarity with UNIX or Sprite than I did in the previous two chapters.

4.1. Search Paths

PMake supports the dispersal of files into multiple directories by allowing you to specify places to look for sources with `.PATH` targets in the makefile. The directories you give as sources for these targets make up a ‘‘search path.’’ Only those files used exclusively as sources are actually sought on a search path, the assumption being that anything listed as a target in the makefile can be created by the makefile and thus should be in the current directory.

There are two types of search paths in PMake: one is used for all types of files (including included makefiles) and is specified with a plain `.PATH` target (e.g. ‘‘`.PATH : RCS`’’), while the other is specific to a certain type of file, as indicated by the file’s suffix. A specific search path is indicated by immediately following the `.PATH` with the suffix of the file. For instance

```
.PATH.h          : /sprite/lib/include /sprite/att/lib/include
```

would tell PMake to look in the directories `/sprite/lib/include` and `/sprite/att/lib/include` for any files whose suffix is `.h`.

The current directory is always consulted first to see if a file exists. Only if it cannot be found there are the directories in the specific search path, followed by those in the general search path, consulted.

A search path is also used when expanding wildcard characters. If the pattern has a recognizable suffix on it, the path for that suffix will be used for the expansion. Otherwise the default search path is employed.

When a file is found in some directory other than the current one, all local variables that would have contained the target's name (`.ALLSRC`, and `.IMPSRC`) will instead contain the path to the file, as found by PMake. Thus if you have a file `../lib/mumble.c` and a makefile

```
.PATH.c          : ../lib
mumble           : mumble.c
        $(CC) -o $(.TARGET) $(.ALLSRC)
```

the command executed to create `mumble` would be "`cc -o mumble ../lib/mumble.c`." (As an aside, the command in this case isn't strictly necessary, since it will be found using transformation rules if it isn't given. This is because `.out` is the null suffix by default and a transformation exists from `.c` to `.out`. Just thought I'd throw that in.)

If a file exists in two directories on the same search path, the file in the first directory on the path will be the one PMake uses. So if you have a large system spread over many directories, it would behoove you to follow a naming convention that avoids such conflicts.

Something you should know about the way search paths are implemented is that each directory is read, and its contents cached, exactly once — when it is first encountered — so any changes to the directories while PMake is running will not be noted when searching for implicit sources, nor will they be found when PMake attempts to discover when the file was last modified, unless the file was created in the current directory. While people have suggested that PMake should read the directories each time, my experience suggests that the caching seldom causes problems. In addition, not caching the directories slows things down enormously because of PMake's attempts to apply transformation rules through non-existent files — the number of extra file-system searches is truly staggering, especially if many files without suffixes are used and the null suffix isn't changed from `.out`.

4.2. Archives and Libraries

UNIX and Sprite allow you to merge files into an archive using the `ar` command. Further, if the files are relocatable object files, you can run `ranlib` on the archive and get yourself a library that you can link into any program you want. The main problem with archives is they double the space you need to store the archived files, since there's one copy in the archive and one copy out by itself. The problem with libraries is you usually think of them as `-lm` rather than `/usr/lib/libm.a` and the linker thinks they're out-of-date if you so much as look at them.

PMake solves the problem with archives by allowing you to tell it to examine the files in the archives (so you can remove the individual files without having to regenerate them later). To handle the problem with libraries, PMake adds an additional way of deciding if a library is out-of-date:

- If the table of contents is older than the library, or is missing, the library is out-of-date.

A library is any target that looks like "`-l`name" or that ends in a suffix that was marked as a library using the `.LIBS` target. `.a` is so marked in the system makefile.

Members of an archive are specified as "*archive*(*member*[*member*...])". Thus "'`libdix.a(window.o)`" specifies the file `window.o` in the archive `libdix.a`. You may also use wildcards to specify the members of the archive. Just remember that most the wildcard characters will only find *existing* files.

A file that is a member of an archive is treated specially. If the file doesn't exist, but it is in the archive, the modification time recorded in the archive is used for the file when determining if the file is out-of-date. When figuring out how to make an archived member target (not the file itself, but the file in the archive — the *archive*(*member*) target), special care is taken with the transformation rules, as follows:

- *archive*(*member*) is made to depend on *member*.
- The transformation from the *member*'s suffix to the *archive*'s suffix is applied to the *archive*(*member*) target.
- The *archive*(*member*)'s `.TARGET` variable is set to the name of the *member* if *member* is actually a target, or the path to the member file if *member* is only a source.

- The `.ARCHIVE` variable for the *archive*(*member*) target is set to the name of the *archive*.
- The `.MEMBER` variable is set to the actual string inside the parentheses. In most cases, this will be the same as the `.TARGET` variable.
- The *archive*(*member*)'s place in the local variables of the targets that depend on it is taken by the value of its `.TARGET` variable.

Thus, a program library could be created with the following makefile:

```
.o.a            :
        ...
        rm -f $(.TARGET:T)
OBJS            = obj1.o obj2.o obj3.o
libprog.a       : libprog.a($(OBJS))
        ar cru $(.TARGET) $(.OODATE)
        ranlib $(.TARGET)
```

This will cause the three object files to be compiled (if the corresponding source files were modified after the object file or, if that doesn't exist, the archived object file), the out-of-date ones archived in `libprog.a`, a table of contents placed in the archive and the newly-archived object files to be removed.

All this is used in the `makelib.mk` system makefile to create a single library with ease. This makefile looks like this:

```
#
# Rules for making libraries. The object files that make up the library are
# removed once they are archived.
#
# To make several libararies in parallel, you should define the variable
# "many_libraries". This will serialize the invocations of ranlib.
#
# To use, do something like this:
#
# OBJECTS = <files in the library>
#
# fish.a: fish.a($(OBJECTS)) MAKELIB
#
#

#ifndef _MAKELIB_MK
_MAKELIB_MK     =

#include   <po.mk>

.po.a .o.a :
     ...
     rm -f $(.MEMBER)

ARFLAGS         ?= crl

#
# Re-archive the out-of-date members and recreate the library's table of
# contents using ranlib. If many_libraries is defined, put the ranlib off
# til the end so many libraries can be made at once.
#
MAKELIB         : .USE .PRECIOUS
     ar $(ARFLAGS) $(.TARGET) $(.OODATE)
#ifndef no_ranlib
# ifdef many_libraries
     ...
# endif many_libraries
     ranlib $(.TARGET)
#endif no_ranlib

#endif _MAKELIB_MK
```

4.3. On the Condition...

Like the C compiler before it, PMake allows you to configure the makefile, based on the current environment, using conditional statements. A conditional looks like this:

> `#if` *boolean expression*
> *lines*
> `#elif` *another boolean expression*
> *more lines*
> `#else`
> *still more lines*
> `#endif`

They may be nested to a maximum depth of 30 and may occur anywhere (except in a comment, of course).

The "`#`" must the very first character on the line.

Each *boolean expression* is made up of terms that look like function calls, the standard C boolean operators `&&`, `||`, and `!`, and the standard relational operators ==, `!=`, >, >=, <, and <=, with == and `!=` being overloaded to allow string comparisons as well. `&&` represents logical AND; `||` is logical OR and `!` is logical NOT. The arithmetic and string operators take precedence over all three of these operators, while NOT takes precedence over AND, which takes precedence over OR. This precedence may be overridden with parentheses, and an expression may be parenthesized to your heart's content. Each term looks like a call on one of four functions:

make
: The syntax is `make(`*target*`)` where *target* is a target in the makefile. This is true if the given target was specified on the command line, or as the source for a `.MAIN` target (note that the sources for `.MAIN` are only used if no targets were given on the command line).

defined
: The syntax is `defined(`*variable*`)` and is true if *variable* is defined. Certain variables are defined in the system makefile that identify the system on which PMake is being run.

exists
: The syntax is `exists(`*file*`)` and is true if the file can be found on the global search path (i.e. that defined by `.PATH` targets, not by `.PATH`*suffix* targets).

empty
: This syntax is much like the others, except the string inside the parentheses is of the same form as you would put between parentheses when expanding a variable, complete with modifiers and everything. The function returns true if the resulting string is empty (NOTE: an undefined variable in this context will cause at the very least a warning message about a malformed conditional, and at the worst will cause the process to stop once it has read the makefile. If you want to check for a variable being defined or empty, use the expression "`!defined(`*var*`) || empty(`*var*`)`" as the definition of `||` will prevent the `empty()` from being evaluated and causing an error, if the variable is undefined). This can be used to see if a variable contains a given word, for example:

```
#if !empty(var:Mword)
```

The arithmetic and string operators may only be used to test the value of a variable. The lefthand side must contain the variable expansion, while the righthand side contains either a string, enclosed in double-quotes, or a number. The standard C numeric conventions (except for specifying an octal number) apply to both sides. E.g.

```
#if $(OS) == 4.3

#if $(MACHINE) == "sun3"

#if $(LOAD_ADDR) < 0xc000
```

are all valid conditionals. In addition, the numeric value of a variable can be tested as a boolean as follows:

```
#if $(LOAD)
```

would see if `LOAD` contains a non-zero value and

```
#if !$(LOAD)
```

would test if `LOAD` contains a zero value.

In addition to the bare "`#if`," there are other forms that apply one of the first two functions to each term. They are as follows:

`ifdef`	defined
`ifndef`	!defined
`ifmake`	make
`ifnmake`	!make

There are also the "else if" forms: `elif`, `elifdef`, `elifndef`, `elifmake`, and `elifnmake`.

For instance, if you wish to create two versions of a program, one of which is optimized (the production version) and the other of which is for debugging (has symbols for dbx), you have two choices: you can create two makefiles, one of which uses the `-g` flag for the compilation, while the other uses the `-O` flag, or you can use another target (call it `debug`) to create the debug version. The construct below will take care of this for you. I have also made it so defining the variable `DEBUG` (say with `pmake -D DEBUG`) will also cause the debug version to be made.

```
#if defined(DEBUG) || make(debug)
CFLAGS       += -g
#else
CFLAGS       += -O
#endif
```

There are, of course, problems with this approach. The most glaring annoyance is that if you want to go from making a debug version to making a production version, you have to remove all the object files, or you will get some optimized and some debug versions in the same program. Another annoyance is you have to be careful not to make two targets that "conflict" because of some conditionals in the makefile. For instance

```
#if make(print)
FORMATTER    = ditroff -Plaser_printer
#endif
#if make(draft)
FORMATTER    = nroff -Pdot_matrix_printer
#endif
```

would wreak havok if you tried "`pmake draft print`" since you would use the same formatter for each target. As I said, this all gets somewhat complicated.

4.4. A Shell is a Shell is a Shell

In normal operation, the Bourne Shell (better known as "`sh`") is used to execute the commands to re-create targets. PMake also allows you to specify a different shell for it to use when executing these commands. There are several things PMake must know about the shell you wish to use. These things are specified as the sources for the `.SHELL` target by keyword, as follows:

path=*path*
: PMake needs to know where the shell actually resides, so it can execute it. If you specify this and nothing else, PMake will use the last component of the path and look in its table of the shells it knows and use the specification it finds, if any. Use this if you just want to use a different version of the Bourne or C Shell (yes, PMake knows how to use the C Shell too).

name=*name*
: This is the name by which the shell is to be known. It is a single word and, if no other keywords are specified (other than **path**), it is the name by which PMake attempts to find a specification for it (as mentioned above). You can use this if you would just rather use the C Shell than the Bourne Shell ("`.SHELL: name=csh`" will do it).

quiet=*echo-off command*
: As mentioned before, PMake actually controls whether commands are printed by introducing commands into the shell's input stream. This keyword, and the next two, control what those commands are. The **quiet** keyword is the command used to turn echoing off. Once it is turned off, echoing is expected to remain off until the echo-on command is given.

echo=*echo-on command*
: The command PMake should give to turn echoing back on again.

filter=*printed echo-off command*
: Many shells will echo the echo-off command when it is given. This keyword tells PMake in what format the shell actually prints the echo-off command. Wherever PMake sees this string in the shell's output, it will delete it and any following whitespace, up to and including the next newline. See the

example at the end of this section for more details.

echoFlag=*flag to turn echoing on*

Unless a target has been marked `.SILENT`, PMake wants to start the shell running with echoing on. To do this, it passes this flag to the shell as one of its arguments. If either this or the next flag begins with a '−', the flags will be passed to the shell as separate arguments. Otherwise, the two will be concatenated (if they are used at the same time, of course).

errFlag=*flag to turn error checking on*

Likewise, unless a target is marked `.IGNORE`, PMake wishes error-checking to be on from the very start. To this end, it will pass this flag to the shell as an argument. The same rules for an initial '−' apply as for the **echoFlag**.

check=*command to turn error checking on*

Just as for echo-control, error-control is achieved by inserting commands into the shell's input stream. This is the command to make the shell check for errors. It also serves another purpose if the shell doesn't have error-control as commands, but I'll get into that in a minute. Again, once error checking has been turned on, it is expected to remain on until it is turned off again.

ignore=*command to turn error checking off*

This is the command PMake uses to turn error checking off. It has another use if the shell doesn't do error-control, but I'll tell you about that. . . now.

hasErrCtl=*yes or no*

This takes a value that is either **yes** or **no**. Now you might think that the existence of the **check** and **ignore** keywords would be enough to tell PMake if the shell can do error-control, but you'd be wrong. If **hasErrCtl** is **yes**, PMake uses the check and ignore commands in a straight-forward manner. If this is **no**, however, their use is rather different. In this case, the check command is used as a template, in which the string **%s** is replaced by the command that's about to be executed, to produce a command for the shell that will echo the command to be executed. The ignore command is also used as a template, again with **%s** replaced by the command to be executed, to produce a command that will execute the command to be executed and ignore any error it returns. When these strings are used as templates, you must provide newline(s) ("`\n`") in the appropriate place(s).

The strings that follow these keywords may be enclosed in single or double quotes (the quotes will be stripped off) and may contain the usual C backslash-characters (\n is newline, \r is return, \b is backspace, \' escapes a single-quote inside single-quotes, \" escapes a double-quote inside double-quotes). Now for an example.

This is actually the contents of the `<shx.mk>` system makefile, and causes PMake to use the Bourne Shell in such a way that each command is printed as it is executed. That is, if more than one command is given on a line, each will be printed separately. Similarly, each time the body of a loop is executed, the commands within that loop will be printed, etc. The specification runs like this:

```
#
# This is a shell specification to have the bourne shell echo
# the commands just before executing them, rather than when it reads
# them. Useful if you want to see how variables are being expanded, etc.
#
.SHELL   : path=/bin/sh \
    quiet="set -" \
    echo="set -x" \
    filter="+ set - " \
    echoFlag=x \
    errFlag=e \
    hasErrCtl=yes \
    check="set -e" \
    ignore="set +e"
```

It tells PMake the following:

- The shell is located in the file `/bin/sh`. It need not tell PMake that the name of the shell is `sh` as PMake can figure that out for itself (it's the last component of the path).
- The command to stop echoing is `set -`.
- The command to start echoing is `set -x`.
- When the echo off command is executed, the shell will print `+ set -` (The '+' comes from using the `-x` flag (rather than the `-v` flag PMake usually uses)). PMake will remove all occurences of this string from the output, so you don't notice extra commands you didn't put there.
- The flag the Bourne Shell will take to start echoing in this way is the `-x` flag. The Bourne Shell will only take its flag arguments concatenated as its first argument, so neither this nor the **errFlag** specification begins with a **-**.
- The flag to use to turn error-checking on from the start is `-e`.
- The shell can turn error-checking on and off, and the commands to do so are `set +e` and `set -e`, respectively.

I should note that this specification is for Bourne Shells that are not part of Berkeley UNIX, as shells from Berkeley don't do error control. You can get a similar effect, however, by changing the last three lines to be:

```
hasErrCtl=no \
check="echo \"+ %s\"\n" \
ignore="sh -c '%s || exit 0\n"
```

This will cause PMake to execute the two commands

```
echo "+ cmd"
sh -c 'cmd || true'
```

for each command for which errors are to be ignored. (In case you are wondering, the thing for `ignore` tells the shell to execute another shell without error checking on and always exit 0, since the **||** causes the `exit 0` to be executed only if the first command exited non-zero, and if the first command exited zero, the shell will also exit zero, since that's the last command it executed).

4.5. Compatibility

There are three (well, 3 ½) levels of backwards-compatibility built into PMake. Most makefiles will need none at all. Some may need a little bit of work to operate correctly when run in parallel. Each level encompasses the previous levels (e.g. **–B** (one shell per command) implies **–V**) The three levels are described in the following three sections.

4.5.1. DEFCON 3 — Variable Expansion

As noted before, PMake will not expand a variable unless it knows of a value for it. This can cause problems for makefiles that expect to leave variables undefined except in special circumstances (e.g. if more flags need to be passed to the C compiler or the output from a text processor should be sent to a different printer). If the variables are enclosed in curly braces ("`${PRINTER}`"), the shell will let them pass. If they are enclosed in parentheses, however, the shell will declare a syntax error and the make will come to a grinding halt.

You have two choices: change the makefile to define the variables (their values can be overridden on the command line, since that's where they would have been set if you used Make, anyway) or always give the **–V** flag (this can be done with the `.MAKEFLAGS` target, if you want).

4.5.2. DEFCON 2 — The Number of the Beast

Then there are the makefiles that expect certain commands, such as changing to a different directory, to not affect other commands in a target's creation script. You can solve this is either by going back to executing one shell per command (which is what the **–B** flag forces PMake to do), which slows the process down a good bit and requires you to use semicolons and escaped newlines for shell constructs, or by changing the makefile to execute the offending command(s) in a subshell (by placing the line inside parentheses), like so:

```
install :: .MAKE
     (cd src; $(.PMAKE) install)
     (cd lib; $(.PMAKE) install)
     (cd man; $(.PMAKE) install)
```

This will always execute the three makes (even if the **−n** flag was given) because of the combination of the "::" operator and the `.MAKE` attribute. Each command will change to the proper directory to perform the install, leaving the main shell in the directory in which it started.

4.5.3. DEFCON 1 — Imitation is the Not the Highest Form of Flattery

The final category of makefile is the one where every command requires input, the dependencies are incompletely specified, or you simply cannot create more than one target at a time, as mentioned earlier. In addition, you may not have the time or desire to upgrade the makefile to run smoothly with PMake. If you are the conservative sort, this is the compatibility mode for you. It is entered either by giving PMake the **−M** flag (for Make), or by executing PMake as "`make`." In either case, PMake performs things exactly like Make (while still supporting most of the nice new features PMake provides). This includes:

- No parallel execution.
- Targets are made in the exact order specified by the makefile. The sources for each target are made in strict left-to-right order, etc.
- A single Bourne shell is used to execute each command, thus the shell's `$$` variable is useless, changing directories doesn't work across command lines, etc.
- If no special characters exist in a command line, PMake will break the command into words itself and execute the command directly, without executing a shell first. The characters that cause PMake to execute a shell are: `#`, `=`, |, `^`, `(`, `)`, `{`, `}`, `;`, `&`, `<`, `>`, `*`, `?`, `[`, `]`, `:`, `$`, `` ` ``, and `\`. You should notice that these are all the characters that are given special meaning by the shell (except `'` and `,` which PMake deals with all by its lonesome).
- The use of the null suffix is turned off.

4.6. The Way Things Work

When PMake reads the makefile, it parses sources and targets into nodes in a graph. The graph is directed only in the sense that PMake knows which way is up. Each node contains not only links to all its parents and children (the nodes that depend on it and those on which it depends, respectively), but also a count of the number of its children that have already been processed.

The most important thing to know about how PMake uses this graph is that the traversal is breadth-first and occurs in two passes.

After PMake has parsed the makefile, it begins with the nodes the user has told it to make (either on the command line, or via a `.MAIN` target, or by the target being the first in the file not labeled with the `.NOTMAIN` attribute) placed in a queue. It continues to take the node off the front of the queue, mark it as something that needs to be made, pass the node to `Suff_FindDeps` (mentioned earlier) to find any implicit sources for the node, and place all the node's children that have yet to be marked at the end of the queue. If any of the children is a `.USE` rule, its attributes are applied to the parent, then its commands are appended to the parent's list of commands and its children are linked to its parent. The parent's unmade children counter is then decremented (since the `.USE` node has been processed). You will note that this allows a `.USE` node to have children that are `.USE` nodes and the rules will be applied in sequence. If the node has no children, it is placed at the end of another queue to be examined in the second pass. This process continues until the first queue is empty.

At this point, all the leaves of the graph are in the examination queue. PMake removes the node at the head of the queue and sees if it is out-of-date. If it is, it is passed to a function that will execute the commands for the node asynchronously. When the commands have completed, all the node's parents have their unmade children counter decremented and, if the counter is then 0, they are placed on the examination queue. Likewise, if the node is up-to-date. Only those parents that were marked on the downward pass are processed in this way. Thus PMake traverses the graph back up to the nodes the user instructed it to create. When the examination queue is empty and no shells are running to create a target, PMake is finished.

Once all targets have been processed, PMake executes the commands attached to the `.END` target, either explicitly or through the use of an ellipsis in a shell script. If there were no errors during the entire process but there are still some targets unmade (PMake keeps a running count of how many targets are left to be made), there is a cycle in the graph. PMake does a depth-first traversal of the graph to find all the targets that weren't made and prints them out one by one.

5. Answers to Exercises

(3.1) This is something of a trick question, for which I apologize. The trick comes from the UNIX definition of a suffix, which PMake doesn't necessarily share. You will have noticed that all the suffixes used in this tutorial (and in UNIX in general) begin with a period (`.ms`, `.c`, etc.). Now, PMake's idea of a suffix is more like English's: it's the characters at the end of a word. With this in mind, one possible solution to this problem goes as follows:

```
.SUFFIXES       : ec.exe .exe ec.obj .obj .asm
ec.objec.exe .obj.exe :
        link -o $(.TARGET) $(.IMPSRC)
.asmec.obj      :
        asm -o $(.TARGET) -DDO_ERROR_CHECKING $(.IMPSRC)
.asm.obj        :
        asm -o $(.TARGET) $(.IMPSRC)
```

(3.2) The trick to this one lies in the ":=" variable-assignment operator and the ":S" variable-expansion modifier. Basically what you want is to take the pointer variable, so to speak, and transform it into an invocation of the variable at which it points. You might try something like

```
$(PTR:S/^/\$(/:S/$/))
```

which places "`$(`" at the front of the variable name and "`)`" at the end, thus transforming "`VAR`," for example, into "`$(VAR)`," which is just what we want. Unfortunately (as you know if you've tried it), since, as it says in the hint, PMake does no further substitution on the result of a modified expansion, that's *all* you get. The solution is to make use of ":=" to place that string into yet another variable, then invoke the other variable directly:

```
*PTR            := $(PTR:S/^/\$(/:S/$/)/)
```

You can then use "`$(*PTR)`" to your heart's content.

6. Glossary of Jargon

attribute: A property given to a target that causes PMake to treat it differently.

command script: The lines immediately following a dependency line that specify commands to execute to create each of the targets on the dependency line. Each line in the command script must begin with a tab.

command-line variable: A variable defined in an argument when PMake is first executed. Overrides all assignments to the same variable name in the makefile.

conditional: A construct much like that used in C that allows a makefile to be configured on the fly based on the local environment, or on what is being made by that invocation of PMake.

creation script: Commands used to create a target. See "command script."

dependency: The relationship between a source and a target. This comes in three flavors, as indicated by the operator between the target and the source. ':' gives a straight time-wise dependency (if the target is older than the source, the target is out-of-date), while '!' provides simply an ordering and always considers the target out-of-date. '::' is much like ':', save it creates multiple instances of a target each of which depends on its own list of sources.

dynamic source: This refers to a source that has a local variable invocation in it. It allows a single dependency line to specify a different source for each target on the line.

global variable: Any variable defined in a makefile. Takes precedence over variables defined in the environment, but not over command-line or local variables.

input graph: What PMake constructs from a makefile. Consists of nodes made of the targets in the makefile, and the links between them (the dependencies). The links are directed (from source to target) and there may not be any cycles (loops) in the graph.

local variable: A variable defined by PMake visible only in a target's shell script. There are seven local variables, not all of which are defined for every target: `.TARGET`, `.ALLSRC`, `.OODATE`, `.PREFIX`, `.IMPSRC`, `.ARCHIVE`, and `.MEMBER`. `.TARGET`, `.PREFIX`, `.ARCHIVE`, and `.MEMBER` may be used on dependency lines to create ‘‘dynamic sources.’’

makefile: A file that describes how a system is built. If you don't know what it is after reading this tutorial. . . .

modifier: A letter, following a colon, used to alter how a variable is expanded. It has no effect on the variable itself.

operator: What separates a source from a target (on a dependency line) and specifies the relationship between the two. There are three: ‘`:`’, ‘`::`’, and ‘`!`’.

search path: A list of directories in which a file should be sought. PMake's view of the contents of directories in a search path does not change once the makefile has been read. A file is sought on a search path only if it is exclusively a source.

shell: A program to which commands are passed in order to create targets.

source: Anything to the right of an operator on a dependency line. Targets on the dependency line are usually created from the sources.

special target: A target that causes PMake to do special things when it's encountered.

suffix: The tail end of a file name. Usually begins with a period, `.c` or `.ms`, e.g.

target: A word to the left of the operator on a dependency line. More generally, any file that PMake might create. A file may be (and often is) both a target and a source (what it is depends on how PMake is looking at it at the time — sort of like the wave/particle duality of light, you know).

transformation rule: A special construct in a makefile that specifies how to create a file of one type from a file of another, as indicated by their suffixes.

variable expansion: The process of substituting the value of a variable for a reference to it. Expansion may be altered by means of modifiers.

variable: A place in which to store text that may be retrieved later. Also used to define the local environment. Conditionals exist that test whether a variable is defined or not.

Table of Contents