CHAPTER 16

Managing Projects with NMAKE

This chapter describes the Microsoft Program Maintenance Utility (NMAKE) version 1.20. NMAKE is a sophisticated command processor that saves time and simplifies project management. Once you specify which project files depend on others, NMAKE automatically builds your project without recompiling files that haven't changed since the last build.

If you are using the Programmer's Workbench (PWB) to build your project, PWB automatically creates a makefile and calls NMAKE to run the file. You may want to read this chapter if you intend to build your program outside of PWB, if you want to understand or modify a makefile created by PWB, or if you want to use a foreign makefile in PWB.

NMAKE can swap itself to expanded memory, extended memory, or disk to leave room for the commands it spawns. (For more information, see the description of the /M option on page 531.)

New Features

NMAKE version 1.20 offers the following new features. For details of each feature, see the reference part of this chapter.

- New options: /B, /K, /M, /V
- The **!MESSAGE** directive
- Two preprocessing operators: **DEFINED**, **EXIST**
- Three keywords for use with the **!ELSE** directive: **IF**, **IFDEF**, **IFNDEF**
- New directives: **!ELSEIF**, **!ELSEIFDEF**, **!ELSEIFNDEF**
- Addition of .CPP and .CXX to the **.SUFFIXES** list
- Predefined macros for C++ programs: **CPP**, **CXX**, **CPPFLAGS**, **CXXFLAGS**
- Predefined inference rules for C++ programs

Overview

NMAKE works by looking at the “time stamp” of a file. A time stamp is the time and date the file was last modified. Time stamps are assigned by most operating systems in 2-second intervals. NMAKE compares the time stamps of a “target” file and its “dependent” files. A target is usually a file you want to create, such as an executable file, though it could be a label for a set of commands you wish to execute. A dependent is usually a file from which a target is created, such as a source file. A target is “out-of-date” if any of its dependents has a later time stamp than the target or if the target does not exist. (For information on how the 2-second interval affects your build, see the description of the /B option on page 530.)

Warning For NMAKE to work properly, the date and time setting on your system must be consistent relative to previous settings. If you set the date and time each time you start the system, be careful to set it accurately. If your system stores a setting, be certain that the battery is working.

When you run NMAKE, it reads a “makefile” that you supply. A makefile (sometimes called a description file) is a text file containing a set of instructions that NMAKE uses to build your project. The instructions consist of description blocks, macros, directives, and inference rules. Each description block typically lists a target (or targets), the target’s dependents, and the commands that build the target. NMAKE compares the time stamp on the target file with the time stamp on the dependent files. If the time stamp of any dependent is the same as or later than the time stamp of the target, NMAKE updates the target by executing the commands listed in the description block.

It is possible to run NMAKE without a makefile. In this case, NMAKE uses predefined macros and inference rules along with instructions given on the command line or in TOOLS.INI. (For information on the TOOLS.INI file, see page 534.)

NMAKE’s main purpose is to help you build programs quickly and easily. However, it is not limited to compiling and linking; NMAKE can run other types of programs and can execute operating system commands. You can use NMAKE to prepare backups, move files, and perform other project-management tasks that you ordinarily do at the operating-system prompt.

This chapter uses the term “build,” as in building a target, to mean evaluating the time stamps of a target and its dependent and, if the target is out of date, executing the commands associated with the target. The term “build” has this meaning whether or not the commands actually create or change the target file.

Running NMAKE

You invoke NMAKE with the following syntax:

NMAKE [[*options*]] [[*macros*]] [[*targets*]]

The *options* field lists NMAKE options, which are described in the following section, “Command-Line Options.”

The *macros* field lists macro definitions, which allow you to change text in the makefile. The syntax for macros is described in “User-Defined Macros” on page 551.

The *targets* field lists targets to build. NMAKE builds only the targets listed on the command line. If you don’t specify a target, NMAKE builds only the first target in the first dependency in the makefile. (You can use a pseudotarget to tell NMAKE to build more than one target. See “Pseudotargets” on page 540.)

NMAKE uses the following priorities to determine how to conduct the build:

1. If the /F option is used, NMAKE searches the current or specified directory for the specified makefile. NMAKE halts and displays an error message if the file does not exist.
2. If you do not use the /F option, NMAKE searches the current directory for a file named MAKEFILE.
3. If MAKEFILE does not exist, NMAKE checks the command line for target files and tries to build them using inference rules (either defined in TOOLS.INI or predefined). This feature lets you use NMAKE without a makefile as long as NMAKE has an inference rule for the target.
4. If a makefile is not used and the command line does not specify a target, NMAKE halts and displays an error message.

Example

The following command specifies an option (/S) and a macro definition (`"program=sample"`) and tells NMAKE to build two targets (`sort.exe` and `search.exe`). The command does not specify a makefile, so NMAKE looks for MAKEFILE or uses inference rules.

```
NMAKE /S "program=sample" sort.exe search.exe
```

For information on NMAKE macros, see page 550.

Command-Line Options

NMAKE accepts options for controlling the NMAKE session. Options are not case sensitive and can be preceded by either a slash (/) or a dash (–).

You can specify some options in the makefile or in TOOLS.INI.

/A
Forces NMAKE to build all evaluated targets, even if the targets are not out-of-date with respect to their dependents. This option does not force NMAKE to build unrelated targets.

/B
Tells NMAKE to execute a dependency even if time stamps are equal. Most operating systems assign time stamps with a resolution of 2 seconds. If your commands execute quickly, NMAKE may conclude that a file is up to date when in fact it is not. This option may result in some unnecessary build steps but is recommended when running NMAKE on very fast systems.

/C
Suppresses default NMAKE output, including nonfatal NMAKE error or warning messages, time stamps, and the NMAKE copyright message. If both /C and /K are specified, /C suppresses the warnings issued by /K.

/D
Displays information during the NMAKE session. The information is interspersed in NMAKE's default output to the screen. NMAKE displays the time stamp of each target and dependent evaluated in the build and issues a message when a target does not exist. Dependents for a target precede the target and are indented. The /D and /P options are useful for debugging a makefile.

To set or clear /D for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" on page 572.

/E
Causes environment variables to override macro definitions in the makefile. See "Macros" on page 550.

/F *filename*
Specifies *filename* as the name of the makefile. Zero or more spaces or tabs precede *filename*. If you supply a dash (**–**) instead of a filename, NMAKE gets makefile input from the standard input device. (End keyboard input with either F6 or CTRL+Z.) NMAKE accepts more than one makefile; use a separate /F specification for each makefile input.

If you omit /F, NMAKE searches the current directory for a file called MAKEFILE (with no extension) and uses it as the makefile. If MAKEFILE doesn't exist, NMAKE uses inference rules for the command-line targets.

/HELP
Calls the QuickHelp utility. If NMAKE cannot locate the Help file or QuickHelp, it displays a brief summary of NMAKE command-line syntax.

/I

Ignores exit codes from all commands listed in the makefile. NMAKE processes the whole makefile even if errors occur. To ignore exit codes for part of a makefile, use the dash (–) command modifier or the **.IGNORE** directive; see "Command Modifiers" on page 544 and "Dot Directives" on page 570. To set or clear /I for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" on page 572. To ignore errors for unrelated parts of the build, use the /K option; /I overrides /K if both are specified.

/K

Continues the build for unrelated parts of the dependency tree if a command terminates with an error. By default, NMAKE halts if any command returns a nonzero exit code. If this option is specified and a command returns a nonzero exit code, NMAKE ceases to execute the block containing the command. It does not attempt to build the targets that depend on the results of that command; instead, it issues a warning and builds other targets. When /K is specified and the build is not complete, NMAKE returns an exit code of 1. This differs from the /I option, which ignores exit codes entirely; /I overrides /K if both are specified. The /C option suppresses the warnings issued by /K.

/M

Swaps NMAKE to disk to conserve extended or expanded memory under MS-DOS. By default, NMAKE leaves room for commands to be executed in low memory by swapping itself to extended memory (if enough space exists there) or to expanded memory (if there is not sufficient extended memory but there is enough expanded memory) or to disk. The /M option tells NMAKE to ignore any extended or expanded memory and to swap only to disk.

/N

Displays but does not execute the commands that would be executed by the build. Preprocessing commands are executed. This option is useful for debugging makefiles and checking which targets are out-of-date. To set or clear /N for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" on page 572.

/NOLOGO

Suppresses the NMAKE copyright message.

/P

Displays NMAKE information to the standard output device, including all macro definitions, inference rules, target descriptions, and the **.SUFFIXES** list, before running the NMAKE session. If /P is specified without a makefile or command-line target, NMAKE displays the information and does not issue an error. The /P and /D options are useful for debugging a makefile.

/Q
Checks time stamps of targets that would be updated by the build but does not run the build. NMAKE returns a zero exit code if the targets are up-to-date and a nonzero exit code if any target is out-of-date. Only preprocessing commands in the makefile are executed. This option is useful when running NMAKE from a batch file.

/R
Clears the **.SUFFIXES** list and ignores inference rules and macros that are defined in the TOOLS.INI file or that are predefined.

/S
Suppresses the display of all executed commands. To suppress the display of commands in part of a makefile, use the **@** command modifier or the **.SILENT** directive; see "Command Modifiers" on page 544 and "Dot Directives" on page 570. To set or clear /S for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" on page 572.

/T
Changes time stamps of command-line targets (or first target in the makefile if no command-line targets are specified) to the current time and executes preprocessing commands but does not run the build. Contents of target files are not modified.

/V
Causes all macros to be inherited when recursing. By default, only macros defined on the command line and environment-variable macros are inherited when NMAKE is called recursively. This option makes all macros available to the recursively called NMAKE session. See "Inherited Macros" on page 563.

/X *filename*
Sends all NMAKE error output to *filename*, which can be a file or a device. Zero or more spaces or tabs can precede *filename*. If you supply a dash (**–**) instead of a filename, NMAKE sends its error output to standard output. By default, NMAKE sends errors to standard error. This option does not affect output that is sent to standard error by commands in the makefile.

/?
Displays a brief summary of NMAKE command-line syntax and exits to the operating system.

Example

The following command line specifies two NMAKE options:

```
NMAKE /F sample.mak /C targ1 targ2
```

The /F option tells NMAKE to read the makefile SAMPLE.MAK. The /C option tells NMAKE not to display nonfatal error messages and warnings. The command specifies two targets (`targ1` and `targ2`) to update.

NMAKE Command File

You can place a sequence of command-line arguments in a text file and pass the file's name as a command-line argument to NMAKE. NMAKE opens the command file and reads the arguments. You can use a command file to overcome the limit on the length of a command line in the operating system (128 characters in MS-DOS).

To provide input to NMAKE with a command file, type

NMAKE **@***commandfile*

The *commandfile* is the name of a text file containing the information NMAKE expects on the command line. Precede the name of the command file with an at sign (**@**). You can specify a path with the filename.

NMAKE treats the file as if it were a single set of arguments. It replaces each line break with a space. Macro definitions that contain spaces must be enclosed in quotation marks; see "Where to Define Macros" on page 552.

You can split input between the command line and a command file. Specify **@***commandfile* on the command line at the place where the file's information is expected. Command-line input can precede and/or follow the command file. You can specify more than one command file.

Example 1

If a file named UPDATE contains the line

```
/S "program = sample" sort.exe search.exe
```

you can start NMAKE with the command

```
NMAKE @update
```

The effect is the same as if you entered the following command line:

```
NMAKE /S "program = sample" sort.exe search.exe
```

Example 2

The following is another version of the UPDATE file:

```
/S "program \
= sample" sort.exe search.exe
```

The backslash (\) allows the macro definition to span two lines.

Example 3

If the command file called UPDATE contains the line

```
/S "program = sample" sort.exe
```

you can start NMAKE with the command

```
NMAKE @update search.exe
```

The TOOLS.INI File

You can customize NMAKE by placing commonly used information in the TOOLS.INI initialization file. Settings for NMAKE must follow a line that begins with the tag `[NMAKE]`. The tag is not case sensitive. This section of the initialization file can contain any makefile information. NMAKE uses this information in every session, unless you run NMAKE with the /R option. NMAKE looks for TOOLS.INI first in the current directory and then in the directory specified by the INIT environment variable.

You can use the **!CMDSWITCHES** directive to specify command-line options in TOOLS.INI. An option specified this way is in effect for every NMAKE session. This serves the same purpose as does an environment variable, which is a feature available in other utilities. For more information on **!CMDSWITCHES**, see page 572.

Macros and inference rules appearing in TOOLS.INI can be overridden. See “Precedence Among Macro Definitions” on page 563 and “Precedence Among Inference Rules” on page 570.

NMAKE reads information in TOOLS.INI before it reads makefile information. Thus, for example, a description block appearing in TOOLS.INI acts as the first description block in the makefile; this is true for every NMAKE session, unless /R is specified.

To place a comment in TOOLS.INI, specify the comment on a separate line beginning with a semicolon (;). You can also specify comments with a number sign (#) as you can in a makefile; for more information, see “Comments” on page 536.

Example

The following is an example of text in a TOOLS.INI file:

```
[NMAKE]
; macros
AS     = masm
AFLAGS = /FR /LA /ML /MX /W2
; inference rule
.asm.obj:
    $(AS) /ZD ZI $(AFLAGS) $.asm
```

NMAKE reads and applies the lines following `[NMAKE]`. The example redefines the macro **AS** to invoke the Microsoft Macro Assembler MASM.EXE utility., redefines the macro **AFLAGS**, and redefines the inference rule for making .OBJ files from .ASM sources. These NMAKE features are explained throughout this chapter.

Contents of a Makefile

An NMAKE makefile contains description blocks, macros, inference rules, and directives. This section presents general information about writing makefiles. The rest of the chapter describes each of the elements of makefiles in detail.

Using Special Characters as Literals

You may need to specify as a literal character one of the characters that NMAKE uses for a special purpose. These characters are:

```
:  ;  #  (  )  $  ^  \  {  }  !  @  —
```

To use one of these characters without its special meaning, place a caret (^) in front of it. NMAKE ignores carets that precede characters other than the special characters listed previously. A caret within a quoted string is treated as a literal caret character.

You can also use a caret at the end of a line to insert a literal newline character in a string or macro. The caret tells NMAKE to interpret the newline character as part of the macro, not a line break. Note that this effect differs from using a backslash (\ ) to continue a line in a macro definition. A newline character that follows a backslash is replaced with a space. For more information, see “User-Defined Macros” on page 551.

In a command, a percent symbol (**%**) can represent the beginning of a file specifier. (See "Filename-Parts Syntax" on page 546.) NMAKE interprets %s as a filename, and it interprets the character sequence of %| followed by d, e, f, p, or F as part or all of a filename or path. If you need to represent these characters literally in a command, specify a double percent sign (**%%**) in place of a single one. In all other situations, NMAKE interprets a single % literally. However, NMAKE always interprets a double %% as a single %. Therefore, to represent a literal %%, you can specify either three percent signs, %%%, or four percent signs, %%%%.

To use the dollar sign (**$**) as a literal character in a command, you must specify two dollar signs (**$$**); this method can also be used in other situations where **^$** also works.

For information on literal characters in macro definitions, see "Special Characters in Macros" on page 551.

Wildcards

You can use MS-DOS wildcards (***** and **?**) to specify target and dependent names. NMAKE expands wildcards that appear on dependency lines. A wildcard specified in a command is passed to the command; NMAKE does not expand it.

Example

In the following description block, the wildcard `*` is used twice:

```
project.exe : *.asm
    ml *.asm /Feproject.exe
```

NMAKE expands the `*.asm` in the dependency line and looks at all files having the .ASM extension in the current directory. If any .ASM file is out-of-date, the ML command expands the `*.c` and compiles and links all files.

Comments

To place a comment in a makefile, precede it with a number sign (#). If the entire line is a comment, the # must appear at the beginning of the line. Otherwise, the # follows the item being commented. NMAKE ignores all text from the number sign to the next newline character.

Command lines cannot contain comments; this is true even for a command that is specified on the same line as a dependency line or inference rule. This is because NMAKE does not parse a command; instead, it passes the entire command to the operating system. However, a comment can appear between lines in a commands block. To change a command to a comment, insert a # at the beginning of the command line.

You can use comments in the following situations:

```
# Comment on line by itself

OPTIONS = /MAP  # Comment on macro definition line

all.exe : one.obj two.obj  # Comment on dependency line
    link one.obj two.obj;
# Comment in commands block
    copy one.exe \release
# Command turned into comment:
#   copy *.obj \objects

.obj.exe:  # Comment in inference rule
```

To specify a literal #, precede it with a caret (^), as in the following:

```
DEF = ^#define  #Macro representing a C preprocessing directive
```

Comments can also appear in a TOOLS.INI file. TOOLS.INI allows an additional form of comment using a semicolon (;). See “The TOOLS.INI File” on page 534.

Long Filenames

You can use long filenames if they are supported by your file system. However, you must enclose the name in double quotation marks, as in the following dependency line:

```
all : "VeryLongFileName.exe"
```

Description Blocks

Description blocks form the heart of the makefile. The following is a typical NMAKE description block:

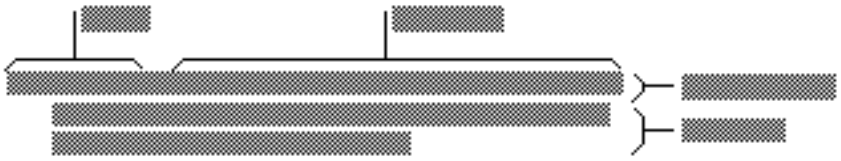

Figure 16.1 NMAKE Description Block

The first line in a description block is the “dependency line.” In this example, the the dependency contains one “target” and three “dependents.” The dependency is followed by a commands block that lists one or more commands.

The following sections discuss dependencies, targets, and dependents. The contents of a commands block are described in “Commands” on page 543.

Dependency Line

A description block begins with a “dependency line.” A dependency line specifies one or more “target” files and then lists zero or more “dependent” files. If a target does not exist, or if its time stamp is earlier than that of any dependent, NMAKE executes the commands block for that target. The following is an example of a dependency line:

```
myapp.exe : myapp.obj another.obj myapp.def
```

This dependency line tells NMAKE to rebuild the MYAPP.EXE target whenever MYAPP.OBJ, ANOTHER.OBJ, or MYAPP.DEF has changed more recently than MYAPP.EXE.

The dependency line must not be indented (it cannot start with a space or tab). The first target must be specified at the beginning of the line. Targets are separated from dependents by a single colon, except as described in “Using Targets in Multiple Description Blocks” on page 539. The colon can be preceded or followed by zero or more spaces or tabs. The entire dependency must appear on one line; however, you can extend the line by placing a backslash (\) after a target or dependent and continuing the dependency on the next line.

Before executing any commands, NMAKE moves through all dependencies and applicable inference rules to build a “dependency tree” that specifies all the steps required to fully update the target. NMAKE checks to see if dependents themselves are targets in other dependency lists, if any dependents in those lists are targets elsewhere, and so on. After it builds the dependency tree, NMAKE checks time stamps. If it finds any dependents in the tree that are newer than the target, NMAKE builds the target.

Targets

The targets section of the dependency line lists one or more target names. At least one target must be specified. Separate multiple target names with one or more spaces or tabs. You can specify a path with the filename. Targets are not case sensitive. A target (including path) cannot exceed 256 characters.

If the name of the last target before the colon (**:**) is a single character, you must put a space between the name and the colon; otherwise, NMAKE interprets the letter-colon combination as a drive specifier.

Usually a target is the name of a file to be built using the commands in the description block. However, a target can be any valid filename, or it can be a pseudotarget. (For more information, see “Pseudotargets” on page 540.)

NMAKE builds targets specified on the NMAKE command line. If a command-line target is not specified, NMAKE builds the first target in the first dependency in the makefile.

The example in the previous section tells NMAKE how to build a single target file called MYAPP.EXE if it is missing or out-of-date.

Using Targets in Multiple Description Blocks

A target can appear in only one description block when specified using the single-colon (**:**) syntax to separate the target from the dependent. To update a target using more than one description block, specify two consecutive colons (**::**) between targets and dependents. One use for this feature is for building a complex target that contains components created with different commands.

Example

The following makefile updates a library:

```
target.lib :: one.asm two.asm three.asm
    ML one.asm two.asm three.asm
    LIB target -+one.obj -+two.obj -+three.obj;
target.lib :: four.c five.c
    CL /c four.c five.c
    LIB target -+four.obj -+five.obj;
```

If any of the assembly-language files have changed more recently than the library, NMAKE assembles the source files and updates the library. Similarly, if any of the C-language files have changed, NMAKE compiles the C files and updates the library.

Accumulating Targets in Dependencies

Dependency lines are cumulative when the same target appears more than once in a single description block. For example,

```
bounce.exe : jump.obj
bounce.exe : up.obj
    echo Building bounce.exe...
```

is evaluated by NMAKE as

```
bounce.exe : jump.obj up.obj
    echo Building bounce.exe...
```

This evaluation has several effects. Since NMAKE builds the dependency tree based on one target at a time, the lines can contain other targets, as in:

```
bounce.exe leap.exe : jump.obj
bounce.exe climb.exe : up.obj
    echo Building bounce.exe...
```

The preceding example is evaluated by NMAKE as

```
bounce.exe : jump.obj
leap.exe : jump.obj
bounce.exe : up.obj
climb.exe : up.obj...
    echo Building bounce.exe...
```

NMAKE evaluates a dependency for each of the three targets as if each were specified in a separate description block. If `bounce.exe` or `climb.exe` is out-of-date, NMAKE runs the given command. If `leap.exe` is out-of-date, the given command does not apply, and NMAKE tries to use an inference rule.

If the same target is specified in two single-colon dependency lines in different locations in the makefile, and if commands appear after only one of the lines, NMAKE interprets the dependency lines as if they were adjacent or combined. This can cause an unwanted side effect: NMAKE does not invoke an inference rule for the dependency that has no commands (see “Inference Rules” on page 563). Rather, it assumes that the dependencies belong to one description block and executes the commands specified with the other dependency.

The following makefile is interpreted in the same way as the preceding examples:

```
bounce.exe : jump.obj
    echo Building bounce.exe...
.
.
.
bounce.exe : up.obj
```

This effect does not occur if the colons are doubled (**::**) after the duplicate targets. A double-colon dependency with no commands block invokes an inference rule, even if another double-colon dependency containing the same target is followed by a commands block.

Pseudotargets

A “pseudotarget” is a target that doesn’t specify a file but instead names a label for use in executing a group of commands. NMAKE interprets the pseudotarget as a file that does not exist and thus is always out-of-date. When NMAKE evaluates a pseudotarget, it always executes its commands block. Be sure that the current directory does not contain a file with a name that matches the pseudotarget.

A pseudotarget name must follow the syntax rules for filenames. Like a filename target, a pseudotarget name is not case sensitive. However, if the name does not have an extension (that is, it does not contain a period), it can exceed the 8-character limit for filenames and can be up to 256 characters long.

A pseudotarget can be listed as a dependent. A pseudotarget used this way must appear as a target in another dependency; however, that dependency does not need to have a commands block.

A pseudotarget used as a target has an assumed time stamp that is the most recent time stamp of all its dependents. If a pseudotarget has no dependents, the assumed time stamp is the current time. NMAKE uses the assumed time stamp if the pseudotarget appears as a dependent elsewhere in the makefile.

Pseudotargets are useful when you want NMAKE to build more than one target automatically. NMAKE builds only those targets specified on the NMAKE command line, or, when no command-line target is specified, it builds only the first target in the first dependency in the makefile. To tell NMAKE to build multiple targets without having to list them on the command line, write a description block with a dependency containing a pseudotarget and list as its dependents the targets you want to build. Either place this description block first in the makefile or specify the pseudotarget on the NMAKE command line.

Example 1

In the following example, `UPDATE` is a pseudotarget.

```
UPDATE : *.*
    !COPY $** a:\product
```

If `UPDATE` is evaluated, NMAKE copies all files in the current directory to the specified drive and directory.

Example 2

In the following makefile, the pseudotarget `all` builds both PROJECT1.EXE and PROJECT2.EXE if either `all` or no target is specified on the command line. The pseudotarget `setenv` changes the LIB environment variable before the .EXE files are updated:

```
all : setenv project1.exe project2.exe

project1.exe : project1.obj
    LINK project1;

project2.exe : project2.obj
    LINK project2;

setenv :
    set LIB=\project\lib
```

Dependents

The dependents section of the dependency line lists zero or more dependent names. Usually a dependent is a file used to build the target. However, a dependent can be any valid filename, or it can be a pseudotarget. You can specify a path with the filename. Dependents are not case sensitive. Separate each dependent name with one or more spaces or tabs. A single or double colon (**:** or **::**) separates it from the targets section.

Along with dependents you explicitly list in the dependency line, NMAKE can assume an "inferred dependent." An inferred dependent is derived from an inference rule. (For more information, see "Inference Rules" on page 563.) NMAKE considers an inferred dependent to appear earlier in a dependents list than explicit dependents. It builds inferred dependents into the dependency tree. It is important to note that when an inferred dependent in a dependency is out-of-date with respect to a target, NMAKE invokes the commands block associated with the dependency, just as it does with an explicit dependent.

NMAKE uses the dependency tree to make sure that dependents themselves are updated before it updates their targets. If a dependent file doesn't exist, NMAKE looks for a way to build it; if it already exists, NMAKE looks for a way to make sure it is up-to-date. If the dependent is listed as a target in another dependency, or if it is implied as a target in an inference rule, NMAKE checks that the dependent is up-to-date with respect to its own dependents; if the dependent file is out-of-date or doesn't exist, NMAKE executes the commands block for that dependency.

The following example lists three dependents after MYAPP.EXE:

```
myapp.exe : myapp.obj another.obj myapp.def
```

Specifying Search Paths for Dependents

You can specify the directories in which NMAKE should search for a dependent. The syntax for a directory specification is:

{*directory*[[**;***directory*...]]**}***dependent*

Enclose one or more directory names in braces (**{ }**). Separate multiple directories with a semicolon (**;**). No spaces are allowed. You can use a macro to specify part or all of a search path. NMAKE searches the current directory first, then the directories in the order specified. A search path applies only to a single dependent.

Example

The following dependency line contains a directory specification:

```
forward.exe : {\src\alpha;d:\proj}pass.obj
```

The target FORWARD.EXE has one dependent, PASS.OBJ. The directory list specifies two directories. NMAKE first searches for PASS.OBJ in the current directory. If PASS.OBJ isn't there, NMAKE searches the \ SRC \ ALPHA directory, then the D:\ PROJ directory.

Commands

The commands section of a description block or inference rule lists the commands that NMAKE must run if the dependency is out-of-date. You can specify any command or program that can be executed from an MS-DOS command line (with a few exceptions, such as PATH). Multiple commands can appear in a commands block. Each appears on its own line (except as noted in the next section). If a description block doesn't contain any commands, NMAKE looks for an inference rule that matches the dependency. (See "Inference Rules" on page 563.) The following example shows two commands following a dependency line:

```
myapp.exe : myapp.obj another.obj myapp.def
    link myapp another, , NUL, mylib, myapp
    copy myapp.exe c:\project
```

NMAKE displays each command line before it executes it, unless you specify the /S option, the **.SILENT** directive, the **!CMDSWITCHES** directive, or the **@** modifier.

Command Syntax

A command line must begin with one or more spaces or tabs. NMAKE uses this indentation to distinguish between a dependency line and a command line.

Blank lines cannot appear between the dependency line and the commands block. However, a line containing only spaces or tabs can appear; this line is interpreted as a null command, and no error occurs. Blank lines can appear between command lines.

A long command can span several lines if each line ends with a backslash (\ ). A backslash at the end of a line is interpreted as a space on the command line. For example, the LINK command shown in previous examples in this chapter can be expressed as:

```
    link myapp\
another, , NUL, mylib, myapp
```

NMAKE passes the continued lines to the operating system as one long command. A command continued with a backslash must still be within the

operating system's limit on the length of a command line. If any other character, such as a space or tab, follows the backslash, NMAKE interprets the backslash and the trailing characters literally.

You can also place a single command at the end of a dependency line, whether or not other commands follow in the indented commands block. Use a semicolon (**;**) to separate the command from the rightmost dependent, as in:

```
project.obj : project.c project.h ; cl /c project.c
```

Command Modifiers

Command modifiers provide extra control over the commands in a description block. You can use more than one modifier for a single command. Specify a command modifier preceding the command being modified, optionally separated by spaces or tabs. Like a command, a modifier cannot appear at the beginning of a line. It must be preceded by one or more spaces or tabs.

The following describes the three NMAKE command modifiers.

@*command*
: Prevents NMAKE from displaying the command. Any results displayed by commands are not suppressed. Spaces and tabs can appear before the command. By default, NMAKE echoes all makefile commands that it executes. The /S option suppresses display for the entire makefile; the **.SILENT** directive suppresses display for part of the makefile.

–[[*number*]]*command*
: Turns off error checking for the command. Spaces and tabs can appear before the command. By default, NMAKE halts when any command returns an error in the form of a nonzero exit code. This modifier tells NMAKE to ignore errors from the specified command. If the dash is followed by a number, NMAKE stops if the exit code returned by the command is greater than that number. No spaces or tabs can appear between the dash and the number; they must appear between the number and the command. (For more information on using this number, see "Exit Codes from Commands" on page 545.) The /I option turns off error checking for the entire makefile; the **.IGNORE** directive turns off error checking for part of the makefile.

!*command*
: Executes the command for each dependent file if the command preceded by the exclamation point uses the predefined macros **$**** or **$?**. (See "Filename Macros" on page 555.) Spaces and tabs can appear before the command. The **$**** macro represents all dependent files in the dependency line. The **$?** macro refers to all dependent files in the dependency line that have a later time stamp than the target.

Example 1

In the following example, the at sign (**@**) suppresses display of the ECHO command line:

```
sort.exe : sort.obj
    @ECHO Now sorting...
```

The output of the ECHO command is not suppressed.

Example 2

In the following description block, if the program `sample` returns a nonzero exit code, NMAKE does not halt; if `sort` returns an exit code that is greater than 5, NMAKE stops:

```
light.lst : light.txt
    -sample light.txt
    -5 sort light.txt
```

Example 3

The description block

```
print : one.txt two.txt three.txt
    !print $** lpt1:
```

generates the following commands:

```
print one.txt lpt1:
print two.txt lpt1:
print three.txt lpt1:
```

Exit Codes from Commands

NMAKE stops execution if a command or program executed in the makefile encounters an error and returns a nonzero exit code. The exit code is displayed in an NMAKE error message.

You can control how NMAKE behaves when a nonzero exit code occurs by using the /I or /K option, the **.IGNORE** directive, the **!CMDSWITCHES** directive, or the dash (**–**) command modifier.

Another way to use exit codes is during preprocessing. You can run a command or program and test its exit code using the **!IF** preprocessing directive. For more information, see “Executing a Program in Preprocessing” on page 575.

Filename-Parts Syntax

NMAKE provides a syntax that you can use in commands to represent components of the name of the first dependent file. This file is generally the first file listed to the right of the colon in a dependency line. However, if a dependent is implied from an inference rule, NMAKE considers the inferred dependent to be the first dependent file, ahead of any explicit dependents. If more than one inference rule applies, the **.SUFFIXES** list determines which dependent is first. The filename components are the file's drive, path, base name, and extension as you have specified it, not as it exists on disk.

You can represent the complete filename with the following syntax:

%s

For example, if a description block contains

```
sample.exe : c:\project\sample.obj
    LINK %s;
```

NMAKE interprets the command as

```
LINK c:\project\sample.obj;
```

You can represent parts of the complete filename with the following syntax:

%|[[*parts*]]**F**

where *parts* can be zero or more of the following letters, in any order:

Letter	Description
No letter	Complete name
d	Drive
p	Path
f	File base name
e	File extension

Using this syntax, you can represent the full filename specification by `%|F` or by `%|dpfeF`, as well as by `%s`.

Example

The following description block uses filename-parts syntax:

```
sample.exe : c:\project\sample.obj
    LINK %s, a:%|pfF.exe;
```

NMAKE interprets the first representation as the complete filename of the depen-dent. It interprets the second representation as a filename with the same path and base name as the dependent but on the specified drive and with the specified extension. It executes the following command:

```
LINK c:\project\sample.obj, a:\project\sample.exe;
```

Note For another way to represent components of a filename, see "Modifying Filename Macros" on page 556.

Inline Files

NMAKE can create "inline files" in the commands section of a description block or inference rule. An inline file is created on disk by NMAKE and contains text you specify in the makefile. The name of the inline file can be used in commands in the same way as any filename. NMAKE creates the inline file only when it executes the command in which the file is created.

One way to use an inline file is as a response file for another utility such as LINK or LIB. Response files avoid the operating system limit on the maximum length of a command line and automate the specification of input to a utility. Inline files eliminate the need to maintain a separate response file. They can also be used to pass a list of commands to the operating system.

Specifying an Inline File

The syntax for specifying an inline file in a command is:

<<[[*filename*]]

Specify the double angle brackets (<<) on the command line at the location where you want a filename to appear. Because command lines must be indented, the angle brackets cannot appear at the beginning of a line. The angle bracket syntax must be specified literally; it cannot be represented by a macro expansion.

When NMAKE executes the description block, it replaces the inline file specification with the name of the inline file being created. The effect is the same as if a filename was literally specified in the commands section.

The *filename* supplies a name for the inline file. It must immediately follow the angle brackets; no space is permitted. You can specify a path with the filename. No extension is required or assumed. If a file by the same name already exists, NMAKE overwrites it; such a file is deleted if the inline file is temporary. (Temporary inline files are discussed in the next section.)

A name is optional; if you don't specify *filename*, NMAKE gives the inline file a unique name. If *filename* is specified, NMAKE places the file in the directory specified with the name or in the current directory if no path is specified. If *filename* is not specified, NMAKE places the inline file in the directory specified by the TMP environment variable or in the current directory if TMP is not defined. You can reuse a previous inline *filename*; NMAKE overwrites the previous file.

Creating an Inline File

The instructions for creating the inline file begin on the first line after the <<[[*filename*]] specification. The syntax to create the inline file is:

<<[[*filename*]]
inlinetext
.
.
.
<<[[**KEEP** | **NOKEEP**]]

The set of angle brackets marking the end of the inline file must appear at the beginning of a separate line in the makefile. All *inlinetext* before the delimiting angle brackets is placed in the inline file. The text can contain macro expansions and substitutions. Directives and comments are not permitted in an inline file; NMAKE treats them as literal text. Spaces, tabs, and newline characters are treated literally.

The inline file can be temporary or permanent. To retain the file after the end of the NMAKE session, specify **KEEP** immediately after the closing set of angle brackets. If you don't specify a preference, or if you specify **NOKEEP** (the default), the file is temporary. **KEEP** and **NOKEEP** are not case sensitive. The temporary file exists for the duration of the NMAKE session.

It is possible to specify **KEEP** for a file that you do not name; in this case, the NMAKE-generated filename appears in the appropriate directory after the NMAKE session.

Example

The following makefile uses a temporary inline file to clear the screen and then display the contents of the current directory:

```
COMMANDS = cls ^
dir
showdir :
    <<showdir.bat
$(COMMANDS)
<<
```

In this example, the name of the inline file serves as the only command in the description block. This command has the same effect as running a batch file named SHOWDIR.BAT that contains the same commands as those listed in the macro definition.

Reusing an Inline File

After an inline file is created, you can use it more than once. To reuse an inline file in the command in which it is created, you must supply a *filename* for the file where it is defined and first used. You can then reuse the name later in the same command.

You can also reuse an inline file in subsequent commands in the same description block or elsewhere in the makefile. Be sure that the command that creates the inline file executes before all commands that use the file. Regardless of whether you specify **KEEP** or **NOKEEP**, NMAKE keeps the file for the duration of the NMAKE session.

Example

The following makefile creates a temporary LIB response file named LIB.LRF:

```
OBJECTS = add.obj sub.obj mul.obj div.obj
math.lib : $(OBJECTS)
    LIB math.lib @<<lib.lrf
-+$(?: = &^
-+)
listing;
<<
    copy lib.lrf \projinfo\lib.lrf
```

The resulting response file tells LIB which library to use, the commands to execute, and the name of the listing file to produce:

```
-+add.obj &
-+sub.obj &
-+mul.obj &
-+div.obj
listing;
```

The second command in the descripton block tells NMAKE to copy the response file to another directory.

Using Multiple Inline Files

You can specify more than one inline file in a single command line. For each inline specification, specify one or more lines of inline text followed by a closing line containing the delimiter. Begin the second file's text on the line following the delimiting line for the first file.

Example

The following example creates two inline files:

```
target.abc : depend.xyz
    copy <<file1 + <<file2 both.txt
I am the contents of file1.
<<
I am the contents of file2.
<<KEEP
```

This is equivalent to specifying

```
copy file1 + file2 both.txt
```

to concatenate two files, where FILE1 contains

```
I am the contents of file1.
```

and FILE2 contains

```
I am the contents of file2.
```

The **KEEP** keyword tells NMAKE not to delete FILE2. After the NMAKE session, the files FILE2 and BOTH.TXT exist in the current directory.

Macros

Macros offer a convenient way to replace a particular string in the makefile with another string. You can define your own macros or use predefined macros. Macros are useful for a variety of tasks, such as:

- Creating a single makefile that works for several projects. You can define a macro that replaces a dummy filename in the makefile with the specific filename for a particular project.
- Controlling the options NMAKE passes to the compiler or linker. When you specify options in a macro, you can change options throughout the makefile in a single step.
- Specifying paths in an inference rule. (For an example, see Example 3 in “User-Defined Inference Rules” on page 567.)

This section describes user-defined macros, shows how to use a macro, and discusses the macros that have special meaning for NMAKE. It ends by discussing macro substitutions, inherited macros, and precedence rules.

User-Defined Macros

To define a macro, use the following syntax:

macroname=string

The *macroname* can be any combination of letters, digits, and the underscore (_) character, up to 1024 characters. Macro names are case sensitive; NMAKE interprets `MyMacro` and `MYMACRO` as different macro names. The *macroname* can contain a macro invocation. If *macroname* consists entirely of an invoked macro, the macro being invoked cannot be null or undefined.

The *string* can be any sequence of zero or more characters up to 64K–25 (65,510 bytes). A string of zero characters is called a "null string." A string consisting only of spaces, tabs, or both is also considered a null string.

Other syntax rules, such as the use of spaces, apply depending on where you specify the macro; see "Where to Define Macros" on page 552. The *string* can contain a macro invocation.

Example

The following specification defines a macro named `DIR` and assigns to it a string that represents a directory.

```
DIR=c:\objects
```

Special Characters in Macros

Certain characters have special meaning within a macro definition. You use these characters to perform specific tasks. If you want one of these characters to have a literal meaning, you must specify it using a special syntax.

- To specify a comment with a macro definition, place a number sign (#) and the comment after the definition, as in:

  ```
  LINKCMD = link /CO  # Prepare for debugging
  ```

  NMAKE ignores the number sign and all characters up to the next newline character. To specify a literal number sign in a macro, use a caret (^), as in ^#.

- To extend a macro definition to a new line, end the line with a backslash (\ ). The newline character that follows the backslash is replaced with a space when the macro is expanded, as in the following example:

  ```
  LINKCMD = link myapp\
  another, , NUL, mylib, myapp
  ```

  When this macro is expanded, a space separates `myapp` and `another`.

To specify a literal backslash at the end of the line, precede it with a caret (^), as in:

```
exepath = c:\bin^\
```

You can also make a backslash literal by following it with a comment specifier (#). NMAKE interprets a backslash as literal if it is followed by any other character.

- To insert a literal newline character into a macro, end the line with a caret (^). The caret tells NMAKE to interpret the newline character as part of the macro, not as a line break ending the macro definition. The following example defines a macro composed of two operating-system commands separated by a newline character:

  ```
  CMDS = cls^
  dir
  ```

  For an illustration of how this macro can be used, see the first example under "Inline Files" on page 548.

- To specify a literal dollar sign (**$**) in a macro definition, use two dollar signs (**$$**). NMAKE interprets a single dollar sign as the specifier for invoking a macro; see "Using Macros" on page 554.

For information on how to handle other special characters literally, regardless of whether they appear in a macro, see "Using Special Characters as Literals" on page 535.

Where to Define Macros

You can define macros in the makefile, on the command line, in a command file, or in TOOLS.INI. (For more information, see "Precedence Among Macro Definitions" on page 563.) Each macro defined in the makefile or in TOOLS.INI must appear on a separate line. The line cannot start with a space or tab.

When you define a macro in the makefile or in TOOLS.INI, NMAKE ignores any spaces or tabs on either side of the equal sign. The *string* itself can contain embedded spaces. You do not need to enclose *string* in quotation marks (if you do, they become part of the string). The macro name being defined must appear at the beginning of the line. Only one macro can be defined per line. For example, the following macro definition can appear in a makefile or TOOLS.INI:

```
LINKCMD = LINK /MAP
```

Slightly different rules apply when you define a macro on the NMAKE command line or in a command file. The command-line parser treats spaces and tabs as argument delimiters. Therefore, spaces must not precede or follow the equal sign. If *string* contains embedded spaces or tabs, either the string itself or the entire macro must be enclosed in double quotation marks ("). For example, either form of the following command-line macro is allowed:

```
NMAKE "LINKCMD = LINK /MAP"
NMAKE LINKCMD="LINK /MAP"
```

However, the following form of the same macro is not permitted. It contains spaces that are not enclosed by quotation marks:

```
NMAKE LINKCMD = "LINK /MAP"
```

Null Macros and Undefined Macros

An undefined macro is not the same thing as a macro defined to be null. Both kinds of macros expand to a null string. However, a macro defined to be null is still considered to be defined when used with preprocessing directives such as **!IFDEF**. A macro name can be "undefined" in a makefile by using the **!UNDEF** preprocessing directive. (For information on **!IFDEF** and **!UNDEF**, see "Preprocessing Directives" on page 572).

To define a macro to be null:

- In a makefile or TOOLS.INI, specify zero or more spaces between the equal sign (=) and the end of the line, as in the following:

  ```
  LINKOPTIONS =
  ```

- On the command line or in a command file, specify zero or more spaces enclosed in double quotation marks (" "), or specify the entire null definition enclosed in double quotation marks, as in either of the following:

  ```
  LINKOPTIONS=""
  "LINKOPTIONS ="
  ```

To undefine a macro in a makefile or in TOOLS.INF, use the **!UNDEF** preprocessing directive, as in:

```
!UNDEF LINKOPTIONS
```

Using Macros

To use a macro (defined or not), enclose its name in parentheses preceded by a dollar sign (**$**), as follows:

$(*macroname*)

No spaces are allowed. For example, you can use the `LINKCMD` macro defined as

```
LINKCMD = LINK /map
```

by specifying

```
$(LINKCMD)
```

NMAKE replaces the specification `$(LINKCMD)` with `LINK /map`.

If the name you use as a macro has never been defined, or was previously defined but is now undefined, NMAKE treats that name as a null string. No error occurs.

The parentheses are optional if *macroname* is a single character. For example, `$L` is equivalent to `$(L)`. However, parentheses are recommended for consistency and to avoid possible errors.

Example

The following makefile defines and uses three macros:

```
program = sample
L       = LINK
OPTIONS =

$(program).exe : $(program).obj
    $(L) $(OPTIONS) $(program).obj;
```

NMAKE interprets the description block as

```
sample.exe : sample.obj
    LINK sample.obj;
```

NMAKE replaces every occurrence of `$(program)` with `sample`, every instance of `$(L)` with `LINK`, and every instance of `$(OPTIONS)` with a null string.

Special Macros

NMAKE provides several special macros to represent various filenames and commands. One use for these macros is in the predefined inference rules. (For

more information, see “Predefined Inference Rules” on page 567.) Like user-defined macro names, special macro names are case sensitive. For example, NMAKE interprets `CC` and `cc` as different macro names.

The following sections describe the four categories of special macros. The filename macros offer a convenient representation of filenames from a dependency line. The recursion macros allow you to call NMAKE from within your makefile. The command macros and options macros make it convenient for you to invoke the Microsoft language compilers.

Filename Macros

NMAKE provides macros that are predefined to represent filenames. The filenames are as you have specified them in the dependency line and not the full specification of the filenames as they exist on disk. As with all one-character macros, these do not need to be enclosed in parentheses. (The **$$@** and **$**** macros are exceptions to the parentheses rule for macros; they do not require parentheses even though they contain two characters.)

$@
The current target’s full name (path, base name, and extension), as currently specified.

$$@
The current target’s full name (path, base name, and extension), as currently specified. This macro is valid only for specifying a dependent in a dependency line.

$*
The current target’s path and base name minus the file extension.

$**
All dependents of the current target.

$?
All dependents that have a later time stamp than the current target.

$<
The dependent file that has a later time stamp than the current target. You can use this macro only in commands in inference rules.

Example 1

The following example uses the **$?** macro, which represents all dependents that have changed more recently than the target. The **!** command modifier causes NMAKE to execute a command once for each dependent in the list. As a result, the LIB command is executed up to three times, each time replacing a module with a newer version.

```
trig.lib : sin.obj cos.obj arctan.obj
    !LIB trig.lib -+$?;
```

Example 2

In the next example, NMAKE updates a file in another directory by replacing it with a file of the same name from the current directory. The **$@** macro is used to represent the current target's full name.

```
# File in objects directory depends on version in current directory
DIR = c:\objects
$(DIR)\a.obj : a.obj
    COPY a.obj $@
```

Modifying Filename Macros

You can append one of the modifiers in the following table to any of the filename macros to extract part of a filename. If you add one of these modifiers to the macro, you must enclose the macro name and the modifier in parentheses.

Modifier	Resulting Filename Part
D	Drive plus directory
B	Base name
F	Base name plus extension
R	Drive plus directory plus base name

Example 1

Assume that **$@** represents the target C:\SOURCE\PROG\SORT.OBJ. The following table shows the effect of combining each modifier with **$@**:

Macro Reference	Value
$(@D)	C:\SOURCE\PROG
$(@F)	SORT.OBJ
$(@B)	SORT
$(@R)	C:\SOURCE\PROG\SORT

If **$@** has the value SORT.OBJ without a preceding directory, the value of **$(@R)** is SORT, and the value of **$(@D)** is a period (**.**) to represent the current directory.

Example 2

The following example uses the **F** modifier to specify a file of the same name in the current directory:

```
# Files in objects directory depend on versions in current directory
DIR = c:\objects
$(DIR)\a.obj $(DIR)\b.obj $(DIR)\c.obj : $$(@F)
    COPY $(@F) $@
```

Note For another way to represent components of a filename, see "Filename-Parts Syntax" on page 546.

Recursion Macros

There are three macros that you can use when you want to call NMAKE recursively from within a makefile. These macros can make recursion more efficient.

MAKE
Defined as the name which you specified to the operating system when you ran NMAKE; this name is `NMAKE` unless you have renamed the NMAKE.EXE utility file. Use this macro to call NMAKE recursively. The /N command-line option to prevent execution of commands does not prevent this command from executing. It is recommended that you do not redefine **MAKE**.

MAKEDIR
Defined as the current directory when NMAKE was called.

MAKEFLAGS
Defined as the NMAKE options currently in effect. This macro is passed automatically when you call NMAKE recursively. However, specification of this macro when invoking recursion is harmless; thus, you can use older makefiles that specify this macro. You cannot redefine **MAKEFLAGS**. To change the /D, /I, /N, and /S options within a makefile, use the preprocessing directive **!CMDSWITCHES**. (See "Preprocessing Directives" on page 572.) To add other options to the ones already in effect for NMAKE when recursing, specify them as part of the recursion command.

Calling NMAKE Recursively

In a commands block, you can specify a call to NMAKE itself. Either invoke the MAKE macro or specify NMAKE literally. The following NMAKE information is available to the called NMAKE session during recursion:

- Environment-variable macros (see "Inherited Macros" on page 563). To cause all macros to be inherited, specify the /V option.
- The **MAKEFLAGS** macro. If **.IGNORE** (or **!CMDSWITCHES** +I) is set, **MAKEFLAGS** contains an I when it is passed to the recursive call. Likewise, if **.SILENT** (or **!CMDSWITCHES** +S) is set, **MAKEFLAGS** contains an S when passed to the call.
- Macros specified on the command line for the recursive call.
- All information in TOOLS.INI.

Inference rules defined in the makefile are not passed to the called NMAKE session. Settings for **.SUFFIXES** and **.PRECIOUS** are also not inherited. However, you can make **.SUFFIXES**, **.PRECIOUS**, and all inference rules available to the recursive call either by specifying them in TOOLS.INI or by placing them in a file that is specified in an **!INCLUDE** directive in the makefile for each NMAKE session.

Example

The **MAKE** macro is useful for building different versions of a program. The following makefile calls NMAKE recursively to build targets in the \VERS1 and \VERS2 directories.

```
all : vers1 vers2

vers1 :
    cd \vers1
    $(MAKE)
    cd ..

vers2 :
    cd \vers2
    $(MAKE) /F vers2.mak
    cd ..
```

If the dependency containing `vers1` as a target is executed, NMAKE performs the commands to change to the \VERS1 directory and call itself recursively using the MAKEFILE in that directory. If the dependency containing `vers2` as a target is executed, NMAKE changes to the \VERS2 directory and calls itself using the file VERS2.MAK in that directory.

Command Macros

NMAKE predefines several macros to represent commands for Microsoft products. You can use these macros as commands in either a description block or an inference rule; they are automatically used in NMAKE's predefined inference rules. (See "Inference Rules" on page 563.) You can redefine these macros to represent part or all of a command line, including options.

AS
Defined as `ml`, the command to run the Microsoft Macro Assembler

BC
Defined as `bc`, the command to run the Microsoft Basic Compiler

CC
Defined as `cl`, the command to run the Microsoft C Compiler

COBOL
Defined as `cobol`, the command to run the Microsoft COBOL Compiler

CPP
Defined as `cl`, the command to run the Microsoft C++ Compiler

CXX
Defined as `cl`, the command to run the Microsoft C++ Compiler

FOR
Defined as `fl`, the command to run the Microsoft FORTRAN Compiler

PASCAL
Defined as `pl`, the command to run the Microsoft Pascal Compiler

RC
Defined as `rc`, the command to run the Microsoft Resource Compiler

Options Macros

The following macros represent options to be passed to the commands for invoking the Microsoft language compilers. These macros are used automatically in the predefined inference rules. (See “Predefined Inference Rules” on page 567.) By default, these macros are undefined. You can define them to mean the options you want to pass to the compilers, and you can use these macros in commands in description blocks and inference rules. As with all macros, the options macros can be used even if they are undefined; a macro that is undefined or defined to be a null string generates a null string where it is used.

AFLAGS
Passes options to the Microsoft Macro Assembler

BFLAGS
Passes options to the Microsoft Basic Compiler

CFLAGS
Passes options to the Microsoft C Compiler

COBFLAGS
Passes options to the Microsoft COBOL Compiler

CPPFLAGS
Passes options to the Microsoft C++ Compiler

CXXFLAGS
Passes options to the Microsoft C++ Compiler

FFLAGS
Passes options to the Microsoft FORTRAN Compiler

PFLAGS
Passes options to the Microsoft Pascal Compiler

RFLAGS
Passes options to the Microsoft Resource Compiler

Substitution Within Macros

Just as macros allow you to substitute text in a makefile, you can also substitute text within a macro itself. The substitution applies only to the current use of the macro and does not modify the original macro definition. To substitute text within a macro, use the following syntax:

$(*macroname***:***string1***=***string2***)**

Every occurrence of *string1* is replaced by *string2* in the macro *macroname*. Do not put any spaces or tabs before the colon. Spaces that appear after the colon are interpreted as part of the string in which they occur. If *string2* is a null string, all occurrences of *string1* are deleted from the *macroname* macro.

Macro substitution is literal and case sensitive. This means that the case as well as the characters in *string1* must match the target string in the macro exactly, or the substitution is not performed. This also means that *string2* is substituted exactly as it is specified. Because substitution is literal, the strings cannot contain macro expansions.

Example 1

The following makefile illustrates macro substitution:

```
SOURCES = project.c one.c two.c

project.exe : $(SOURCES:.c=.obj)
    LINK $**;
```

The predefined macro **$**** stands for the names of all the dependent files (See “Filename Macros” on page 555.) When this makefile is run, NMAKE executes the following command:

```
LINK project.obj one.obj two.obj;
```

The macro substitution does not alter the `SOURCES` macro definition; if it is used again elsewhere in the makefile, `SOURCES` has its original value as it was defined.

Example 2

If the macro OBJS is defined as

```
OBJS = ONE.OBJ TWO.OBJ THREE.OBJ
```

you can replace each space in the defined value of `OBJS` with a space, followed by a plus sign, followed by a newline character, by using

```
$(OBJS: = +^
)
```

The caret (^) tells NMAKE to treat the end of the line as a literal newline character. The expanded macro after substitution is:

```
ONE.OBJ +
TWO.OBJ +
THREE.OBJ
```

This example is useful for creating response files.

Substitution Within Predefined Macros

You can also substitute text in any predefined macro (except **$$@**) using the same syntax as for other macros.

The command in the following description block makes a substitution within the predefined macro **$@**, which represents the full name of the current target. Note that although **$@** is a single-character macro, when it is used in a substitution, it must be enclosed in parentheses.

```
target.abc : depend.xyz
    echo $(@:targ=blank)
```

NMAKE substitutes `blank` for `targ` in the target, resulting in the string `blanket.abc`. If dependent `depend.xyz` has a later time stamp than target `target.abc`, then NMAKE executes the command

```
echo blanket.abc
```

Environment-Variable Macros

When NMAKE executes, it inherits macro definitions equivalent to every environment variable that existed before the start of the NMAKE session. If a variable such as LIB or INCLUDE has been set in the operating-system environment, you can use its value as if you had specified an NMAKE macro with the same name and value. The inherited macro names are converted to uppercase. Inheritance occurs before preprocessing. The /E option causes macros inherited from environment variables to override any macros with the same name in the makefile.

You can redefine environment-variable macros the same way that you define or redefine other macros. Changing a macro does not change the corresponding environment variable; to change the variable, use a SET command. Also, using the SET command to change an environment variable in an NMAKE session does not change the corresponding macro; to change the macro, use a macro definition.

If an environment variable has not been set in the operating-system environment, it cannot be set using a macro definition. However, you can use a SET command in the NMAKE session to set the variable. The variable is then in effect for the rest of the NMAKE session unless redefined or cleared by a later SET command. A SET definition that appears in a makefile does not create a corresponding macro for that variable name; if you want a macro for an environment variable that is created during an NMAKE session, you must explicitly define the macro in addition to setting the variable.

If an environment variable is defined as a string that would be syntactically incorrect in a makefile, NMAKE does not create a macro from that variable. No warning is generated.

Warning If an environment variable contains a dollar sign (**$**), NMAKE interprets it as the beginning of a macro invocation. The resulting macro expansion can cause unexpected behavior and possibly an error.

Example

The following makefile redefines the environment-variable macro called `LIB`:

```
LIB = c:\tools\lib

sample.exe : sample.obj
    LINK sample;
```

No matter what value the environment variable LIB had before, it has the value `c:\tools\lib` when NMAKE executes the LINK command in this description block. Redefining the inherited macro does not affect the original environment variable; when NMAKE terminates, LIB still has its original value.

If LIB is not defined before the NMAKE session, the LIB macro definition in the preceding example does not set a LIB environment variable for the LINK command. To do this, use the following makefile:

```
sample.exe : sample.obj
    SET LIB=c:\tools.lib
    LINK sample;
```

Inherited Macros

When NMAKE is called recursively, the only macros that are inherited by the called NMAKE are those defined on the command line or in environment variables. Macros defined in the makefile are not inherited when NMAKE is called recursively. There are several ways to pass macros to a recursive NMAKE session:

- Run NMAKE with the /V option. This option causes all macros to be inherited by the recursively called NMAKE. You can use this option on the NMAKE command for the entire session, or you can specify it in a command for a recursive NMAKE call to affect just the specified recursive session.
- Use the SET command before the recursive call to set an environment variable before the called NMAKE session.
- Define a macro on the command line for the recursive call.
- Define a macro in the TOOLS.INI file. Each time NMAKE is recursively called, it reads TOOLS.INI.

Precedence Among Macro Definitions

If you define the same macro name in more than one place, NMAKE uses the macro with the highest precedence. The precedence from highest to lowest is as follows:

1. A macro defined on the command line
2. A macro defined in a makefile or include file
3. An inherited environment-variable macro
4. A macro defined in the TOOLS.INI file
5. A predefined macro, such as **AS** or **CC**

The /E option causes macros inherited from environment variables to override any macros with the same name in the makefile. The **!UNDEF** directive in a makefile overrides a macro defined on the command line.

Inference Rules

Inference rules are templates that define how a file with one extension is created from a file with another extension. NMAKE uses inference rules to supply commands for updating targets and to infer dependents for targets. In the dependency tree, inference rules cause targets to have inferred dependents as well as explicitly specified dependents; see “Inferred Dependents” on page 569.

The **.SUFFIXES** list determines priorities for applying inference rules; see “Dot Directives” on page 570.

Inference rules provide a convenient shorthand for common operations. For instance, you can use an inference rule to avoid repeating the same command in several description blocks. You can define your own inference rules or use predefined inference rules. Inference rules can be specified in the makefile or in TOOLS.INI.

Inference rules can be used in the following situations:

- If NMAKE encounters a description block that has no commands, it checks the **.SUFFIXES** list and the files in the current or specified directory and then searches for an inference rule that matches the extensions of the target and an existing dependent file with the highest possible **.SUFFIXES** priority.
- If a dependent file doesn’t exist and is not listed as a target in another description block, NMAKE looks for an inference rule that shows how to create the missing dependent from another file with the same base name.
- If a target has no dependents and its description block has no commands, NMAKE can use an inference rule to create the target.
- If a target is specified on the command line and there is no makefile (or no mention of the target in the makefile), inference rules are used to build the target.

If a target is used in more than one single-colon dependency, an inference rule might not be applied as expected; see “Accumulating Targets in Dependencies” on page 539.

Inference Rule Syntax

To define an inference rule, use the following syntax:

.*fromext***.***toext***:**
    commands

The first line lists two extensions: *fromext* represents the extension of a dependent file, and *toext* represents the extension of a target file. Extensions are not case sensitive. Macros can be invoked to represent *fromext* and *toext*; the macros are expanded during preprocessing.

The period (**.**) preceding *fromext* must appear at the beginning of the line. The colon (**:**) can be preceded by zero or more spaces or tabs; it can be followed only by spaces or tabs, a semicolon (**;**) to specify a command, a number sign (**#**) to specify a comment, or a newline character. No other spaces are allowed.

The rest of the inference rule gives the commands to be run if the dependency is out-of-date. Use the same rules for commands in inference rules as in description blocks. (See “Commands” on page 543.)

An inference rule can be used only when a target and dependent have the same base name. You cannot use a rule to match multiple targets or dependents. For example, you cannot define an inference rule that replaces several modules in a library because all but one of the modules must have a different base name from the target library.

Inference rules can exist only for dependents with extensions that are listed in the **.SUFFIXES** directive. (For information on **.SUFFIXES**, see “Dot Directives” on page 570.) If an out-of-date dependency does not have a commands block, and if the **.SUFFIXES** list contains the extension of the dependent, NMAKE looks for an inference rule matching the extensions of the target and of an existing file in the current or specified directory. If more than one rule matches existing dependent files, NMAKE uses the order of the **.SUFFIXES** list to determine which rule to invoke. Priority in the list descends from left to right. NMAKE may invoke a rule for an inferred dependent even if an explicit dependent is specified; for more information, see “Inferred Dependents” on page 569.

Inference rules tell NMAKE how to build a target specified on the command line if no makefile is provided or if the makefile does not have a dependency containing the specified target. When a target is specified on the command line and NMAKE cannot find a description block to run, it looks for an inference rule to tell it how to build the target. You can run NMAKE without a makefile if the inference rules that are predefined or defined in TOOLS.INI are all you need for your build.

Inference Rule Search Paths

The inference-rule syntax described previously tells NMAKE to look for the specified files in the current directory. You can also specify directories to be searched by NMAKE when it looks for files. An inference rule that specifies paths has the following syntax:

{*frompath***}.***fromext***{***topath***}.***toext***:**
    commands

No spaces are allowed. The *frompath* directory must match the directory specified for the dependent file; similarly, *topath* must match the target’s directory specification. For NMAKE to apply an inference rule to a dependency, the paths in the dependency line must match the paths specified in the inference rule exactly. For example, if the current directory is called PROJ, the inference rule

```
{..\proj}.exe{..\proj}.obj:
```

does not apply to the dependency

```
project1.exe : project1.obj
```

If you use a path on one extension in the inference rule, you must use paths on both. You can specify the current directory by either a period (**.**) or an empty pair of braces (**{}**).

You can specify only one path for each extension in an inference rule. To specify more than one path, you must create a separate inference rule for each path.

Macros can be invoked to represent *frompath* and *topath*; the macros are expanded during preprocessing.

User-Defined Inference Rules

The following examples illustrate several ways to write inference rules.

Example 1

The following makefile contains an inference rule and a minimal description block:

```
.c.obj:
    cl /c $<

sample.obj :
```

The inference rule tells NMAKE how to build a .OBJ file from a .C file. The predefined macro **$<** represents the name of a dependent that has a later time stamp than the target. The description block lists only a target, SAMPLE.OBJ; there is no dependent or command. However, given the target's base name and extension, plus the inference rule, NMAKE has enough information to build the target.

After checking to be sure that `.c` is one of the extensions in the **.SUFFIXES** list, NMAKE looks for a file with the same base name as the target and with the .C extension. If SAMPLE.C exists (and no files with higher-priority extensions exist), NMAKE compares its time to that of SAMPLE.OBJ. If SAMPLE.C has changed more recently, NMAKE compiles it using the CL command listed in the inference rule:

```
    cl /c sample.c
```

Example 2

The following inference rule compares a .C file in the current directory with the corresponding .OBJ file in another directory:

```
{.}.c{c:\objects}.obj:
    cl /c $<;
```

The path for the .C file is represented by a period. A path for the dependent extension is required because one is specified for the target extension.

This inference rule matches a dependency line containing the same combination of paths, such as:

```
c:\objects\test.obj : test.c
```

This rule does not match a dependency line such as:

```
test.obj : test.c
```

In this case, NMAKE uses the predefined inference rule for .c.obj when building the target.

Example 3

The following inference rule uses macros to specify paths in an inference rule:

```
C_DIR = proj1src
OBJ_DIR = proj1obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $
```

If the macros are redefined, NMAKE uses the definition that is current at that point during preprocessing. To reuse an inference rule with different macro definitions, you must repeat the rule after the new definition:

```
C_DIR = proj1src
OBJ_DIR = proj1obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $<
C_DIR = proj2src
OBJ_DIR = proj2obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $<
```

Predefined Inference Rules

NMAKE provides predefined inference rules containing commands for creating object, executable, and resource files. Table 16.1 describes the predefined inference rules.

Table 16.1 Predefined Inference Rules

Rule	Command	Default Action
.asm.exe	$(AS) $(AFLAGS) $**.asm	ML $**.ASM
.asm.obj	$(AS) $(AFLAGS) /c $**.asm	ML /c $**.ASM
.c.exe	$(CC) $(CFLAGS) $**.c	CL $**.C
.c.obj	$(CC) $(CFLAGS) /c $**.c	CL /c $**.C
.cpp.exe	$(CPP) $(CPPFLAGS) $**.cpp	CL $**.CPP

Table 16.1 Predefined Inference Rules (*continued*)

Rule	Command	Default Action
.cpp.obj	$(CPP) $(CPPFLAGS) /c $**.cpp	CL /c $**.CPP
.cxx.exe	$(CXX) $(CXXFLAGS) $**.cxx	CL $**.CXX
.cxx.obj	$(CXX) $(CXXFLAGS) /c $**.cxx	CL /c $**.CXX
.bas.obj	$(BC) $(BFLAGS) $**.bas;	BC $**.BAS;
.cbl.exe	$(COBOL) $(COBFLAGS) $**.cbl, $**.exe;	COBOL $**.CBL, $**.EXE;
.cbl.obj	$(COBOL) $(COBFLAGS) $**.cbl;	COBOL $**.CBL;
.for.exe	$(FOR) $(FFLAGS) $**.for	FL $**.FOR
.for.obj	$(FOR) /c $(FFLAGS) $**.for	FL /c $**.FOR
.pas.exe	$(PASCAL) $(PFLAGS) $**.pas	PL $**.PAS
.pas.obj	$(PASCAL) /c $(PFLAGS) $**.pas	PL /c $**.PAS
.rc.res	$(RC) $(RFLAGS) /r $**	RC /r $**

For example, assume you have the following makefile:

```
sample.exe :
```

This description block lists a target without any dependents or commands. NMAKE looks at the target's extension (.EXE) and searches for an inference rule that describes how to create an .EXE file. Table 16.1 shows that more than one inference rule exists for building an .EXE file. NMAKE uses the order of the extensions appearing in the **.SUFFIXES** list to determine which rule to invoke. It then looks in the current or specified directory for a file that has the same base name as the target `sample` and one of the extensions in the **.SUFFIXES** list; it checks the extensions one by one until it finds a matching dependent file in the directory.

For example, if a file called SAMPLE.ASM exists, NMAKE applies the `.asm.exe` inference rule. If both SAMPLE.C and SAMPLE.ASM exist, and if `.c` appears before `.asm` in the **.SUFFIXES** list, NMAKE uses the `.c.exe` inference rule to compile SAMPLE.C and links the resulting file SAMPLE.OBJ to create SAMPLE.EXE.

Note By default, the options macros (**AFLAGS**, **CFLAGS**, and so on) are undefined. As explained in "Using Macros" on page 554, this causes no problem; NMAKE replaces an undefined macro with a null string. Because the predefined options macros are included in the inference rules, you can define these macros and have their assigned values passed automatically to the predefined inference rules.

Inferred Dependents

NMAKE can assume an "inferred dependent" for a target if there is an applicable inference rule. An inference rule is applicable if:

- The *toext* in the rule matches the extension of the target being evaluated.
- The *fromext* in the rule matches the extension of a file that has the same base name as the target and that exists in the current or specified directory.
- The *fromext* is in the **.SUFFIXES** list.
- No other *fromext* in a matching rule is listed in **.SUFFIXES** with a higher priority.
- No explicitly specified dependent has a higher priority extension.

If an existing dependent matches an inference rule and has an extension with a higher **.SUFFIXES** priority, NMAKE does not infer a dependent.

NMAKE does not necessarily execute the commands block in an inference rule for an inferred dependent. If the target's description block contains commands, NMAKE executes the description block's commands and not the commands in the inference rule. The effect of an inferred dependent is illustrated in the following example:

```
project.obj :
    cl /Zi /c project.c
```

If a makefile contains this description block and if the current directory contains a file named PROJECT.C and no other files, NMAKE uses the predefined inference rule for `.c.obj` to infer the dependent `project.c`. It does not execute the predefined rule's command, `cl /c project.c`. Instead, it runs the command specified in the makefile.

Inferred dependents can cause unexpected side effects. In the following examples, assume that both PROJECT.ASM and PROJECT.C exist and that **.SUFFIXES** contains the default setting. If the makefile contains

```
project.obj : project.c
```

NMAKE infers the dependent `project.asm` ahead of `project.c` because **.SUFFIXES** lists `.asm` before `.c` and because a rule for `.asm.obj` exists. If either PROJECT.ASM or PROJECT.C is out-of-date, NMAKE executes the commands in the rule for `.asm.obj`.

However, if the dependency in the preceding example is followed by a commands block, NMAKE executes those commands and not the commands in the inference rule for the inferred dependent.

Another side effect occurs because NMAKE builds a target if it is out-of-date with respect to any of its dependents, whether explicitly specified or inferred. For example, if PROJECT.OBJ is up-to-date with respect to PROJECT.C but not with respect to PROJECT.ASM, and if the makefile contains

```
project.obj : project.c
    cl /Zi /c project.c
```

NMAKE infers the dependent `project.asm` and updates the target using the command specified in this description block.

Precedence Among Inference Rules

If the same inference rule is defined in more than one place, NMAKE uses the rule with the highest precedence. The precedence from highest to lowest is as follows:

1. An inference rule defined in the makefile. If more than one rule is defined, the last rule applies.
2. An inference rule defined in the TOOLS.INI file. If more than one rule is defined, the last rule applies.
3. A predefined inference rule.

User-defined inference rules always override predefined inference rules. NMAKE uses a predefined inference rule only if no user-defined inference rule exists for a given target and dependent.

If two inference rules match a target's extension and a dependent is not specified, NMAKE uses the inference rule whose dependent's extension appears first in the **.SUFFIXES** list.

Directives

NMAKE provides several ways to control the NMAKE session through dot directives and preprocessing directives. Directives are instructions to NMAKE that are placed in the makefile or in TOOLS.INI. NMAKE interprets dot directives and preprocessing directives and applies the results to the makefile before processing dependencies and commands.

Dot Directives

Dot directives must appear outside a description block and must appear at the beginning of a line. Dot directives begin with a period (**.**) and are followed by a colon (**:**). Spaces and tabs can precede and follow the colon. These directive names are case sensitive and must be uppercase.

.IGNORE :

Ignores nonzero exit codes returned by programs called from the makefile. By default, NMAKE halts if a command returns a nonzero exit code. This directive affects the makefile from the place it is specified to the end of the file. To turn it off again, use the **!CMDSWITCHES** preprocessing directive. To ignore the exit code for a single command, use the dash (**–**) command modifier. To ignore exit codes for an entire file, invoke NMAKE with the /I option.

.PRECIOUS : *targets*

Tells NMAKE not to delete *targets* if the commands that build them are interrupted. This directive has no effect if a command is interrupted and handles the interrupt by deleting the file. Separate the target names with one or more spaces or tabs. By default, NMAKE deletes the target if building was interrupted by CTRL+C or CTRL+BREAK. Multiple specifications are cumulative; each use of **.PRECIOUS** applies to the entire makefile.

.SILENT :

Suppresses display of the command lines as they are executed. By default, NMAKE displays the commands it invokes. This directive affects the makefile from the place it is specified to the end of the file. To turn it off again, use the **!CMDSWITCHES** preprocessing directive. To suppress display of a single command line, use the **@** command modifier. To suppress the command display for an entire file, invoke NMAKE with the /S option.

.SUFFIXES : *list*

Lists file suffixes (extensions) for NMAKE to try to match when it attempts to apply an inference rule. (For details about using **.SUFFIXES**, see “Inference Rules” on page 563.) The list is predefined as follows:

```
.SUFFIXES : .exe .obj .asm .c .cpp .cxx .bas .cbl .for .pas .res .rc
```

To add additional suffixes to the end of the list, specify

.SUFFIXES : *suffixlist*

where *suffixlist* is a list of the additional suffixes, separated by one or more spaces or tabs. To clear the list, specify

```
.SUFFIXES :
```

without extensions. To change the list order or to specify an entirely new list, you must clear the list and specify a new setting. To see the current setting, run NMAKE with the /P option.

Preprocessing Directives

NMAKE preprocessing directives are similar to compiler preprocessing directives. You can use several of the directives to conditionally process the makefile. With other preprocessing directives you can display error messages, include other files, undefine a macro, and turn certain options on or off. NMAKE reads and executes the preprocessing directives before processing the makefile as a whole.

Preprocessing directives begin with an exclamation point (**!**), which must appear at the beginning of the line. Zero or more spaces or tabs can appear between the exclamation point and the directive keyword; this allows indentation for readability. These directives (and their keywords and operators) are not case sensitive.

!CMDSWITCHES {+|–}*opt*...

Turns on or off one or more options. (For descriptions of options, see page 529.) Specify an operator, either a plus sign (+) to turn options on or a minus sign (–) to turn options off, followed by one or more letters representing options. Letters are not case sensitive. Do not specify the slash (/). Separate the directive from the operator by one or more spaces or tabs; no space can appear between the operator and the options. To turn on some options and turn off other options, use separate specifications of the **!CMDSWITCHES** directives.

All options with the exception of /F, /HELP, /NOLOGO, /X, and /? can appear in **!CMDSWITCHES** specifications in TOOLS.INI. In a makefile, only the letters D, I, N, and S can be specified. If **!CMDSWITCHES** is specified within a description block, the changes do not take effect until the next description block. This directive updates the **MAKEFLAGS** macro; the changes are inherited during recursion.

!ERROR *text*

Displays *text* to standard error in the message for error U1050, then stops the NMAKE session. This directive stops the build even if /K, /I, **.IGNORE**, **!CMDSWITCHES**, or the dash (–) command modifier is used. Spaces or tabs before *text* are ignored.

!MESSAGE *text*

Displays *text* to standard output, then continues the NMAKE session. Spaces or tabs before *text* are ignored.

!INCLUDE [[<]]*filename*[[>]]
Reads and evaluates the file *filename* as a makefile before continuing with the current makefile. NMAKE first looks for *filename* in the current directory if *filename* is specified without a path; if a path is specified, NMAKE looks in the specified directory. Next, if the **!INCLUDE** directive is itself contained in a file that is included, NMAKE looks for *filename* in the parent file's directory; this search is recursive, ending with the original makefile's directory. Finally, if *filename* is enclosed by angle brackets (< >), NMAKE searches in the directories specified by the **INCLUDE** macro. The **INCLUDE** macro is initially set to the value of the INCLUDE environment variable.

!IF *constantexpression*
Processes the statements between the **!IF** and the next **!ELSE** or **!ENDIF** if *constantexpression* evaluates to a nonzero value.

!IFDEF *macroname*
Processes the statements between the **!IFDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is defined. NMAKE considers a macro with a null value to be defined.

!IFNDEF *macroname*
Processes the statements between the **!IFNDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is not defined.

!ELSE [[**IF** *constantexpression*|**IFDEF** *macroname*|**IFNDEF** *macroname*]]
Processes the statements between the **!ELSE** and the next **!ENDIF** if the preceding **!IF**, **!IFDEF**, or **!IFNDEF** statement evaluated to zero. The optional keywords give further control of preprocessing.

!ELSEIF
Synonym for **!ELSE IF**.

!ELSEIFDEF
Synonym for **!ELSE IFDEF**.

!ELSEIFNDEF
Synonym for **!ELSE IFNDEF**.

!ENDIF
Marks the end of an **!IF**, **!IFDEF**, or **!IFNDEF** block. Anything following **!ENDIF** on the same line is ignored.

!UNDEF *macroname*
Undefines a macro by removing *macroname* from NMAKE's symbol table. (For more information, see "Null Macros and Undefined Macros" on page 553.)

Example

The following set of directives

```
!IF
!ELSE
!   IF
!   ENDIF
!ENDIF
```

is equivalent to the set of directives

```
!IF
!ELSE IF
!ENDIF
```

Expressions in Preprocessing

The *constantexpression* used with the **!IF** or **!ELSE IF** directives can consist of integer constants, string constants, or program invocations. You can group expressions by enclosing them in parentheses. NMAKE treats numbers as decimals unless they start with 0 (octal) or 0x (hexadecimal).

Expressions in NMAKE use C-style signed long integer arithmetic; numbers are represented in 32-bit two's-complement form and are in the range –2147483648 to 2147483647.

Two unary operators evaluate a condition and return a logical value of true (1) or false (0):

DEFINED (*macroname*)
: Evaluates to true if *macroname* is defined. In combination with the **!IF** or **!ELSE IF** directives, this operator is equivalent to the **!IFDEF** or **!ELSE IFDEF** directives. However, unlike these directives, **DEFINED** can be used in complex expressions using binary logical operators.

EXIST (*path*)
: Evaluates to true if *path* exists. **EXIST** can be used in complex expressions using binary logical operators. If *path* contains spaces (allowed in some file systems), enclose it in double quotation marks.

Integer constants can use the unary operators for numerical negation (**–**), one's complement (**~**), and logical negation (**!**).

Constant expressions can use any binary operator listed in Table 16.2. To compare two strings, use the equality (**==**) operator and the inequality (**!=**) operator. Enclose strings in double quotation marks.

Table 16.2 Binary Operators for Preprocessing

Operator	Description
+	Addition
–	Subtraction

*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
&&	Logical AND
||	Logical OR

Table 16.2 Binary Operators for Preprocessing (*continued*)

Operator	Description
<<	Left shift
>>	Right shift
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Example

The following example shows how preprocessing directives can be used to control whether the linker inserts debugging information into the .EXE file:

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe : winner.obj
!IF DEFINED(debug)
!    IF "$(debug)"=="y"
         LINK /CO winner.obj;
!    ELSE
         LINK winner.obj;
!    ENDIF
!ELSE
!    ERROR Macro named debug is not defined.
!ENDIF
```

In this example, the **!INCLUDE** directive inserts the INFRULES.TXT file into the makefile. The **!CMDSWITCHES** directive sets the /D option, which displays the time stamps of the files as they are checked. The **!IF** directive checks to see if the macro `debug` is defined. If it is defined, the next **!IF** directive checks to see if it is set to `y`. If it is, NMAKE reads the LINK command with the /CO option; otherwise, NMAKE reads the LINK command without /CO. If the `debug` macro is not defined, the **!ERROR** directive prints the specified message and NMAKE stops.

Executing a Program in Preprocessing

You can invoke a program or command from within NMAKE and use its exit code during preprocessing. NMAKE executes the command during preprocessing, and it replaces the specification in the makefile with the command's exit code. A nonzero exit code usually indicates an error. You can use this value in an expression to control preprocessing.

Specify the command, including any arguments, within brackets (**[ ]**). You can use macros in the command specification; NMAKE expands the macro before executing the command.

Example

The following part of a makefile tests the space on disk before continuing the NMAKE session:

```
!IF [c:\util\checkdsk] != 0
!    ERROR Not enough disk space; NMAKE terminating.
!ENDIF
```

Sequence of NMAKE Operations

When you write a complex makefile, it can be helpful to know the sequence in which NMAKE performs operations. This section describes those operations and their order.

When you run NMAKE from the command line, NMAKE's first task is to find the makefile:

1. If the /F option is used, NMAKE searches for the filename specified in the option. If NMAKE cannot find that file, it returns an error.
2. If the /F option is not used, NMAKE looks for a file named MAKEFILE in the current directory. If there are targets on the command line, NMAKE builds them according to the instructions in MAKEFILE. If there are no targets on the command line, NMAKE builds only the first target it finds in MAKEFILE.
3. If NMAKE cannot find MAKEFILE, NMAKE looks for target files on the command line and attempts to build them using inference rules (either defined by the user in TOOLS.INI or predefined by NMAKE). If no target is specified, NMAKE returns an error.

NMAKE then assigns macro definitions with the following precedence (highest to lowest):

1. Macros defined on the command line
2. Macros defined in a makefile or include file
3. Inherited macros
4. Macros defined in the TOOLS.INI file
5. Predefined macros (such as **CC** and **RFLAGS**)

Macro definitions are assigned first in order of priority and then in the order in which NMAKE encounters them. For example, a macro defined in an include file overrides a macro with the same name from the TOOLS.INI file. Note that a macro within a makefile can be redefined; a macro is valid from the point it is defined until it is redefined or undefined.

NMAKE also assigns inference rules, using the following precedence (highest to lowest):

1. Inference rules defined in a makefile or include file
2. Inference rules defined in the TOOLS.INI file
3. Predefined inference rules (such as .asm.obj)

You can use command-line options to change some of these priorities.

- The /E option allows macros inherited from the environment to override macros defined in the makefile.
- The /R option tells NMAKE to ignore macros and inference rules that are defined in TOOLS.INI or are predefined.

Next, NMAKE evaluates any preprocessing directives. If an expression for conditional preprocessing contains a program in brackets (**[ ]**), the program is invoked during preprocessing and the program's exit code is used in the expression. If an **!INCLUDE** directive is specified for a file, NMAKE preprocesses the included file before continuing to preprocess the rest of the makefile. Preprocessing determines the final makefile that NMAKE reads.

NMAKE is now ready to update the targets. If you specified targets on the command line, NMAKE updates only those targets. If you did not specify targets on the command line, NMAKE updates only the first target in the makefile. If you specify a pseudotarget, NMAKE always updates the target. If you use the /A option, NMAKE always updates the target, even if the file is not out-of-date.

NMAKE updates a target by comparing its time stamp to the time stamp of each dependent of that target. A target is out-of-date if any dependent has a later time stamp; if the /B option is specified, a target is out-of-date if any dependent has a later or equal time stamp.

If the dependents of the targets are themselves out-of-date or do not exist, NMAKE updates them first. If the target has no explicit dependent, NMAKE looks for an inference rule that matches the target. If a rule exists, NMAKE updates the target using the commands given with the inference rule. If more than one rule applies to the target, NMAKE uses the priority in the **.SUFFIXES** list to determine which inference rule to use.

NMAKE normally stops processing the makefile when a command returns a nonzero exit code. In addition, if NMAKE cannot tell whether the target was built successfully, it deletes the target. The /I command-line option, **.IGNORE** directive, **!CMDSWITCHES** directive, and dash (–) command modifier all tell NMAKE to ignore error codes and attempt to continue processing. The /K option tells NMAKE to continue processing unrelated parts of the build if an error occurs. The **.PRECIOUS** directive prevents NMAKE from deleting a partially created target if you interrupt the build with CTRL+C or CTRL+BREAK. You can document errors by using the **!ERROR** directive to print descriptive text. The directive causes NMAKE to print some text, then stop the build.

A Sample NMAKE Makefile

The following example illustrates many of NMAKE's features. The makefile creates an executable file from C-language source files:

```
# This makefile builds SAMPLE.EXE from SAMPLE.C,
# ONE.C, and TWO.C, then deletes intermediate files.

CFLAGS   = /c /AL /Od $(CODEVIEW)  # controls compiler options
LFLAGS   = /CO                     # controls linker options
CODEVIEW = /Zi                     # controls debugging information

OBJS = sample.obj one.obj two.obj

all : sample.exe

sample.exe : $(OBJS)
    link $(LFLAGS) @<<sample.lrf
$(OBJS: =+^
)
sample.exe
sample.map;
<<KEEP

sample.obj : sample.c sample.h common.h
     CL $(CFLAGS) sample.c

one.obj : one.c one.h common.h
     CL $(CFLAGS) one.c

two.obj : two.c two.h common.h
     CL $(CFLAGS) two.c
```

```
clean :
    -del *.obj
    -del *.map
    -del *.lrf
```

Assume that this makefile is named SAMPLE.MAK. To invoke it, enter

```
NMAKE /F SAMPLE.MAK all clean
```

NMAKE builds SAMPLE.EXE and deletes intermediate files.

Here is how the makefile works. The `CFLAGS`, `CODEVIEW`, and `LFLAGS` macros define the default options for the compiler, linker, and inclusion of debugging information. You can redefine these options from the command line to alter or delete them. For example,

```
NMAKE /F SAMPLE.MAK CODEVIEW= CFLAGS= all clean
```

creates an .EXE file that does not contain debugging information.

The `OBJS` macro specifies the object files that make up the executable file SAMPLE.EXE, so they can be reused without having to type them again. Their names are separated by exactly one space so that the space can be replaced with a plus sign (+) and a carriage return in the link response file. (This is illustrated in the second example in “Substitution Within Macros” on page 560.)

The `all` pseudotarget points to the real target, `sample.exe`. If you do not specify any target on the command line, NMAKE ignores the `clean` pseudotarget but still builds `all` because `all` is the first target in the makefile.

The dependency line containing the target `sample.exe` makes the object files specified in `OBJS` the dependents of `sample.exe`. The command section of the block contains only link instructions. No compilation instructions are given since they are given explicitly later in the file. (You can also define an inference rule to specify how an object file is to be created from a C source file.)

The `link` command is unusual because the LINK parameters and options are not passed directly to LINK. Rather, an inline response file is created containing these elements. This eliminates the need to maintain a separate link response file.

The next three dependencies define the relationship of the source code to the object files. The .H (header or include) files are also dependents since any changes to them also require recompilation.

The `clean` pseudotarget deletes unneeded files after a build. The dash (–) command modifier tells NMAKE to ignore errors returned by the deletion commands. If you want to save any of these files, don’t specify `clean` on the command line; NMAKE then ignores the `clean` pseudotarget.

NMAKE Exit Codes

NMAKE returns an exit code to the operating system or the calling program. A value of 0 indicates execution of NMAKE with no errors. Warnings return exit code 0.

Code	Meaning
0	No error
1	Incomplete build (issued only when /K is used)
2	Program error, possibly due to one of the following: • A syntax error in the makefile • An error or exit code from a command • An interruption by the user
4	System error—out of memory
255	Target is not up-to-date (issued only when /Q is used)