```
$(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" later in this file.) 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 file-name 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" earlier in this file.

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 that you specified to the operating system when you ran NMAKE; this name is `NMAKE` unless you have renamed the utility file. Use this macro to call NMAKE recursively. 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. To pass these options when you call NMAKE recursively, specify `/$(MAKEFLAGS)` with the NMAKE call. You cannot redefine **MAKEFLAGS**. To change the /D, /I, /N, and /S options within a makefile, use the preprocessing directive **!CMDSWITCHES**. (See "Preprocessing Directives" later in this file.) 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 and start a new NMAKE session. Either invoke the MAKE macro as `$(MAKE)` or specify `NMAKE` literally. The called NMAKE session is independent from the original session.

Environment-variable macros and information in TOOLS.INI are available to the recursive session. For information on making macros available to the called NMAKE, see "Macros in Recursion" later in this file.

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
    $(MAKE) /$(MAKEFLAGS)
    cd ..

vers2 :
    cd \vers2
    $(MAKE) /$(MAKEFLAGS) /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" later in this

file.) You can redefine these macros to represent part or all of a command line, including options. The command macros are listed and described below.

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" later in this file.) 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" earlier in this file.) 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;
```

Macros in Recursion

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:

- 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 **CC** and **AS**

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" later in this file. The **.SUFFIXES** list determines priorities for applying inference rules; see "Dot Directives" later in this file.

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" later in this file.

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" earlier in this file.)

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" later in this file.) 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" later in this file.

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. The following table describes the predefined inference rules.

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
.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. The table above 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.FOR exists, NMAKE applies the `.for.exe` inference rule. If both SAMPLE.C and SAMPLE.FOR exist, and if `.c` appears before `.for` in the **.SUFFIXES** list, NMAKE instead 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" earlier in this file, 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" earlier in this file.) 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 "Command-Line Options" earlier in this file.) 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" earlier in this file.)

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 the following table. To compare two strings, use the equality (==) operator and the inequality (!=) operator. Enclose strings in double quotation marks.

Binary Operators for Preprocessing

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
&&	Logical AND
||	Logical OR
<<	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 .c.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 and 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, the command line

```
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" earlier in this file.)

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 because 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 because 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 errorout of memory
255	Target is not up-to-date (issued only when /Q is used)