NMAKE.WRI File

Documentation for the NMAKE Utility for Microsoft(R) Visual C++, Version 1.0

Managing Projects with NMAKE

This file describes the Microsoft Program Maintenance Utility (NMAKE) version 1.30. 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 Visual Workbench to build your project, it handles the build process for you. You may want to read this file if you intend to build your program outside of Visual Workbench, if you want to understand existing makefiles, or if you want to use an external project in Visual Workbench.

This version of NMAKE is a 32-bit MS-DOS-extended program. It requires an 80386 or higher processor running in protected mode. The file DOSXNT.EXE must be in the same directory as NMAKE or on the path. In addition, for NMAKE to run with the Microsoft Windows operating system, DOSXNT.386 must be specified in a DEVICE statement in the [386Enh] section of SYSTEM.INI.

New and Updated Features

NMAKE version 1.30 has the following new or updated features:

- NMAKE's behavior during recursion has changed. Recursive calls to NMAKE do not differ from the original call. The MAKEFLAGS macro is no longer passed automatically. There is no special inheritance of macros or options. The /N option no longer displays commands in recursive calls to NMAKE. For more information, see "Calling NMAKE Recursively" later in this file.
- NMAKE is MS-DOS extended and requires the services described above. It cannot run with real-mode MS-DOS.
- The /HELP option now gives the same result as the /? option.
- The /M and /V options have been removed.

The following features are included in NMAKE versions 1.30:

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

Overview

NMAKE works by comparing the time stamps of a "target" file and its "dependent" files. 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. A target file 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 want to execute. A dependent file 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 under "Command-Line Options" later in this file.)

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.

NMAKE's main purpose is to help you build programs quickly and easily. However, NMAKE is not limited to compiling and linking; it 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.

In this file, the term "build," as in building a target, means 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" later in this file.

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" later in this file.)

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
```

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. See "The TOOLS.INI File" later in this file. For information on specifying options with the **!CMDSWITCHES** directive, see "Preprocessing Directives" later in this file.

/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" later in this file.

/E

Causes environment variables to override macro definitions in the makefile. See "Macros" later in this file.

/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

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" and "Dot Directives" later in this file. To set or clear /I for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" later in this file. 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.

/N

Displays but does not execute the commands that would be executed by the build. Preprocessing commands are executed. Commands in recursive calls to NMAKE are not automatically displayed. 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" later in this file.

/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" and "Dot Directives" later in this file. To set or clear /S for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" later in this file.

/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.

/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.

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" later in this file.

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 "Preprocessing Directives" later in this file.

Macros and inference rules appearing in TOOLS.INI can be overridden. See "Precedence among Macro Definitions" and "Precedence among Inference Rules" later in this file.

NMAKE reads information in TOOLS.INI before it reads makefile information. 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" later in this file.

Example

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

```
[NMAKE]
; macros
CC     = qcl
CFLAGS = /Gc /Gs /W3 /Oat
; inference rule
.c.obj:
    $(CC) /Zi /c $(CFLAGS) $*.c
```

NMAKE reads and applies the lines following `[NMAKE]`. The example redefines the macro **CC** to invoke the Microsoft QuickC Compiler, defines the macro **CFLAGS**, and redefines the inference rule for making .OBJ files from .C sources. These NMAKE features are explained throughout this file.

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 file 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" later in this file.

In a command, a percent symbol (**%**) can represent the beginning of a file specifier. (See "Filename-Parts Syntax" later in this file.) 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" later in this file.

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 : *.c
    cl *.c /Feproject.exe
```

NMAKE expands the *.c in the dependency line and looks at all files having the .C extension in the current directory. If any .C file is out-of-date, the CL 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 number sign must appear at the beginning of the line. Otherwise, the number sign 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 comment out a command, insert a number sign 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 number sign, 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" earlier in this file.

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:

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

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" later in this file.

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" later in this file. 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 them-selves 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" later in this file.)

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. Because 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...
```

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" later in this file.) 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.

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" later in this file.) 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" later in this file.) 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 (described in "Command-Line Options" earlier in this file), the **.SILENT** directive, the **!CMDSWITCHES** directive, or the @ modifier (all described later in this file).

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 file 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 you specify other commands 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 (described earlier in this file) suppresses display for the entire makefile; the **.SILENT** directive (described later in this file) 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. At least one space or tab must appear between the number and the command. (For more information on using this number, see "Exit Codes from Commands" later in this file.) The /I option (described earlier in this file) turns off error checking for the entire makefile; the **.IGNORE** directive (described later in this file) 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" later in this file.) 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" later in this file.

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" later in this file.

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 (see "Command Syntax" earlier in this file), 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 command. The syntax to create the inline file is:

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, which commands to execute, and names 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" later in this file.)

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, recursion, 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 (limited only by virtual memory). 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" later in this file. 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" earlier in this file.

- 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" later in this file.

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

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" later in this file.) 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**. (See "Preprocessing Directives" later in this file.) A macro name can be "undefined" in a makefile by using the **!UNDEF** preprocessing directive.

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, use **!UNDEF**, 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 =
```