NAME

dmake - maintain program groups, or interdependent files

SYNOPSIS

dmake [-ABceEhiknpqrsStTuVxX] [-v{dfimt}] [-P#] [-{f|C|K} file] [macro[*][+][:]=value ...] [target ...]

DESCRIPTION

dmake executes commands found in an external file called a makefile to update one or more target names. Each target may depend on zero or more prerequisite targets. If any of the target's prerequisites is newer than the target or if the target itself does not exist, then dmake will attempt to make the target.

If no -f command line option is present then dmake searches for an existing makefile from the list of prerequisites specified for the special target .MAKEFILES (see the STARTUP section for more details). If "-" is the name of the file specified to the -f flag then dmake uses standard input as the source of the makefile text.

Any macro definitions (arguments with embedded "=" signs) that appear on the command line are processed first and supersede definitions for macros of the same name found within the makefile. In general it is impossible for definitions found inside the makefile to redefine a macro defined on the command line, see the MACROS section for an exception.

If no target names are specified on the command line, then dmake uses the first non-special target found in the makefile as the default target. See the SPECIAL TARGETS section for the list of special targets and their function. dmake is a re-implementation of the UNIX Make utility with significant enhancements. Makefiles written for most previous versions of Make will be handled correctly by dmake. Known differences between dmake and other versions of make are discussed in the COMPATIBILITY section found at the end of this document.

OPTIONS

-A Enable AUGMAKE special inference rule transformations (see the "PERCENT(%) RULES" section), these are set to off by default.

-B Enable the use of spaces instead of <tabs> to begin recipe lines. This flag equivalent to the .NOTABS special macro and is further described below.

-c Use non-standard comment stripping. If you specify -c then dmake will treat any # character as a start of

comment character wherever it may appear unless it is escaped by a \.

-C [+]file
This option writes to file a copy of standard output and standard error from any child processes and from the dmake process itself. If you specify a + prior to the file name then the text is appended to the previous contents of file. This option is active in the MSDOS implementation only and is ignored by non-MSDOS versions of dmake.

-E Read the environment and define all strings of the form 'ENV-VAR=evalue' defined within as macros whose name is ENV-VAR, and whose value is 'evalue'. The environment is processed prior to processing the user specified makefile thereby allowing definitions in the makefile to override definitions in the environment.

-e Same as -E, except that the environment is processed after the user specified makefile has been processed (thus definitions in the environment override definitions in the makefile). The -e and -E options are mutually exclusive. If both are given the latter takes effect.

-f file
Use file as the source for the makefile text. Only one -f option is allowed.

-h Print the command summary for dmake.

-i Tells dmake to ignore errors, and continue making other targets. This is equivalent to the .IGNORE attribute or macro.

-K file
Turns on .KEEP_STATE state tracking and tells dmake to use file as the state file.

-k Causes dmake to ignore errors caused by command execution and to make all targets not depending on targets that could not be made. Ordinarily dmake stops after a command returns a non-zero status, specifying -k causes dmake to ignore the error and continue to make as much as possible.

-n Causes dmake to print out what it would have executed, but does not actually execute the commands. A special check is made for the string "$(MAKE)" inside a recipe line, if found, the line is expanded and invoked, thereby enabling recursive makes to give a full description of all that they will do. The check for "$(MAKE)" is disabled inside group recipes.

-p Print out a version of the digested makefile in human readable form. (useful for debugging, but cannot be re-read by dmake)

-P# On systems that support multi-processing cause dmake to use # concurrent child processes to make targets. See the "MULTI PROCESSING" section for more information.

-q Check and see if the target is up to date. Exits with code 0 if up to date, 1 otherwise.

-r Tells dmake not to read the initial startup makefile, see STARTUP section for more details.

-s Tells dmake to do all its work silently and not echo the commands it is executing to stdout (also suppresses warnings). This is equivalent to the .SILENT attribute or macro.

-S Force sequential execution of recipes on architectures which support concurrent makes. For backward compatibility with old makefiles that have nasty side-effect prerequisite dependencies.

-t Causes dmake to touch the targets and bring them up to date without executing any commands. Note that targets will not be created if they do not already exist.

-T Tells dmake to not perform transitive closure on the inference graph.

-u Force an unconditional update. (ie. do everything that would be done if everything that a target depended on was out of date)

-v[dfimt]
Verbose flag, when making targets print to stdout what we are going to make and what we think its time stamp is. The optional flags [dfimt] can be used to restrict the information that is displayed. In the absence of any optional flags all are assumed to be given (ie. -v is equivalent to -vdfimt). The meanings of the optional flags are:

d Notify of change directory operations only.

f Notify of file I/O operations only.

i Notify of inference algorithm operation only.

m Notify of target update operations only.

t Keep any temporary files created; normally they are automatically deleted.

-V Print the version of dmake, and values of builtin macros.

-x Upon processing the user makefile export all non-internally defined macros to the user's environment. This option together with the -e option allows SYSV AUGMAKE recursive makes to function as expected.

-X Inhibit the execution of #! lines found at the beginning of a makefile. The use of this flag prevents non-termination of recursive make invocations.

INDEX
Here is a list of the sections that follow and a short description of each. Perhaps you won't have to read the whole man page to find what you need.

STARTUP	Describes dmake initialization.
SYNTAX	Describes the syntax of makefile expressions.
ATTRIBUTES	Describes the notion of attributes and how they are used when making targets.
MACROS	Defining and expanding macros.
RULES AND TARGETS	How to define targets and their prerequisites.
RECIPES	How to tell dmake how to make a target.
TEXT DIVERSIONS	How to use text diversions in recipes and macro expansions.
SPECIAL TARGETS	Some targets are special.
SPECIAL MACROS	Macros used by dmake to alter the processing of the makefile, and those defined by dmake for the user.
CONTROL MACROS	Itemized list of special control macros.
RUN-TIME MACROS	Discussion of special run-time macros such as $@ and $<.

FUNCTION MACROS	GNU style function macros, only $(mktmp ...) for now.
DYNAMIC PREREQUISITES	Processing of prerequisites which contain macro expansions in their name.
BINDING TARGETS	The rules that dmake uses to bind a target to an existing file in the file system.
PERCENT(%) RULES	Specification of recipes to be used by the inference algorithm.
MAKING INFERENCES	The rules that dmake uses when inferring how to make a target which has no explicit recipe. This and the previous section are really a single section in the text.
MAKING TARGETS	How dmake makes targets other than libraries.
MAKING LIBRARIES	How dmake makes libraries.
KEEP STATE	A discussion of how .KEEP_STATE works.
MULTI PROCESSING	Discussion of dmake's parallel make facilities for architectures that support them.
CONDITIONALS	Conditional expressions which control the processing of the makefile.
EXAMPLES	Some hopefully useful examples.
COMPATIBILITY	How dmake compares with previous versions of make.
LIMITS	Limitations of dmake.
PORTABILITY	Comments on writing portable makefiles.
FILES	Files used by dmake.
SEE ALSO	Other related programs, and man pages.
AUTHOR	The guy responsible for this thing.
BUGS	Hope not.

STARTUP

When dmake begins execution it first processes the command line and then processes an initial startup-makefile. This is followed by an attempt to locate and process a user supplied makefile. The startup file defines the default values of all required control macros and the set of default rules for making targets and inferences. When searching for the startup makefile, dmake searches the following locations, in the order specified, until a startup file is located:

1. The location given as the value of the macro MAKESTARTUP defined on the command line.

2. The location given as the value of the environment variable MAKESTARTUP defined in the current environment.

3. The location given as the value of the macro MAKESTARTUP defined internally within dmake.

The above search is disabled by specifying the -r option on the command line. An error is issued if a startup makefile cannot be found and the -r option was not specified. A user may substitute a custom startup file by defining the MAKESTARTUP environment variable or by redefining the MAKESTARTUP macro on the command line. To determine where dmake looks for the default startup file, check your environment or issue the command "dmake -V".

A similar search is performed to locate a default user makefile when no **-f** command line option is specified. By default, the prerequisite list of the special target .MAKEFILES specifies the names of possible makefiles and the search order that dmake should use to determine if one exists. A typical definition for this target is:

```
.MAKEFILES : makefile.mk Makefile makefile
```

dmake will first look for makefile.mk and then the others. If a prerequisite cannot be found dmake will try to make it before going on to the next prerequisite. For example, makefile.mk can be checked out of an RCS file if the proper rules for doing so are defined in the startup file.

If the first line of the user makefile is of the form:

```
#! command command_args
```

then dmake will expand and run the command prior to reading any addtional input. If the return code of the command is zero then dmake will continue on to process the remainder of the user makefile, if the return code is non-zero then dmake will exit.

SYNTAX

This section is a summary of the syntax of makefile statements. The description is given in a style similar to BNF, where { } enclose items that may appear zero or more times, and [ ] enclose items that are optional. Alternative productions for a left hand side are indicated by '->', and newlines are significant. All symbols in bold type are text or names representing text supplied by the user.

```
Makefile -> { Statement }

Statement -> Macro-Definition
          -> Conditional
          -> Rule-Definition
          -> Attribute-Definition

Macro-Definition -> MACRO = LINE
                 -> MACRO *= LINE
                 -> MACRO := LINE
                 -> MACRO *:= LINE
                 -> MACRO += LINE
                 -> MACRO +:= LINE

Conditional ->  .IF expression
                   Makefile
                [ .ELIF expression
                   Makefile ]
                [ .ELSE
                   Makefile ]
                .END

expression -> LINE
           -> STRING == LINE
           -> STRING != LINE


Rule-Definition ->  target-definition
                       [ recipe ]

target-definition -> targets [attrs] op { PREREQUISITE } [; rcp-line]

targets -> target { targets }
        -> "target" { targets }

target -> special-target
       -> TARGET
```



```
attrs -> attribute { attrs }
      -> "attribute" { attrs }

op -> : { modifier }

modifier -> :
         -> ^
         -> !
         -> -

recipe -> { TAB rcp-line }
       -> [@][%][-] [
             { LINE }
          ]

rcp-line -> [@][%][-][+] LINE


Attribute-Definition -> attrs : targets


attribute -> .EPILOG
          -> .IGNORE
          -> .LIBRARY
          -> .MKSARGS
          -> .NOINFER
          -> .NOSTATE
          -> .PHONY
          -> .PRECIOUS
          -> .PROLOG
          -> .SETDIR=path
          -> .SILENT
          -> .SEQUENTIAL
          -> .SWAP
          -> .USESHELL
          -> .SYMBOL
          -> .UPDATEALL

special-target -> .ERROR
               -> .EXPORT
               -> .GROUPEPILOG
               -> .GROUPPROLOG
               -> .IMPORT
               -> .INCLUDE
               -> .INCLUDEDIRS
               -> .MAKEFILES
               -> .REMOVE
               -> .SOURCE
               -> .SOURCE.suffix
               -> .suffix1.suffix2
```

Where, TAB represents a <tab> character, STRING represents an arbitrary sequence of characters, and LINE represents a possibly empty sequence of characters terminated by a non-escaped (not immediately preceded by a backslash '\') new-line character. MACRO, PREREQUISITE, and TARGET each represent a string of characters not including space or tab which respectively form the name of a macro, prerequisite or target. The name may itself be a macro expansion expression. A LINE can be continued over several physical lines by terminating it with a single backslash character. Comments are initiated by the pound # character and extend to the end of line. All comment text is discarded, a '#' may be placed into the makefile text by escaping it with '\' (ie. \# translates to # when it is parsed). An exception to this occurs when a # is seen inside a recipe line that begins with a <tab> or is inside a group recipe. If you specify the **-c** command line switch then this behavior is disabled and **dmake** will treat all # characters as start of comment indicators unless they are escaped by \. A set of continued lines may be commented out by placing a single # at the start of the first line. A continued line cannot span more than one makefile.

white space is defined to be any combination of <space>, <tab>, and the sequence \<nl> when \<nl> is used to terminate a LINE. When processing **macro** definition lines, any amount of white space is allowed on either side of the macro operator (=, *=, :=, *:=, += or +:=), and white space is stripped from both before and after the macro value string. The sequence \<nl> is treated as white space during recipe expansion and is deleted from the final recipe string. You must escape the \<nl> with another \ in order to get a \ at the end of a recipe line. The \<nl> sequence is deleted from macro values when they are expanded.

When processing **target** definition lines, the recipe for a target must, in general, follow the first definition of the target (See the RULES AND TARGETS section for an exception), and the recipe may not span across multiple makefiles. Any targets and prerequisites found on a target definition line are taken to be white space separated tokens. The rule operator (op in SYNTAX section) is also considered to be a token but does not require white space to precede or follow it. Since the rule operator begins with a `:', traditional versions of make do not allow the `:' character to form a valid target name. **dmake** allows `:' to be present in target/prerequisite names as long as the entire target/prerequisite name is quoted. For example:

```
a:fred : test
```

would be parsed as TARGET = a, PREREQUISITES={fred, :,

test}, which is not what was intended. To fix this you must write:

```
"a:fred" : test
```

Which will be parsed as expected. Quoted target and prerequisite specifications may also contain white space thereby allowing the use of complex function macro expressions.. See the EXAMPLES section for how to apply " quoting to a list of targets.

ATTRIBUTES

dmake defines several target attributes. Attributes may be assigned to a single target, a group of targets, or to all targets in the makefile. Attributes are used to modify **dmake** actions during target update. The recognized attributes are:

.EPILOG — Insert shell epilog code when executing a group recipe associated with any target having this attribute set.

.IGNORE — Ignore an error when trying to make any target with this attribute set.

.LIBRARY — Target is a library.

.MKSARGS — If running in an MSDOS environment then use MKS extended argument passing conventions to pass arguments to commands. Non-MSDOS environments ignore this attribute.

.NOINFER — Any target with this attribute set will not be subjected to transitive closure if it is inferred as a prerequisite of a target whose recipe and prerequisites are being inferred. (i.e. the inference algorithm will not use any prerequisite with this attribute set, as a target) If specified as '.NOINFER:' (ie. with no prerequisites or targets) then the effect is equivalent to specifying -T on the command line.

.NOSTATE — Any target with this attribute set will not have command line flag information stored in the state file if .KEEP_STATE has been enabled.

.PHONY — Any target with this attribute set will have its recipe executed each time the target is made even if a file matching the target name can be located. Any targets that have a .PHONY attributed target as a prerequisite will be made each time the .PHONY attributed prerequisite is made.

.PRECIOUS Do not remove associated target under any circumstances. Set by default for any targets whose corresponding files exist in the file system prior to the execution of dmake.

.PROLOG Insert shell prolog code when executing a group recipe associated with any target having this attribute set.

.SEQUENTIAL Force a sequential make of the associated target's prerequisites.

.SETDIR Change current working directory to specified directory when making the associated target. You must specify the directory at the time the attribute is specified. To do this simply give .SETDIR=path as the attribute. path is expanded and the result is used as the value of the directory to change to. If path is surrounded by single quotes then path is not expanded, and is used literally as the directory name. If the path contains any ':' characters then the entire attribute string must be quoted using ". If a target having this attribute set also has the .IGNORE attribute set then if the change to the specified directory fails it will be ignored, and no error message will be issued.

.SILENT Do not echo the recipe lines when making any target with this attribute set, and do not issue any warnings.

.SWAP Under MSDOS when making a target with this attribute set swap the dmake executable to disk prior to executing the recipe line. Also see the '%' recipe line flag defined in the RECIPES section.

.SYMBOL Target is a library member and is an entry point into a module in the library. This attribute is used only when searching a library for a target. Targets of the form lib((entry)) have this attribute set automatically.

.USESHELL Force each recipe line of a target to be executed using a shell. Specifying this attribute is equivalent to specifying the '+' character at the start of each line of a non-group recipe.

.UPDATEALL Indicates that all the targets listed in this rule are updated by the execution of the accompanying recipe. A common example is the production of the y.tab.c and y.tab.h files by yacc when it is run on a grammar. Specifying .UPDATEALL in such a rule prevents the running of yacc twice, once for the y.tab.c file and once for the y.tab.h file.

All attributes are user setable and except for .UPDATEALL, .SETDIR and .MKSARGS may be used in one of two forms. The .MKSARGS attribute is restricted to use as a global attribute, and the use of the .UPDATEALL and .SETDIR attributes is restricted to rules of the second form only.

ATTRIBUTE_LIST : targets

assigns the attributes specified by ATTRIBUTE_LIST to each target in targets or

targets ATTRIBUTE_LIST : ...

assigns the attributes specified by ATTRIBUTE_LIST to each target in targets. In the first form if targets is empty (ie. a NULL list), then the list of attributes will apply to all targets in the makefile (this is equivalent to the common Make construct of ".IGNORE :" but has been modified to the notion of an attribute instead of a special target). Not all of the attributes have global meaning. In particular, .LIBRARY, .NOSTATE, .PHONY, .SETDIR, .SYMBOL and .UPDATEALL have no assigned global meaning.

Any attribute may be used with any target, even with the special targets. Some combinations are useless (e.g. .INCLUDE .PRECIOUS: ...), while others are useful (e.g. .INCLUDE .IGNORE : "file.mk" will not complain if file.mk cannot be found using the include file search rules, see the section on SPECIAL TARGETS for a description of .INCLUDE). If a specified attribute will not be used with the special target a warning is issued and the attribute is ignored.

MACROS

dmake supports six types of macro assignment.

MACRO = LINE This is the most common and familiar form of macro assignment. It assigns LINE literally as the value of MACRO. Future expansions of MACRO recursively expand its value.

MACRO *= LINE — This form behaves exactly as the simple '=' form with the exception that if MACRO already has a value then the assignment is not performed.

MACRO := LINE — This form differs from the simple '=' form in that it expands LINE prior to assigning it as the value of MACRO. Future expansions of MACRO do not recursively expand its value.

MACRO *:= LINE — This form behaves exactly as the ':=' form with the exception that if MACRO already has a value then the assignment and expansion are not performed.

MACRO += LINE — This form of macro assignment allows macro values to grow. It takes the literal value of LINE and appends it to the previous value of MACRO separating the two by a single space. Future expansions of MACRO recursively expand its value.

MACRO +:= LINE — This form is similar to the '+=' form except that the value of LINE is expanded prior to being added to the value of MACRO.

Macro expressions specified on the command line allow the macro value to be redefined within the makefile only if the macro is defined using the '+=' and '+:=' operators. Other operators will define a macro that cannot be further modified.

When **dmake** defines a non-environment macro it strips leading and trailing white space from the macro value. Macros imported from the environment via either the .IMPORT special target (see the SPECIAL TARGETS section), or the **-e**, or **-E** flags are an exception to this rule. Their values are always taken literally and white space is never stripped. In addition, named macros defined using the .IMPORT special target do not have their values expanded when they are used within a makefile. In contrast, environment macros that are imported due to the specification of the **-e** or **-E** flags are subject to expansion when used.

To specify a macro expansion enclose the name in () or {} and precede it with a dollar sign $. Thus $(TEST) represents an expansion of the macro variable named TEST. If TEST is defined then $(TEST) is replaced by its expanded value. If TEST is not defined then $(TEST) expands to the NULL string (this is equivalent to defining a macro as 'TEST='). A short form may be used for single character

named macros. In this case the parentheses are optional, and $(I) is equivalent to $I. Macro expansion is recursive, hence, if the value string contains an expression representing a macro expansion, the expansion is performed. Circular macro expansions are detected and cause an error to be issued.

When defining a macro the given macro name is first expanded before being used to define the macro. Thus it is possible to define macros whose names depend on values of other macros. For example, suppose CWD is defined as

```
CWD = $(PWD:b)
```

then the value of $(CWD) is the name of the current directory. This can be used to define macros specific to this directory, for example:

```
_$(CWD).prt = list of files to print...
```

The actual name of the defined macro is a function of the current directory. A construct such as this is useful when processing a hierarchy of directories using .SETDIR attributed targets and a collection of small distributed makefile stubs.

Macro variables may be defined within the makefile, on the command line, or imported from the environment.

dmake supports several non-standard macro expansions: The first is of the form:

<u>$(macro_name:modifier_list:modifier_list:...)</u>

where <u>modifier_list</u> is chosen from the set { D or d, F or f, B or b, S or s, T or t } and

```
d - directory portion of all path names
f - file (including suffix) portion of path names
b - file (not including suffix) portion of path names
s - simple pattern substitution
t - tokenization.
```

Thus if we have the example:

```
test = d1/d2/d3/a.out f.out d1/k.out
```

The following macro expansions produce the values on the right of '->' after expansion.

```
$(test:d)            -> d1/d2/d3/ d1/
$(test:b)            -> a f k
```

```
$(test:f)             -> a.out f.out k.out
${test:db}            -> d1/d2/d3/a f d1/k
${test:s/out/in/:f}   -> a.in f.in k.in
$(test:f:t"+")        -> a.out+f.out+k.out
```

If a token ends in a string composed from the value of the macro DIRBRKSTR (ie. ends in a directory separator string, e.g. '/' in UNIX) and you use the :d modifier then the expansion returns the directory name less the final directory separator string. Thus successive pairs of :d modifiers each remove a level of directory in the token string.

The tokenization modifier takes all white space separated tokens from the macro value and separates them by the quoted separator string. The separator string may contain the following escape codes \a => <bel>, \b => <backspace>, \f => <formfeed>, \n => <nl>, \r => <cr>, \t => <tab>, \v => <vertical tab>, \" => ", and \xxx => <xxx> where xxx is the octal representation of a character. Thus the expansion:

```
     $(test:f:t"+\n")
produces:
     a.out+
     f.out+
     k.out
```

The second non-standard form of macro expansion allows for recursive macros. It is possible to specify a $(<u>macro name</u>) or ${<u>macro name</u>} expansion where <u>macro name</u> contains more $(...) or ${ ... } macro expansions itself.

For example $(CC$(_HOST)$(_COMPILER)) will first expand CC$(_HOST)$(_COMPILER) to get a result and use that result as the name of the macro to expand. This is useful for writing a makefile for more than one target environment. As an example consider the following hypothetical case. Suppose that _HOST and _COMPILER are imported from the environment and are set to represent the host machine type and the host compiler respectively.

```
CFLAGS_VAX_CC = -c -O     # _HOST == "_VAX", _COMPILER == "_CC"
CFLAGS_PC_MSC = -c -ML # _HOST == "_PC",  _COMPILER == "_MSC"

# redefine CFLAGS macro as:

CFLAGS := $(CFLAGS$(_HOST)$(_COMPILER))
```

This causes CFLAGS to take on a value that corresponds to the environment in which the make is being invoked.

The final non-standard macro expansion is of the form:


```
string1{token_list}string2
```

where string1, string2 and token_list are expanded. After expansion, string1 is prepended to each token found in token_list and string2 is appended to each resulting token from the previous prepend. string1 and string2 are not delimited by white space whereas the tokens in token_list are. A null token in the token list is specified using "". Thus using another example we have:

```
test/{f1 f2}.o     --> test/f1.o test/f2.o
test/ {f1 f2}.o    --> test/ f1.o f2.o
test/{f1 f2} .o    --> test/f1 test/f2 .o
test/{"f1"  ""}.o  --> test/f1.o test/.o

and

test/{d1 d2}/{f1 f2}.o --> test/d1/f1.o test/d1/f2.o
                           test/d2/f1.o test/d2/f2.o
```

This last expansion is activated only when the first characters of <u>token list</u> appear immediately after the opening '{' with no intervening white space. The reason for this restriction is the following incompatibility with Bourne Shell recipes. The line

```
{ echo hello;}
```

is valid /bin/sh syntax; while

```
{echo hello;}
```

is not. Hence the latter triggers the enhanced macro expansion while the former causes it to be suppressed. See the SPECIAL MACROS section for a description of the special macros that **dmake** defines and understands.

RULES AND TARGETS

A makefile contains a series of entries that specify dependencies. Such entries are called <u>target/prerequisite</u> or <u>rule</u> definitions. Each rule definition is optionally followed by a set of lines that provide a recipe for updating any targets defined by the rule. Whenever **dmake** attempts to bring a target up to date and an explicit recipe is provided with a rule defining the target, that recipe is used to update the target. A rule definition begins with a line having the following syntax:

<u><targets></u> [<u><attributes></u>] <u><ruleop></u> [<u><prerequisites></u>] [;<u><recipe></u>]

<u>targets</u> is a non-empty list of targets. If the target is a special target (see SPECIAL TARGETS section below) then it

must appear alone on the rule line. For example:

```
.IMPORT .ERROR : ...
```

is not allowed since both .IMPORT and .ERROR are special targets. Special targets are not used in the construction of the dependency graph and will not be made.

attributes is a possibly empty list of attributes. Any attribute defined in the ATTRIBUTES section above may be specified. All attributes will be applied to the list of named targets in the rule definition. No other targets will be affected.

NOTE: As stated earlier, if both the target list and prerequisite list are empty but the attributes list is not, then the specified attributes affect all targets in the makefile.

ruleop is a separator which is used to identify the targets from the prerequisites. Optionally it also provides a facility for modifying the way in which dmake handles the making of the associated targets. In its simplest form the operator is a single ':', and need not be separated by white space from its neighboring tokens. It may additionally be followed by any of the modifiers { !, ^, -, : }, where:

! says execute the recipe for the associated targets once for each out of date prerequisite. Ordinarily the recipe is executed once for all out of date prerequisites at the same time.

^ says to insert the specified prerequisites, if any, before any other prerequisites already associated with the specified targets. In general, it is not useful to specify ^ with an empty list of prerequisites.

\- says to clear the previous list of prerequisites before adding the new prerequisites. Thus,

```
.SUFFIXES :
.SUFFIXES : .a .b
```

can be replaced by

```
.SUFFIXES :- .a .b
```

however the old form still works as expected. NOTE: .SUFFIXES is ignored by dmake it is used here simply as

an example.

: When the rule operator is not modified by a second ':' only one set of rules may be specified for making a target. Multiple definitions may be used to add to the list of prerequisites that a target depends on. However, if a target is multiply defined only one definition may specify a recipe for making the target.

When a target's rule operator is modified by a second ':' (:: for example) then this definition may not be the only definition with a recipe for the target. There may be other :: target definition lines that specify a different set of prerequisites with a different recipe for updating the target. Any such target is made if any of the definitions find it to be out of date with respect to the related prerequisites and the corresponding recipe is used to update the target.

In the following simple example, each rule has a '::' ruleop. In such an operator we call the first ':' the operator, and the second ':' the modifier.

```
a.o :: a.c b.h
   first recipe for making a.o

a.o :: a.y b.h
   second recipe for making a.o
```

If a.o is found to be out of date with respect to a.c then the first recipe is used to make a.o. If it is found out of date with respect to a.y then the second recipe is used. If a.o is out of date with respect to b.h then both recipes are invoked to make a.o. In the last case the order of invocation corresponds to the order in which the rule definitions appear in the makefile.

Targets defined using a single ':' operator with a recipe may be redefined again with a new recipe by using a ':' operator with a ':' modifier. This is equivalent to a target having been initially defined with a rule using a ':' modifier. Once a target is defined using a ':' modifier it may not be defined again with a recipe using only the ':' operator with no ':' modifier. In both cases the use of a ':' modifier creates a new list of prerequisites and makes it the current prerequisite list for the target. The ':' operator with no recipe always modifies the current list of prerequisites. Thus assuming each of the following definitions has a recipe attached, then:

```
joe :  fred ...     (1)
```

```
joe :: more ...     (2)

and

joe :: fred ...     (3)
joe :: more ...     (4)
```

are legal and mean: add the recipe associated with (2), or (4) to the set of recipes for joe, placing them after existing recipes for making joe. The constructs:

```
joe :: fred ...     (5)
joe : more ... (6)

and

joe : fred ... (7)
joe : more ... (8)
```

are errors since we have two sets of perfectly good recipes for making the target.

prerequisites is a possibly empty list of targets that must be brought up to date before making the current target.

recipe is a short form and allows the user to specify short rule definitions on a single line. It is taken to be the first recipe line in a larger recipe if additional lines follow the rule definition. If the semi-colon is present but the recipe line is empty (ie. null string) then it is taken to be an empty rule. Any target so defined causes the Don't know how to make ... error message to be suppressed when dmake tries to make the target and fails. This silence is maintained for rules that are terminated by a semicolon and have no following recipe lines, for targets listed on the command line, for the first target found in the makefile, and for any target having no recipe but containing a list of prerequisites (see the COMPATIBILITY section for an exception to this rule if the AUGMAKE (-A) flag was specified.

RECIPES

The traditional format used by most versions of Make defines the recipe lines as arbitrary strings that may contain macro expansions. They follow a rule definition line and may be spaced apart by comment or blank lines. The list of recipe lines defining the recipe is terminated by a new target definition, a macro definition, or end-of-file. Each recipe line MUST begin with a <TAB> character which may optionally be followed with one or all of the characters '@%+-'. The '-' indicates that non-zero exit values (ie. errors) are to be ignored when this recipe line is executed, the '+'

indicates that the current recipe line is to be executed using the shell, the '%' indicates that dmake should swap itself out to secondary storage (MSDOS only) before running the recipe and the '@' indicates that the recipe line should NOT be echoed to the terminal prior to being executed. Each switch is off by default (ie. by default, errors are significant, commands are echoed, no swapping is done and a shell is used only if the recipe line contains a character found in the value of the SHELLMETAS macro). Global settings activated via command line options or special attribute or target names may also affect these settings. An example recipe:

```
target :
     first recipe line
     second recipe line, executed independently of the first.
     @a recipe line that is not echoed
     -and one that has errors ignored
     %and one that causes dmake to swap out
     +and one that is executed using a shell.
```

The second and new format of the recipe block begins the block with the character '[' (the open group character) in the last non-white space position of a line, and terminates the block with the character ']' (the close group character) in the first non-white space position of a line. In this form each recipe line need not have a leading TAB. This is called a recipe group. Groups so defined are fed intact as a single unit to a shell for execution whenever the corresponding target needs to be updated. If the open group character '[' is preceded by one or all of -, @ or % then they apply to the entire group in the same way that they apply to single recipe lines. You may also specify '+' but it is redundant as a shell is already being used to run the recipe. See the MAKING TARGETS section for a description of how dmake invokes recipes. Here is an example of a group recipe:

```
target :
[
     first recipe line
     second recipe line
     all of these recipe lines are fed to a
     single copy of a shell for execution.
]
```

TEXT DIVERSIONS

dmake supports the notion of text diversions. If a recipe line contains the macro expression

```
$(mktmp[,[file][,text]] data)
```

then all text contained in the *data* expression is expanded and is written to a temporary file. The return value of the macro is the name of the temporary file.

data can be any text and must be separated from the 'mktmp' portion of the macro name by white-space. The only restriction on the data text is that it must contain a balanced number of parentheses of the same kind as are used to initiate the $(mktmp ...) expression. For example:

```
$(mktmp $(XXX))
```

is legal and works as expected, but:

```
$(mktmp text (to dump to file)
```

is not legal. You can achieve what you wish by either defining a macro that expands to '(' or by using {} in the macro expression; like this:

```
${mktmp text (to dump to file}
```

Since the temporary file is opened when the macro containing the text diversion expression is expanded, diversions may now be nested and any diversions that are created as part of ':=' macro expansions persist for the duration of the dmake run. The diversion text may contain the same escape codes as those described in the MACROS section. Thus if the *data* text is to contain new lines they must be inserted using the \n escape sequence. For example the expression:

```
all:
     cat $(mktmp this is a\n\
     test of the text diversion\n)
```

is replaced by:

```
cat /tmp/mk12294AA
```

where the temporary file contains two lines both of which are terminated by a new-line. If the *data* text spans multiple lines in the makefile then each line must be continued via the use of a \. A second more illustrative example generates a response file to an MSDOS link command:

```
OBJ = fred.obj mary.obj joe.obj
all : $(OBJ)
     link @$(mktmp $(^:t"+\n")\n)
```

The result of making 'all' in the second example is the command:


```
link @/tmp/mk02394AA
```

where the temporary file contains:

```
fred.obj+
mary.obj+
joe.obj
```

The last line of the file is terminated by a new-line which is inserted due to the \n found at the end of the *data* string.

If the optional *file* specifier is present then its expanded value is the name of the temporary file to create. Whenever a $(mktmp ...) macro is expanded the macro $(TMPFILE) is set to a new temporary file name. Thus the construct:

```
$(mktmp,$(TMPFILE) data)
```

is completely equivalent to not specifying the $(TMPFILE) optional argument. Another example that would be useful for MSDOS users with a Turbo-C compiler

```
$(mktmp,turboc.cfg $(CFLAGS))
```

will place the contents of CFLAGS into a local *turboc.cfg* file. The second optional argument, *text*, if present alters the name of the value returned by the $(mktmp ...) macro.

Under MS-DOS text diversions may be a problem. Many DOS tools require that path names which contain directories use the \ character to delimit the directories. Some users however wish to use the '/' to delimit pathnames and use environments that allow them to do so. The macro USESHELL is set to "yes" if the current recipe is forced to use a shell via the .USESHELL or '+' directives, otherwise its value is "no". The dmake startup files define the macro DIVFILE whose value is either the value of TMPFILE or the value of TMPFILE edited to replace any '/' characters to the appropriate value based on the current shell and whether it will be used to execute the recipe.

Previous versions of dmake defined text diversions using <+, +> strings, where <+ started a text diversion and +> terminated one. dmake is backward compatible with this construct if the <+ and +> appear literally on the same recipe line or in the same macro value string. In such instances the expression:

```
<+data+>
```

is mapped to:

```
$(mktmp data)
```

which is fully output compatible with the earlier construct. <+, +> constructs whose text spans multiple lines must be converted by hand to use $(mktmp ...).

If the environment variable TMPDIR is defined then the temporary file is placed into the directory specified by that variable. A makefile can modify the location of temporary files by defining a macro named TMPDIR and exporting it using the .EXPORT special target.

SPECIAL TARGETS

This section describes the special targets that are recognized by **dmake**. Some are affected by attributes and others are not.

.ERROR — If defined then the recipe associated with this target is executed whenever an error condition is detected by **dmake**. All attributes that can be used with any other target may be used with this target. Any prerequisites of this target will be brought up to date during its processing. NOTE: errors will be ignored while making this target, in extreme cases this may cause some problems.

.EXPORT — All prerequisites associated with this target are assumed to correspond to macro names and they and their values are exported to the environment as environment strings at the point in the makefile at which this target appears. Any attributes specified with this target are ignored. Only macros which have been assigned a value in the makefile prior to the export directive are exported, macros as yet undefined or macros whose value contains any of the characters "+=:*" are not exported. is suppre

.IMPORT — Prerequisite names specified for this target are searched for in the environment and defined as macros with their value taken from the environment. If the special name **.EVERYTHING** is used as a prerequisite name then all environment variables defined in the environment are imported. The functionality of the **-E** flag can be forced by placing the construct _.IMPORT : .EVERYTHING_ at the start of a makefile. Similarly, by placing the construct at the end, one can emulate the effect of the **-e** command line flag. If a prerequisite name

cannot be found in the environment an error message is issued. .IMPORT accepts the .IGNORE attribute. When given, it causes **dmake** to ignore the above error. See the MACROS section for a description of the processing of imported macro values.

.INCLUDE — Parse another makefile just as if it had been located at the point of the .INCLUDE in the current makefile. The list of prerequisites gives the list of makefiles to try to read. If the list contains multiple makefiles then they are read in order from left to right. The following search rules are used when trying to locate the file. If the filename is surrounded by " or just by itself then it is searched for in the current directory. If it is not found it is then searched for in each of the directories specified for the .INCLUDEDIRS special target. If the file name is surrounded by < and >, (ie. <my_spiffy_new_makefile>) then it is searched for only in the directories given by the .INCLUDEDIRS special target. In both cases if the file name is a fully qualified name starting at the root of the file system then it is only searched for once, and the .INCLUDEDIRS list is ignored. .INCLUDE accepts the .IGNORE and .SETDIR attributes. If .IGNORE attribute is given and the file cannot be found then **dmake** continues processing, otherwise an error message is generated. The .SETDIR attribute causes **dmake** to change directories to the specified directory prior to attempting the include operation.

.INCLUDEDIRS — The list of prerequisites specified for this target defines the set of directories to search when trying to include a makefile.

.KEEP_STATE — This special target is a synonym for the macro definition

```
.KEEP_STATE := _state.mk
```

It's effect is to turn on STATE keeping and to define __state.mk_ as the state file.

.MAKEFILES — The list of prerequisites is the set of files to try to read as the default makefile. By default this target is defined as:

.MAKEFILES : makefile.mk Makefile makefile

.SOURCE — The prerequisite list of this target defines a set of directories to check when trying to locate a target file name. See the section on BINDING of targets for more information.

.SOURCE.suff — The same as .SOURCE, except that the .SOURCE.suff list is searched first when trying to locate a file matching the a target whose name ends in the suffix .suff.

.REMOVE — The recipe of this target is used whenever **dmake** needs to remove intermediate targets that were made but do not need to be kept around. Such targets result from the application of transitive closure on the dependency graph.

In addition to the special targets above, several other forms of targets are recognized and are considered special, their exact form and use is defined in the sections that follow.

SPECIAL MACROS

dmake defines a number of special macros. They are divided into three classes: control macros, run-time macros, and function macros. The control macros are used by **dmake** to configure its actions, and are the preferred method of doing so. In the case when a control macro has the same function as a special target or attribute they share the same name as the special target or attribute. The run-time macros are defined when **dmake** makes targets and may be used by the user inside recipes. The function macros provide higher level functions dealing with macro expansion and diversion file processing.

CONTROL MACROS

To use the control macros simply assign them a value just like any other macro. The control macros are divided into three groups: string valued macros, character valued macros, and boolean valued macros.

The following are all of the string valued macros. This list is divided into two groups. The first group gives the string valued macros that are defined internally and cannot be directly set by the user.

DIRBRKSTR — Contains the string of chars used to terminate the name of a directory in a pathname. Under

UNIX its value is "/", under MSDOS its value is "/\:".

INCDEPTH — This macro’s value is a string of digits representing the current depth of makefile inclusion. In the first makefile level this value is zero.

MFLAGS — Is the list of flags that were given on the command line including a leading switch character. The -f flag is not included in this list.

MAKECMD — Is the name with which **dmake** was invoked.

MAKEDIR — Is the full path to the initial directory in which **dmake** was invoked.

MAKEFILE — Contains the string "-f _makefile_" where, _makefile_ is the name of initial user makefile that was first read.

MAKEFLAGS — Is the same as $(MFLAGS) but has no leading switch character. (ie. MFLAGS = -$(MAKEFLAGS))

MAKEMACROS — Contains the complete list of macro expressions that were specified on the command line.

MAKETARGETS — Contains the name(s) of the target(s), if any, that were specified on the command line.

MAXPROCESSLIMIT — Is a numeric string representing the maximum number of processes that **dmake** can use when making targets using parallel mode.

NULL — Is permanently defined to be the NULL string. This is useful when comparing a conditional expression to an NULL value.

PWD — Is the full path to the current directory in which make is executing.

TMPFILE — Is set to the name of the most recent temporary file opened by **dmake**. Temporary files are used for text diversions and for group recipe processing.

TMD — Stands for "To Make Dir", and is the path from the present directory (value of $(PWD)) to the directory that **dmake** was started up in (value of $(MAKEDIR)). This macro is modified when

.SETDIR attributes are processed.

USESHELL The value of this macro is set to "yes" if the current recipe is forced to use a shell for its execution via the .USESHELL or '+' directives, its value is "no" otherwise.

The second group of string valued macros control dmake behavior and may be set by the user.

.NOTABS When set to non-NULL enables the use of spaces as well as <tabs> to begin recipe lines. By default a non-group recipe is terminated by a line without any leading white-space or by a line not beggining with a <tab> character. Enabling this mode modifies the first condition of the above termination rule to terminate a non-group recipe with a line that contains only white-space. This mode does not effect the parsing of group recipes bracketed by [].

.SETDIR If this macro is assigned a value then dmake will change to the directory given by that value before making any targets.

AUGMAKE If set to a non NULL value will enable the transformation of special meta targets to support special AUGMAKE inferences (See the COMPATIBILITY section).

DIRSEPSTR Contains the string that is used to separate directory components when path names are constructed. It is defined with a default value at startup.

DIVFILE Is defined in the startup file and gives the name that should be returned for the diversion file name when used in $(mktmp ...) expansions, see the TEXT DIVERSION section for details.

.KEEP_STATE Assigning this macro a value tells dmake the name of the state file to use and turns on the keeping of state information for any targets that are brought up to date by the make.

GROUPFLAGS This macro gives the set of flags to pass to the shell when invoking it to execute a group recipe. The value of the macro is the

list of flags with a leading switch indicator. (ie. '-' under UNIX)

GROUPSHELL This macro defines the full path to the executable image to be used as the shell when processing group recipes. This macro must be defined if group recipes are used. It is assigned a default value in the startup makefile. Under UNIX this value is /bin/sh.

GROUPSUFFIX If defined, this macro gives the string to use as a suffix when creating group recipe files to be handed to the command interpreter. For example, if it is defined as .sh, then all temporary files created by dmake will end in the suffix .sh. Under MSDOS if you are using command.com as your GROUPSHELL, then this suffix must be set to .bat in order for group recipes to function correctly. The setting of GROUPSUFFIX and GROUPSHELL is done automatically for command.com in the startup.mk files.

MAKE Is defined in the startup file by default. The string $(MAKE) is recognized when using the -n option for single line recipes. Initially this macro is defined to have the value "$(MAKECMD) $(MFLAGS)".

MAKESTARTUP This macro defines the full path to the initial startup makefile. Use the -V command line option to discover its initial value.

MAXLINELENGTH This macro defines the maximum size of a single line of makefile input text. The size is specified as a number, the default value is defined internally and is shown via the -V option. A buffer of this size plus 2 is allocated for reading makefile text. The buffer is freed before any targets are made, thereby allowing files containing long input lines to be processed without consuming memory during the actual make. This macro can only be used to extend the line length beyond it's default minimum value.

MAXPROCESS Specify the maximum number of child processes to use when making targets. The default value of this macro is "1" and its value cannot exceed the value of the macro MAXPROCESSLIMIT. Setting the value of MAXPROCESS on the command line or in the makefile is equivalent to supplying a corresponding value to the -P flag on the command line.

PREP
: This macro defines the number of iterations to be expanded automatically when processing % rule definitions of the form:

  % : %.suff

  See the sections on PERCENT(%) RULES for details on how PREP is used.

SHELL
: This macro defines the full path to the executable image to be used as the shell when processing single line recipes. This macro must be defined if recipes requiring the shell for execution are to be used. It is assigned a default value in the startup makefile. Under UNIX this value is /bin/sh.

SHELLFLAGS
: This macro gives the set of flags to pass to the shell when invoking it to execute a single line recipe. The value of the macro is the list of flags with a leading switch indicator. (ie. '-' under UNIX)

SHELLMETAS
: Each time dmake executes a single recipe line (not a group recipe) the line is searched for any occurrence of a character defined in the value of SHELLMETAS. If such a character is found the recipe line is defined to require a shell to ensure its correct execution. In such instances a shell is used to invoke the recipe line. If no match is found the recipe line is executed without the use of a shell.

There is only one character valued macro defined by dmake: SWITCHAR contains the switch character used to introduce options on command lines. For UNIX its value is '-', and for MSDOS its value may be '/' or '-'. The macro is internally defined and is not user setable. The MSDOS version of dmake attempts to first extract SWITCHAR from an environment variable of the same name. If that fails it then attempts to use the undocumented getswitchar system call, and returns the result of that. Under MSDOS version 4.0 you must set the value of the environment macro SWITCHAR to '/' to obtain predictable behavior.

All boolean macros currently understood by dmake correspond directly to the previously defined attributes. These macros provide a second way to apply global attributes, and represent the preferred method of doing so. They are used by assigning them a value. If the value is not a NULL string then the boolean condition is set to on. If the value is a NULL string then the condition is set to off. There are five conditions defined and they correspond directly to the attributes of the same name. Their meanings are defined in the ATTRIBUTES section above. The macros are: .EPILOG, .IGNORE, .MKSARGS, .NOINFER, .PRECIOUS, .PROLOG, .SEQUENTIAL, .SILENT, .SWAP, and .USESHELL. Assigning any of these a non NULL value will globally set the corresponding attribute to on.

RUN_TIME MACROS

These macros are defined when dmake is making targets, and may take on different values for each target. $@ is defined to be the full target name, $? is the list of all out of date prerequisites, $& is the list of all prerequisites, $> is the name of the library if the current target is a library member, and $< is the list of prerequisites specified in the current rule. If the current target had a recipe inferred then $< is the name of the inferred prerequisite even if the target had a list of prerequisites supplied using an explicit rule that did not provide a recipe. In such situations $& gives the full list of prerequisites.

$* is defined as $(@:db) when making targets with explicit recipes and is defined as the value of % when making targets whose recipe is the result of an inference. In the first case $* is the target name with no suffix, and in the second case, is the value of the matched % pattern from the associated %-rule. $^ expands to the set of out of date prerequisites taken from the current value of $<. In addition to these, $$ expands to $, {{ expands to {, }} expands to }, and the strings <+ and +> are recognized as respectively starting and terminating a text diversion when they appear literally together in the same input line.

The difference between $? and $^ can best be illustrated by an example, consider:

```
fred.out : joe amy hello
     rules for making fred

fred.out : my.c your.h his.h her.h   # more prerequisites
```

Assume joe, amy, and my.c are newer then fred.out. When dmake executes the recipe for making fred.out the values of the following macros will be:

```
$@ --> fred.out
$* --> fred
$? --> joe amy my.c  # note the difference between $? and $^
$^ --> joe amy
$< --> joe amy hello
$& --> joe amy hello my.c your.h his.h her.h
```

FUNCTION MACROS

dmake supports a full set of functional macros. One of these, the $(mktmp ...) macro, is discussed in detail in the TEXT DIVERSION section and is not covered here.

$(null,_text_ **true false**)
expands the value of _text._ If it is NULL then the macro returns the value of the expansion of **true** and the expansion of **false** otherwise. The terms **true**, and **false** must be strings containing no white-space.

$(!null,_text_ **true false**)
Behaves identically to the previous macro except that the **true** string is chosen if the expansion of _text_ is not NULL.

$(eq,_text_a_,_text_b_ **true false**)
expands _text_a_ and _text_b_ and compares their results. If equal it returns the result of the expansion of the **true** term, otherwise it returns the expansion of the **false** term.

$(!eq,_text_a_,_text_b_ **true false**)
Behaves identically to the previous macro except that the **true** string is chosen if the expansions of the two strings are not equal

$(shell **command**)
Runs _command_ as if it were part of a recipe and returns, separated by a single space, all the non-white space terms written to stdout by the command. For example:

```
$(shell ls *.c)
```

will return _"a.c_ _b.c_ _c.c_ _d.c"_ if the files exist in the current directory. The recipe modification flags [+@%-] are honored if they appear as the first characters in the command. For example:

```
$(shell +ls *.c)
```


will run the command using the current shell.

$(sort **list**)
Will take all white-space separated tokens in _list_ and will return their sorted equivalent list.

$(strip **data**)
Will replace all strings of white-space in data by a single space.

$(subst,_pat_,_replacement_ **data**)
Will search for _pat_ in **data** and will replace any occurrence of _pat_ with the _replacement_ string. The expansion

```
$(subst,.o,.c $(OBJECTS))
```

is equivalent to:

```
$(OBJECTS:s/.o/.c/)
```

DYNAMIC PREREQUISITES

dmake looks for prerequisites whose names contain macro expansions during target processing. Any such prerequisites are expanded and the result of the expansion is used as the prerequisite name. As an example the line:

```
fred : $$@.c
```

causes the $$@ to be expanded when **dmake** is making fred, and it resolves to the target _fred_. This enables dynamic prerequisites to be generated. The value of @ may be modified by any of the valid macro modifiers. So you can say for example:

```
fred.out : $$(@:b).c
```

where the $$(@:b) expands to _fred_. Note the use of $$ instead of $ to indicate the dynamic expansion, this is due to the fact that the rule line is expanded when it is initially parsed, and $$ then returns $ which later triggers the dynamic prerequisite expansion. If you really want a $ to be part of a prerequisite name you must use $$$$. Dynamic macro expansion is performed in all user defined rules, and the special targets .SOURCE*, and .INCLUDEDIRS.

BINDING TARGETS

This operation takes a target name and binds it to an existing file, if possible. **dmake** makes a distinction between the internal target name of a target and its associated external file name. Thus it is possible for a target's internal name and its external file name to differ. To perform the binding, the following set of rules is used. Assume that we are trying to bind a target whose name is of the form X.suff, where .suff is the suffix and X is the stem portion (ie. that part which contains the directory and the basename). dmake takes this target name and performs a series of search operations that try to find a suitably named file in the external file system. The search operation is user controlled via the settings of the various .SOURCE targets.

1. If target has the .SYMBOL attribute set then look for it in the library. If found, replace the target name with the library member name and continue with step 2. If the name is not found then return.

2. Extract the suffix portion (that following the '.') of the target name. If the suffix is not null, look up the special target .SOURCE.<suff> (<suff> is the suffix). If the special target exists then search each directory given in the .SOURCE.<suff> prerequisite list for the target. If the target's suffix was null (ie. .suff was empty) then perform the above search but use the special target .SOURCE.NULL instead. If at any point a match is found then terminate the search. If a directory in the prerequisite list is the special name '.NULL ' perform a search for the full target name without prepending any directory portion (ie. prepend the NULL directory). (a default target of '.SOURCE : .NULL' is defined by dmake at startup, and is user redefinable)

3. The search in step 2. failed. Repeat the same search but this time use the special target .SOURCE.

4. The search in step 3. failed. If the target has the library member attribute (.LIBMEMBER) set then try to find the target in the library which was passed along with the .LIBMEMBER attribute (see the MAKING LIBRARIES section). The bound file name assigned to a target which is successfully located in a library is the same name that would be assigned had the search failed (see 5.).

5. The search failed. Either the target was not found in any of the search directories or no applicable .SOURCE special targets exist. If applicable .SOURCE special targets exist, but the target was not found, then dmake assigns the first

name searched as the bound file name. If no applicable .SOURCE special targets exist, then the full original target name becomes the bound file name.

There is potential here for a lot of search operations. The trick is to define .SOURCE.x special targets with short search lists and leave .SOURCE as short as possible. The search algorithm has the following useful side effect. When a target having the .LIBMEMBER (library member) attribute is searched for, it is first searched for as an ordinary file. When a number of library members require updating it is desirable to compile all of them first and to update the library at the end in a single operation. If one of the members does not compile and dmake stops, then the user may fix the error and make again. dmake will not remake any of the targets whose object files have already been generated as long as none of their prerequisite files have been modified as a result of the fix.

When defining .SOURCE and .SOURCE.x targets the construct

```
.SOURCE :
.SOURCE : fred gery
```

is equivalent to

```
.SOURCE :- fred gery
```

dmake correctly handles the UNIX Make variable VPATH. By definition VPATH contains a list of ':' separated directories to search when looking for a target. dmake maps VPATH to the following special rule:

```
.SOURCE :^ $(VPATH:s/:/ /)
```

Which takes the value of VPATH and sets .SOURCE to the same set of directories as specified in VPATH.

PERCENT(%) RULES AND MAKING INFERENCES

When dmake makes a target, the target's set of prerequisites (if any) must exist and the target must have a recipe which dmake can use to make it. If the makefile does not specify an explicit recipe for the target then dmake uses special rules to try to infer a recipe which it can use to make the target. Previous versions of Make perform this task by using rules that are defined by targets of the form .<suffix>.<suffix> and by using the .SUFFIXES list of suffixes. The exact workings of this mechanism were sometimes difficult to understand and often limiting in their usefulness. Instead, dmake supports the concept of %-meta rules. The syntax and semantics of these rules differ from standard rule lines as follows:

<%-target> [<attributes>] <ruleop> [<%-prerequisites>] [;<recipe>]

where %-target is a target containing exactly a single '%' sign, attributes is a list (possibly empty) of attributes, ruleop is the standard set of rule operators, %-prerequisites , if present, is a list of prerequisites containing zero or more '%' signs, and recipe, if present, is the first line of the recipe.

The %-target defines a pattern against which a target whose recipe is being inferred gets matched. The pattern match goes as follows: all chars are matched exactly from left to right up to but not including the % sign in the pattern, % then matches the longest string from the actual target name not ending in the suffix given after the % sign in the pattern. Consider the following examples:

```
%.c       matches fred.c but not joe.c.Z
dir/%.c   matches dir/fred.c but not dd/fred.c
fred/%    matches fred/joe.c but not f/joe.c
%         matches anything
```

In each case the part of the target name that matched the % sign is retained and is substituted for any % signs in the prerequisite list of the %-meta rule when the rule is selected during inference and **dmake** constructs the new dependency. As an example the following %-meta rules describe the following:

```
%.c : %.y ; recipe...
```

describes how to make any file ending in .c if a corresponding file ending in .y can be found.

```
foo%.o : fee%.k ; recipe...
```

is used to describe how to make fooxxxx.o from feexxxx.k.

```
%.a :; recipe...
```

describes how to make a file whose suffix is .a without inferring any prerequisites.

```
%.c : %.y yaccsrc/%.y ; recipe...
```

is a short form for the construct:

```
%.c : %.y ; recipe...
%.c : yaccsrc/%.y ; recipe...
```


ie. It is possible to specify the same recipe for two %-rules by giving more than one prerequisite in the prerequisite list. A more interesting example is:

```
% : RCS/%,v ; co $<
```

which describes how to take any target and check it out of the RCS directory if the corresponding file exists in the RCS directory. The equivalent SCCS rule would be:

```
% : s.% ; get $<
```

The previous RCS example defines an infinite rule, because it says how to make anything from RCS/%,v, and anything also includes RCS/fred.c,v. To limit the size of the graph that results from such rules **dmake** uses the macro variable PREP (stands for % repetition). By default the value of this variable is 0, which says that no repetitions of a %-rule are to be generated. If it is set to something greater than 0, then that many repetitions of any infinite %-rule are allowed. If in the above example PREP was set to 1, then **dmake** would generate the dependency graph:

```
% --> RCS/%,v --> RCS/RCS/%,v,v
```

Where each link is assigned the same recipe as the first link. PREP should be used only in special cases, since it may result in a large increase in the number of possible prerequisites tested. **dmake** further assumes that any target that has no suffix can be made from a prerequisite that has at least one suffix.

dmake supports dynamic prerequisite generation for prerequisites of %-meta rules. This is best illustrated by an example. The RCS rule shown above can infer how to check out a file from a corresponding RCS file only if the target is a simple file name with no directory information. That is, the above rule can infer how to find RCS/fred.c,v from the target fred.c, but cannot infer how to find srcdir/RCS/fred.c,v from srcdir/fred.c because the above rule will cause **dmake** to look for RCS/srcdir/fred.c,v; which does not exist (assume that srcdir has its own RCS directory as is the common case).

A more versatile formulation of the above RCS check out rule is the following:

```
% :  $$(@:d)RCS/$$(@:f),v : co $@
```

This rule uses the dynamic macro $@ to specify the prerequisite to try to infer. During inference of this rule the macro $@ is set to the value of the target of the %-meta rule and the appropriate prerequisite is generated by extracting the directory portion of the target name (if any), appending the string RCS/ to it, and appending the target file name with a trailing ,v attached to the previous result.

dmake can also infer indirect prerequisites. An inferred target can have a list of prerequisites added that will not show up in the value of $< but will show up in the value of $? and $&. Indirect prerequisites are specified in an inference rule by quoting the prerequisite with single quotes. For example, if you had the explicit dependency:

```
fred.o : fred.c ; rule to make fred.o
fred.o : local.h
```

then this can be inferred for fred.o from the following inference rule:

```
%.o : %.c 'local.h' ; rule to make a .o from a .c
```

You may infer indirect prerequisites that are a function of the value of '%' in the current rule. The meta-rule:

```
%.o : %.c '$(INC)/%.h' ; rule to make a .o from a .c
```

infers an indirect prerequisite found in the INC directory whose name is the same as the expansion of $(INC), and the prerequisite name depends on the base name of the current target. The set of indirect prerequisites is attached to the meta rule in which they are specified and are inferred only if the rule is used to infer a recipe for a target. They do not play an active role in driving the inference algorithm. The construct:

```
%.o : %.c %.f 'local.h'; recipe
```

is equivalent to:

```
%.o : %.c 'local.h' : recipe
%.o : %.f 'local.h' : recipe
```

If any of the attributes .SETDIR, .EPILOG, .PROLOG, .SILENT, .USESHELL, .SWAP, .PRECIOUS, .LIBRARY, .NOSTATE and .IGNORE are given for a %-rule then when that rule is bound to a target as the result of an inference, the target's set of attributes is augmented by the attributes from the above set that are specified in the bound %-rule. Other attributes specified for %-meta rules are not inherited by the target. The .SETDIR attribute is treated in a special way. If the

target already had a .SETDIR attribute set then **dmake** changes to that directory prior to performing the inference. During inference any .SETDIR attributes for the inferred prerequisite are honored. The directories must exist for a %-meta rule to be selected as a possible inference path. If the directories do not exist no error message is issued, instead the corresponding path in the inference graph is rejected.

dmake also supports the old format special target .<suffix>.<suffix> by identifying any rules of this form and mapping them to the appropriate %-rule. So for example if an old makefile contains the construct:

```
.c.o :; cc -c $< -o $@
```

dmake maps this into the following %-rule:

```
%.o : %.c; cc -c $< -o $@
```

Furthermore, **dmake** understands several SYSV AUGMAKE special targets and maps them into corresponding %-meta rules. These transformation must be enabled by providing the -A flag on the command line or by setting the value of AUGMAKE to non-NULL. The construct

```
.suff :; recipe
```

gets mapped into:

```
% : %.suff; recipe
```

and the construct

```
.c~.o :; recipe
```

gets mapped into:

```
%.o : s.%.c ; recipe
```

In general, a special target of the form .<str>~ is replaced by the %-rule construct s.%.<str>, thereby providing support for the syntax used by SYSV AUGMAKE for providing SCCS support. When enabled, these mappings allow processing of existing SYSV makefiles without modifications.

dmake bases all of its inferences on the inference graph constructed from the %-rules defined in the makefile. It knows exactly which targets can be made from which prerequisites by making queries on the inference graph. For this reason .SUFFIXES is not needed and is completely ignored.

For a %-meta rule to be inferred as the rule whose recipe will be used to make a target, the target's name must match the %-target pattern, and any inferred %-prerequisite must already exist or have an explicit recipe so that the prerequisite can be made. Without _transitive_ _closure_ on the inference graph the above rule describes precisely when an inference match terminates the search. If transitive closure is enabled (the usual case), and a prerequisite does not exist or cannot be made, then dmake invokes the inference algorithm recursively on the prerequisite to see if there is some way the prerequisite can be manufactured. For, if the prerequisite can be made then the current target can also be made using the current %-meta rule. This means that there is no longer a need to give a rule for making a .o from a .y if you have already given a rule for making a .o from a .c and a .c from a .y. In such cases dmake can infer how to make the .o from the .y via the intermediary .c and will remove the .c when the .o is made. Transitive closure can be disabled by giving the -T switch on the command line.

A word of caution. dmake bases its transitive closure on the %-meta rule targets. When it performs transitive closure it infers how to make a target from a prerequisite by performing a pattern match as if the potential prerequisite were a new target. The set of rules:

```
%.o : %.c :; rule for making .o from .c
%.c : %.y :; rule for making .c from .y
% : RCS/%,v :; check out of RCS file
```

will, by performing transitive closure, allow dmake to infer how to make a .o from a .y using a .c as an intermediate temporary file. Additionally it will be able to infer how to make a .y from an RCS file, as long as that RCS file is in the RCS directory and has a name which ends in .y,v. The transitivity computation is performed dynamically for each target that does not have a recipe. This has potential to be costly if the %-meta rules are not carefully specified. The .NOINFER attribute is used to mark a %-meta node as being a final target during inference. Any node with this attribute set will not be used for subsequent inferences. As an example the node RCS/%,v is marked as a final node since we know that if the RCS file does not exist there likely is no other way to make it. Thus the standard startup makefile contains an entry similar to:

```
     .NOINFER : RCS/%,v
```

Thereby indicating that the RCS file is the end of the inference chain.

Whenever the inference algorithm determines that a target can be made from more than one prerequisite and the

inference chains for the two methods are the same length the algorithm reports an ambiguity and prints the ambiguous inference chains.

dmake tries to remove intermediate files resulting from transitive closure if the file is not marked as being PRECIOUS, or the **-u** flag was not given on the command line, and if the inferred intermediate did not previously exist. Intermediate targets that existed prior to being made are never removed. This is in keeping with the philosophy that dmake should never remove things from the file system that it did not add. If the special target .REMOVE is defined and has a recipe then dmake constructs a list of the intermediate files to be removed and makes them prerequisites of .REMOVE. It then makes .REMOVE thereby removing the prerequisites if the recipe of .REMOVE says to. Typically .REMOVE is defined in the startup file as:

```
     .REMOVE :; $(RM) $<
```

MAKING TARGETS

In order to update a target dmake must execute a recipe. When a recipe needs to be executed it is first expanded so that any macros in the recipe text are expanded, and it is then either executed directly or passed to a shell. dmake supports two types of recipes. The regular recipes and group recipes.

When a regular recipe is invoked dmake executes each line of the recipe separately using a new copy of a shell if a shell is required. Thus effects of commands do not generally persist across recipe lines. (e.g. cd requests in a recipe line do not carry over to the next recipe line) The decision on whether a shell is required to execute a command is based on the value of the macro SHELLMETAS or on the specification of '+' or .USESHELL for the current recipe or target respectively. If any character in the value of SHELLMETAS is found in the expanded recipe text-line or the use of a shell is requested explicitly via '+' or .USESHELL then the command is executed using a shell, otherwise the command is executed directly. The shell that is used for execution is given by the value of the macro SHELL. The flags that are passed to the shell are given by the value of SHELLFLAGS. Thus dmake constructs the command line:

```
     $(SHELL) $(SHELLFLAGS) $(expanded_recipe_command)
```

Normally dmake writes the command line that it is about to invoke to standard output. If the .SILENT attribute is set for the target or for the recipe line (via @), then the recipe line is not echoed.

Group recipe processing is similar to that of regular recipes, except that a shell is always invoked. The shell that is invoked is given by the value of the macro GROUPSHELL, and its flags are taken from the value of the macro GROUPFLAGS. If a target has the .PROLOG attribute set then **dmake** prepends to the shell script the recipe associated with the special target .GROUPPROLOG, and if the attribute .EPILOG is set as well, then the recipe associated with the special target .GROUPEPILOG is appended to the script file. This facility can be used to always prepend a common header and common trailer to group recipes. Group recipes are echoed to standard output just like standard recipes, but are enclosed by lines beginning with [and].

The recipe flags [+,-,%,@] are recognized at the start of a recipe line even if they appear in a macro. For example:

```
SH = +
all:
     $(SH)echo hi
```

is completely equivalent to writing

```
SH = +
all:
     +echo hi
```

The last step performed by **dmake** prior to running a recipe is to set the macro CMNDNAME to the name of the command to execute (determined by finding the first white-space ending token in the command line). It then sets the macro CMNDARGS to be the remainder of the line. **dmake** then expands the macro COMMAND which by default is set to

```
COMMAND = $(CMNDNAME) $(CMNDARGS)
```

The result of this final expansion is the command that will be executed. The reason for this expansion is to allow for a different interface to the argument passing facilities (esp. under DOS) than that provided by **dmake.** You can for example define COMMAND to be

```
COMMAND = $(CMNDNAME) @$(mktmp $(CMNDARGS))
```

which dumps the arguments into a temporary file and runs the command

```
$(CMNDNAME) @/tmp/ASAD23043
```

which has a much shorter argument list. It is now up to the command to use the supplied argument as the source for all

other arguments. As an optimization, if COMMAND is not defined **dmake** does not perform the above expansion. On systems, such as UNIX, that handle long command lines this provides a slight saving in processing the makefiles.

MAKING LIBRARIES

Libraries are easy to maintain using **dmake.** A library is a file containing a collection of object files. Thus to make a library you simply specify it as a target with the .LIBRARY attribute set and specify its list of prerequisites. The prerequisites should be the object members that are to go into the library. When **dmake** makes the library target it uses the .LIBRARY attribute to pass to the prerequisites the .LIBMEMBER attribute and the name of the library. This enables the file binding mechanism to look for the member in the library if an appropriate object file cannot be found. A small example best illustrates this.

```
mylib.a .LIBRARY : mem1.o mem2.o mem3.o
     rules for making library...
     # remember to remove .o's when lib is made

# equivalent to:  '%.o : %.c ; ...'
.c.o :; rules for making .o from .c say
```

dmake will use the .c.o rule for making the library members if appropriate .c files can be found using the search rules. NOTE: this is not specific in any way to C programs, they are simply used as an example.

dmake tries to handle the old library construct format in a sensible way. The construct <u>lib(member.o)</u> is separated and the <u>lib</u> portion is declared as a library target. The new target is defined with the .LIBRARY attribute set and the <u>member.o</u> portion of the construct is declared as a prerequisite of the lib target. If the construct <u>lib(member.o)</u> appears as a prerequisite of a target in the makefile, that target has the new name of the lib assigned as its prerequisite. Thus the following example:

```
a.out : ml.a(a.o) ml.a(b.o); $(CC) -o $@  $<

.c.o :; $(CC) -c $(CFLAGS) -o $@  $<
%.a:
     ar rv $@ $?
     ranlib $@
     rm -rf $?
```

constructs the following dependency graph.

```
a.out : ml.a; $(CC) -o $@  $<
ml.a .LIBRARY : a.o b.o
```

```
%.o : %.c ; $(CC) -c $(CFLAGS) -o $@  $<
%.a :
     ar rv $@ $?
     ranlib $@
     rm -rf $?
```

and making a.out then works as expected.

The same thing happens for any target of the form lib((entry)). These targets have an additional feature in that the entry target has the .SYMBOL attribute set automatically.

NOTE: If the notion of entry points is supported by the archive and by dmake (currently not the case) then dmake will search the archive for the entry point and return not only the modification time of the member which defines the entry but also the name of the member file. This name will then replace entry and will be used for making the member file. Once bound to an archive member the .SYMBOL attribute is removed from the target. This feature is presently disabled as there is little standardization among archive formats, and we have yet to find a makefile utilizing this feature (possibly due to the fact that it is unimplemented in most versions of UNIX Make).

Finally, when dmake looks for a library member it must first locate the library file. It does so by first looking for the library relative to the current directory and if it is not found it then looks relative to the current value of $(TMD). This allows commonly used libraries to be kept near the root of a source tree and to be easily found by dmake.

KEEP STATE

dmake supports the keeping of state information for targets that it makes whenever the macro .KEEP_STATE is assigned a value. The value of the macro should be the name of a state file that will contain the state information. If state keeping is enabled then each target that does not poses the .NOSTATE attribute will have a record written into the state file indicating the target's name, the current directory, the command used to update the target, and which, if any, :: rule is being used. When you make this target again if any of this information does not match the previous settings and the target is not out dated it will still be re-made. The assumption is that one of the conditions above has changed and that we wish to remake the target. For example, state keeping is used in the maintenance of dmake to test compile different versions of the source using different compilers. Changing the compiler causes the compilation flags to be modified and hence all sources to be recompiled.

The state file is an ascii file and is portable, however it is not in human readable form as the entries represent hash keys of the above information.

The Sun Microsystem's Make construct

```
.KEEP_STATE :
```

is recognized and is mapped to .KEEP_STATE:=_state.mk. The dmake version of state keeping does not include scanning C source files for dependencies like Sun Make. This is specific to C programs and it was felt that it does not belong in make. dmake instead provides the tool, cdepend, to scan C source files and to produce depedency information. Users are free to modify cdepend to produce other dependency files. (NOTE: cdepend does not come with the distribution at this time, but will be available in a patch in the near future)

MULTI PROCESSING

If the architecture supports it then dmake is capable of making a target's prerequisites in parallel. dmake will make as much in parallel as it can and use a number of child processes up to the maximum specified by MAXPROCESS or by the value supplied to the -P command line flag. A parallel make is enabled by setting the value of MAXPROCESS (either directly or via -P option) to a value which is > 1. dmake guarantees that all dependencies as specified in the makefile are honored. A target will not be made until all of its prerequisites have been made. If a parallel make is being performed then the following restrictions on parallelism are enforced.

1. Individual recipe lines in a non-group recipe are performed sequentially in the order in which they are specified within the makefile and in parallel with the recipes of other targets.

2. If a target contains multiple recipe definitions (cf. :: rules) then these are performed sequentially in the order in which the :: rules are specified within the makefile and in parallel with the recipes of other targets.

3. If a target rule contains the '!' modifier, then the recipe is performed sequentially for the list of outdated prerequisites and in parallel with the recipes of other targets.

4. If a target has the .SEQUENTIAL attribute set then all of its prerequisites are made sequentially relative to one another (as if MAXPROCESS=1), but

in parallel with other targets in the makefile.

Note: If you specify a parallel make then the order of target update and the order in which the associated recipes are invoked will not correspond to that displayed by the -n flag.

CONDITIONALS

dmake supports a makefile construct called a conditional. It allows the user to conditionally select portions of makefile text for input processing and to discard other portions. This becomes useful for writing makefiles that are intended to function for more than one target host and environment. The conditional expression is specified as follows:

```
.IF  expression
   ... if text ...
.ELIF  expression
   ... if text ...
.ELSE
   ... else text ...
.END
```

The .ELSE and .ELIF portions are optional, and the conditionals may be nested (ie. the text may contain another conditional). .IF, .ELSE, and .END may appear anywhere in the makefile, but a single conditional expression may not span multiple makefiles.

expression can be one of the following three forms:

```
<text> | <text> == <text> | <text> != <text>
```

where text is either text or a macro expression. In any case, before the comparison is made, the expression is expanded. The text portions are then selected and compared. White space at the start and end of the text portion is discarded before the comparison. This means that a macro that evaluates to nothing but white space is considered a NULL value for the purpose of the comparison. In the first case the expression evaluates TRUE if the text is not NULL otherwise it evaluates FALSE. The remaining two cases both evaluate the expression on the basis of a string comparison. If a macro expression needs to be equated to a NULL string then compare it to the value of the macro $(NULL). You can use the $(shell ...) macro to construct more complex test expressions.

EXAMPLES

```
# A simple example showing how to use make
#
```



```
prgm : a.o b.o
     cc a.o b.o -o prgm
a.o : a.c g.h
     cc a.c -o $@
b.o : b.c g.h
     cc b.c -o $@
```

In the previous example prgm is remade only if a.o and/or b.o is out of date with respect to prgm. These dependencies can be stated more concisely by using the inference rules defined in the standard startup file. The default rule for making .o's from .c's looks something like this:

```
%.o : %.c; cc -c $(CFLAGS) -o $@ $<
```

Since there exists a rule (defined in the startup file) for making .o's from .c's **dmake** will use that rule for manufacturing a .o from a .c and we can specify our dependencies more concisely.

```
prgm : a.o b.o
     cc -o prgm $<
a.o b.o : g.h
```

A more general way to say the above using the new macro expansions would be:

```
SRC = a b
OBJ = {$(SRC)}.o

prgm : $(OBJ)
     cc -o $@ $<

$(OBJ) : g.h
```

If we want to keep the objects in a separate directory, called objdir, then we would write something like this.

```
SRC = a b
OBJ = {$(SRC)}.o

prgm : $(OBJ)
     cc $< -o $@

$(OBJ) : g.h
%.o : %.c
     $(CC) -c $(CFLAGS) -o $(@:f) $<
     mv $(@:f) objdir

.SOURCE.o : objdir       # tell make to look here for .o's
```

An example of building library members would go something like this: (NOTE: The same rules as above will be used to produce .o's from .c's)

```
SRC  = a b
LIB  = lib
LIBm = { $(SRC) }.o

prgm: $(LIB)
     cc -o $@ $(LIB)

$(LIB) .LIBRARY : $(LIBm)
     ar rv $@ $<
     rm $<
```

Finally, suppose that each of the source files in the previous example had the ':' character in their target name. Then we would write the above example as:

```
SRC  = f:a f:b
LIB  = lib
LIBm = "{ $(SRC) }.o"         # put quotes around each token

prgm: $(LIB)
     cc -o $@ $(LIB)

$(LIB) .LIBRARY : $(LIBm)
     ar rv $@ $<
     rm $<
```

COMPATIBILITY

There are two notable differences between **dmake** and the standard version of BSD UNIX 4.2/4.3 Make.

1. BSD UNIX 4.2/4.3 Make supports wild card filename expansion for prerequisite names. Thus if a directory contains a.h, b.h and c.h, then a line like

   target: *.h

   will cause UNIX make to expand the *.h into "a.h b.h c.h". **dmake** does not support this type of filename expansion.

2. Unlike UNIX make, touching a library member causes **dmake** to search the library for the member name and to update the library time stamp. This is only implemented in the UNIX version. MSDOS and other versions may not have librarians that keep file time stamps, as a result **dmake** touches the library file itself, and prints a warning.

dmake is not compatible with GNU Make. In particular it does not understand GNU Make's macro expansions that query the file system.

dmake is fully compatible with SYSV AUGMAKE, and supports the following AUGMAKE features:

1. The word **include** appearing at the start of a line can be used instead of the ".INCLUDE :" construct understood by **dmake**.

2. The macro modifier expression $(macro:str=sub) is understood and is equivalent to the expression $(macro:s/str/sub), with the restriction that str must match the following regular expression:

   str[|\t][|\t]*

   (ie. str only matches at the end of a token where str is a suffix and is terminated by a space, a tab, or end of line) Normally <u>sub</u> is expanded before the substitution is made, if you specify -A on the command line then sub is not expanded.

3. The macro % is defined to be $@ (ie. $% expands to the same value as $@).

4. The AUGMAKE notion of libraries is handled correctly.

5. When defining special targets for the inference rules and the AUGMAKE special target handling is enabled then the special target .X is equivalent to the %-rule "% : %.X".

6. Directories are always made if you specify **-A**. This is consistent with other UNIX versions of Make.

7. Makefiles that utilize virtual targets to force making of other targets work as expected if AUGMAKE special target handling is enabled. For example:

   ```
   FRC:
   myprog.o : myprog.c $(FRC) ; ...
   ```

   Works as expected if you issue the command

   '**dmake** -A FRC=FRC'

   but fails with a 'don't know how to make FRC' error message if you do not specify AUGMAKE special target handling via the -A flag (or by setting AUGMAKE:=yes internally).

LIMITS

In some environments the length of an argument string is restricted. (e.g. MSDOS command line arguments cannot be longer than 128 bytes if you are using the standard command.com command interpreter as your shell, dmake text diversions may help in these situations.)

PORTABILITY

To write makefiles that can be moved from one environment to another requires some forethought. In particular you must define as macros all those things that may be different in the new environment. dmake has two facilities that help to support writing portable makefiles, recursive macros and conditional expressions. The recursive macros, allow one to define environment configurations that allow different environments for similar types of operating systems. For example the same make script can be used for SYSV and BSD but with different macro definitions.

To write a makefile that is portable between UNIX and MSDOS requires both features since in almost all cases you will need to define new recipes for making targets. The recipes will probably be quite different since the capabilities of the tools on each machine are different. Different macros will be needed to help handle the smaller differences in the two environments.

FILES

Makefile, makefile, startup.mk (use dmake -V to tell you where the startup file is)

SEE ALSO

sh(1), csh(1), touch(1), f77(1), pc(1), cc(1)
S.I. Feldman *Make - A Program for Maintaining Computer Programs*

AUTHOR

Dennis Vadura, CS Dept. University of Waterloo.
dvadura@watdragon.uwaterloo.ca
Many thanks to Carl Seger for his helpful suggestions, and to Trevor John Thompson for his many excellent ideas and informative bug reports.

BUGS

Some system commands return non-zero status inappropriately. Use -i ('-' within the makefile) to overcome the difficulty.

Some systems do not have easily accessible time stamps for library members (MSDOS, AMIGA, etc) for these dmake uses the time stamp of the library instead and prints a warning the

first time it does so. This is almost always ok, except when multiple makefiles update a single library file. In these instances it is possible to miss an update if one is not careful.

This man page is way too long.

WARNINGS

Rules supported by make(1) may not work if transitive closure is turned off (-T, .NOINFER).

PWD from csh/ksh will cause problems if a cd operation is performed and -e or -E option is used.

Using internal macros such as COMMAND, may wreak havoc if you don't understand their functionality.

If multiple MACRO=line arguments appear on the command line, only the first is used. Beware of this in conjunction with the MAKEMACROS variable.