Alcatel·Lucent

Alcatel-Lucent *nmake* Product Builder

nmake 10 User's Guide

Issue 1.8
June 2008

Notice

Trademarks

Contacts and Additional Information

For more information regarding Alcatel-Lucent nmake Product Builder, including current release information, downloads and technical support, visit our web site at http://www.bell-labs.com/project/nmake/. For technical support send email to nmake@alcatel-lucent.com. For license updates and ordering information send email to software@alcatel-lucent.com.

tained by the U.S. Departments of State, Treasury, or Commerce. In addition, any SOFTWARE subject to this agreement may not be exported, reexported, or transferred to any end-user engaged in activities, or for any end-use, directly or indirectly related to the design, development, production, use, or stockpiling of weapons of mass destruction, e.g. nuclear, chemical, or biological weapons, and the missile technology to deliver them.

3. **LIMITED WARRANTY**

   A. ALCATEL-LUCENT WARRANTS ONLY THAT THE SOFTWARE WILL BE IN GOOD WORKING ORDER AND, FOR A PERIOD OF THIRTY (30) DAYS FROM THE INITIAL PURCHASE OF THE ASSOCIATED LICENSE KEY(S), WILL REPLACE, WITHOUT CHARGE, ANY SOFTWARE WHICH IS NOT IN GOOD WORKING ORDER.

   B. ALCATEL-LUCENT DOES NOT WARRANT THAT THE FUNCTIONS OF THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT SOFTWARE OPERATION WILL BE ERROR-FREE OR UNINTERRUPTED.

   C. ALCATEL-LUCENT HAS USED REASONABLE EFFORTS TO MINIMIZE DEFECTS OR ERRORS IN THE SOFTWARE. HOWEVER, YOU ASSUME THE RISK OF ANY AND ALL LIABILITY, DAMAGE OR LOSS FROM USE, OR INABILITY TO USE THE SOFTWARE.

   D. YOU UNDERSTAND THAT, EXCEPT FOR THE 30 DAY LIMITED WARRANTY RECITED ABOVE, ALCATEL-LUCENT, ITS AFFILIATES, CONTRACTORS, SUPPLIERS AND AGENTS MAKE NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIM ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR WARRANTY AGAINST INFRINGEMENT.

4. **EXCLUSIVE REMEDIES AND LIMITATIONS OF LIABILITIES**

   A. YOU AGREE THAT YOUR SOLE AND EXCLUSIVE REMEDY AGAINST ALCATEL-LUCENT, ITS AFFILIATES, CONTRACTORS, SUPPLIERS, AND AGENTS FOR LIABILITY, LOSS OR DAMAGE CAUSED IN ANY WAY BY THE SOFTWARE REGARDLESS OF THE FORM OF ACTION, WHETHER IN CONTRACT, TORT, INCLUDING NEGLIGENCE, STRICT LIABILITY OR OTHERWISE, SHALL BE THE REPLACEMENT OF ALCATEL-LUCENT INC. FURNISHED SOFTWARE WITHIN THE LIMITED WARRANTY PERIOD. THIS SHALL BE EXCLUSIVE OF ALL OTHER REMEDIES AGAINST ALCATEL-LUCENT, ITS AFFILIATES, CONTRACTORS, SUPPLIERS OR AGENTS, EXCEPT FOR YOUR RIGHT TO CLAIM DAMAGES FOR BODILY INJURY TO ANY PERSON.

   B. REGARDLESS OF ANY OTHER PROVISIONS OF THIS AGREEMENT, NEITHER ALCATEL-LUCENT NOR ITS AFFILIATES, CONTRACTORS, SUPPLIERS OR AGENTS SHALL BE LIABLE FOR ANY INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS) SUSTAINED OR INCURRED IN CONNECTION WITH THE USE, OPERATION, OR INABILITY TO USE THE SOFTWARE.

   C. YOU AGREE THAT IN THE EVENT ANY CLAIM, SUIT OR PROCEEDING IS BROUGHT AGAINST ALCATEL-LUCENT IN CONNECTION WITH THE SOFTWARE OR THIS AGREEMENT, SUCH CLAIM, SUIT OR PROCEEDING WILL BE BROUGHT OR FILED IN THE COURTS OF THE STATE OF NEW JERSEY, UNITED STATES OF AMERICA, AND THAT ANY SUCH CLAIM, SUIT OR PROCEEDING WILL BE GOVERNED BY THE LAWS OF THE STATE OF NEW YORK, UNITED STATES OF AMERICA, WITHOUT REGARD TO ITS CHOICE OF LAW RULES.

Alcatel-Lucent nmake Product Builder
Right-to-Use Software License Agreement Addendum

About this Document

Conventions Used

The following conventions are used throughout this document:

- References to nmake in the shown serif font refer to the nmake command or tool as part of the Alcatel-Lucent nmake Product Builder package.
- Command Syntax
    - Words or symbols in this type are to be entered literally, exactly as shown.
    - Words in *italics* stand for variables for which you should make the appropriate substitution.
    - Square brackets ([ ]) indicate that the enclosed word (which can be a variable or actual word to enter) is optional. If you use an option, do not enter the brackets.
    - A shell prompt ($) is shown before each specific example of a command line.
    - Output generated in response to a command example is shown immediately following the command and is shown in this type.
- File names and directory names are shown in this type. This type is also used when referencing executable programs, such as cc.
- When a file is referenced as a target or prerequisite in an nmake assertion, it is shown in this type.
- Computer output and file listings are shown in this type.
- A list of prerequisites may extend over multiple lines by terminating each line with a backslash (\). One or more whitespace characters should either precede the backslash or start the next line.
- Input and output lines that wrap to the following line due to the margin constraints of this manual contain a backslash (\) at the end of each line.

System commands are referenced as *name*(n), where *name* is the name of the system command and n identifies the section of the system manual in which the command's man page is found.

Contents

Contents

Contents

Contents

Figures

Figures

Contents

Overview of Alcatel-Lucent nmake Product Builder

1

This chapter provdes a high-level overview of the architecture and processing of the Alcatel-Lucent nmake build tool.

Alcatel-Lucent nmake Architecture and Processing

Figure 1-1 shows the architecture of Alcatel-Lucent nmake and the processes that are executed when nmake is run. The numbers in brackets in the following text refer to the numbers in Figure 1-1.

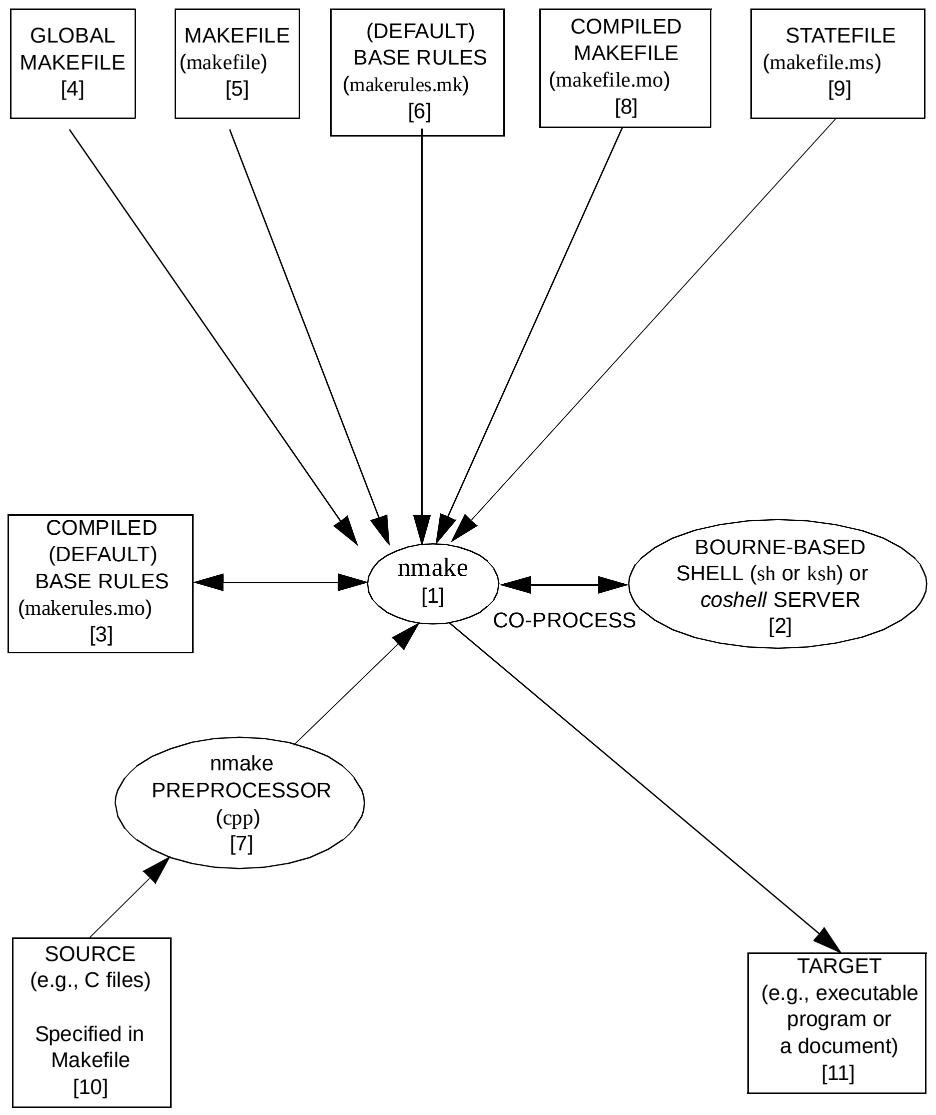

GLOBAL MAKEFILE [4]
MAKEFILE (makefile) [5]
(DEFAULT) BASE RULES (makerules.mk) [6]
COMPILED MAKEFILE (makefile.mo) [8]
STATEFILE (makefile.ms) [9]
COMPILED (DEFAULT) BASE RULES (makerules.mo) [3]
nmake [1]
CO-PROCESS
BOURNE-BASED SHELL (sh or ksh) or *coshell* SERVER [2]
nmake PREPROCESSOR (cpp) [7]
SOURCE (e.g., C files) Specified in Makefile [10]
TARGET (e.g., executable program or a document) [11]

Figure 1-1. Alcatel-Lucent nmake Architecture and Processing

In simple terms, a user writes a makefile [5] (an ASCII file that tells nmake what to build) and then executes nmake [1], which builds a product [11] or products based on the makefile instructions and in accordance with various rules specified in the environment. A more detailed description of the processing follows.

When nmake [1] is executed, it runs under either a Bourne-based shell (if is building a product on a single machine) or a *coshell* server (if it is building a product on several machines attached to a local area network) [2]. The processing proceeds as follows:

1. nmake reads the user's makefile [5] to determine which base rules file [6] is to be used.
2. nmake reads the compiled base rules file [3] to become initialized.
3. nmake uses the instructions specified by the user on the nmake command line or in the global makefile [4] to include global rules files.
4. nmake processes the user's makefile [5] to bring a target or targets up-to-date. In the process, source files [10] are scanned for implicit file prerequisites as requested by assertions (dependency statements) in the makefile [5].

   NOTE:
   The base rules [6] and the makefiles[5] are preprocessed by the nmake preprocessor, cpp [7], whenever necessary.

5. When nmake [1] finishes processing, a target [11] or targets have been generated.
6. After nmake processing is complete, the compiled makefile[8] (*makefile*.mo) and the statefile [9] (*makefile*.ms) are created in the current directory.

   NOTE:
   If the statefile is removed before the next invocation of nmake, all targets will be remade because nmake uses the statefile to determine what is up-to-date.

The following section describe in more detail the processes and files that appear in Figure 1-1.

The Shell Interface

The shell interface is controlled by the COSHELL variable, which can be set to sh or ksh (if nmake is to build a product on a single machine), or to coshell (if nmake is to build a product on multiple machines connected to a local area network).

Single-Machine Mode

When a product is to be built on a single machine, the user can specify COSHELL=sh to use the Bourne shell or COSHELL=ksh to use the Korn shell, but the Korn shell is the more efficient. (See ksh(1) for more information.)

After it has read the makefile, nmake sends a series of shell scripts to the shell. Communication between nmake and the shell takes place over pipes, which are a standard UNIX IPC mechanism. nmake sends commands to the shell on one pipe and receives status information from the shell on another pipe, as shown in Figure 1-2.

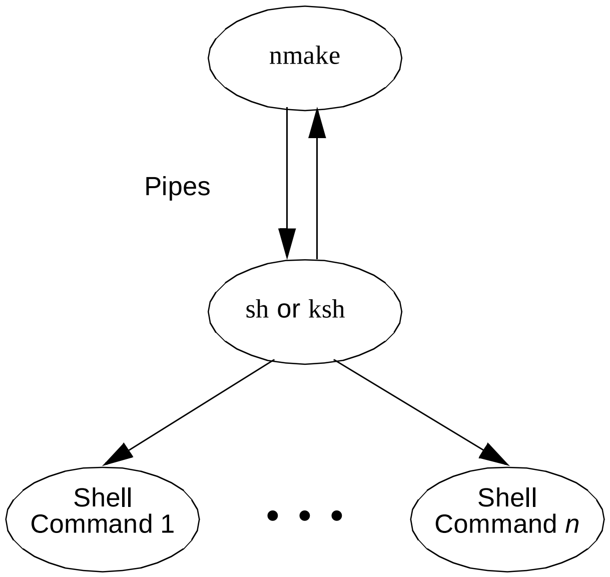


Figure 1-2. Single-Machine Mode of Operation

All commands that update targets in action blocks are sent to a single copy of the shell, so nmake forks just once to initiate the shell as a coprocess. While the commands that update one target are being executed by the shell, nmake can check the prerequisites of the other targets. Therefore, the next target to be made is almost always determined by the time the current target has been made.

Due to the arrangement of the coprocess, nmake makefiles can be viewed as a collection of labelled shell scripts, where labels are targets. Each labelled shell script is sent to the shell as a block, a process that is faster than the “line-at-a-time” technique used in make(1). In addition, nmake allows the shell flow-control constructs (such as if, case, while, and for) to cross the newline boundaries without the intervening backslash and semicolon characters, so natural shell syntax can be used in each block.

In the following example, the same makefile is processed with nmake and again with make(1) to illustrate the shell flow control constructs provided with nmake.

Contents of Makefile

```
test:
        for i in a b
        do
          echo $i
        done
```

Run nmake

```
$ nmake
```

Output

```
+ echo a
a
+ echo b
b
```

Run make(1) on the same Makefile

```
$ make
```

Output

```
for i in a b
    Make: Cannot load for.  Stop.
    *** Error code 1

    Stop.
```

Multi-Machine Mode

When a product is to be built on multiple machines connected to a local area network, the COSHELL variable must be set to coshell. (See Chapter 10, *Building with Alcatel-Lucent nmake*, for more information about using coshell with nmake.)

NOTE:
When COSHELL is set to coshell, nmake uses Korn shell version 88i or a later version.

The interface between nmake and the shell in multi-machine mode is similar to the interface between them in single-machine mode except that the coshell server replaces the Bourne-based shell. The coshell server is the same as the coshell executable.

After it has read the makefile, nmake sends a series of shell scripts to the coshell server, which then distributes work that needs to be done to the idle machines on the network. Status information from the coshell server is returned to nmake.

Communication between nmake and the coshell server takes place over pipes. The coshell server controls a background ksh(1) shell process, initiated by rsh(1), on each of the machines connected to the network. Network pipes are used for communication between the coshell server and each background ksh shell process (see Figure 1-3).

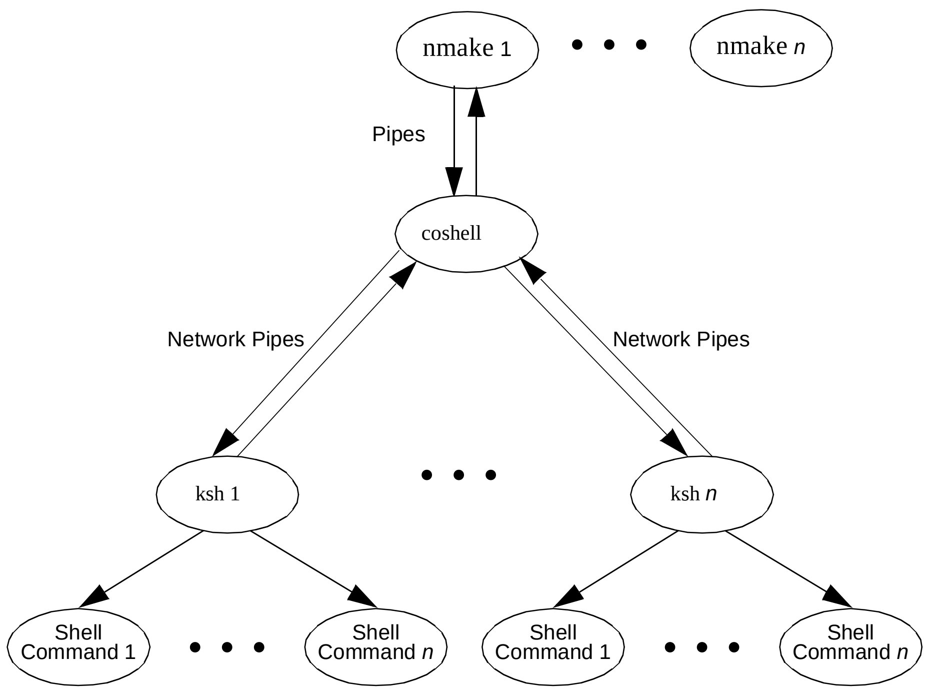


Figure 1-3. Multi-Machine Mode of Operation

The nmake Preprocessor, *cpp*

Alcatel-Lucent nmake has its own preprocessor, cpp, which supports viewpathing. For more information about cpp, see the *Alcatel-Lucent nmake Product Builder Reference Manual*. For information about viewpathing, see Chapter 10, *Building with Alcatel-Lucent nmake*.

The Default Base Rules File

The default base rules, which are defined in a makefile called Makerules.mk, define a collection of options, operators, common target atoms, metarules, and variables that you can use when writing a makefile. The makefile is in the *nmake_install_root*/lib/make directory of the Alcatel-Lucent nmake software distribution, where *nmake_install_root* is the base directory where Alcatel-Lucent nmake is installed. nmake reads Makerules.mk before it reads your makefile, and uses it to interpret many common makefile constructs.

The Global Makefile

The global rules file is an optional makefile that can be used to define rules that are specific to a particular product. It is a suggested organizational mechanism that promotes makefile reuse. The rules defined in this file take precedence over the rules in the default base rules file.

The Compiled Makefile

The compiled makefile (also called the object file) is used to store the compiled version of a makefile. It has the same basename as the makefile and the suffix .mo. The compiled makefile decreases the amount of time required to perform a build when nmake next processes this makefile, by making it unnecessary for nmake to reread and reparse the makefile.

The contents of *makefile*.mo can be listed by entering the following command:

```
$ nmake -blrvf makefile.mo
```

where *makefile* is the name of the makefile that was used to build the target.

The Statefile

The statefile has the same basename as the makefile and the suffix .ms. The statefile contains the current nmake execution status including such things as target dependency information, variable definitions, time-stamps, etc. The statefile is used to minimize the work that must be done on later calls to nmake..

The contents of *makefile*.ms can be listed by entering the following command:

```
$ nmake -blrvf makefile.ms
```

where *makefile* is the name of the makefile that was used to build the target.

NOTE:
If the statefile was created with a version of Alcatel-Lucent nmake older than 3.x, Alcatel-Lucent nmake 3.x automatically converts the statefile to the 3.x format.

The Lock File

The lock file has the same basename as the makefile and the suffix .ml. The lock file is created when nmake starts processing a makefile and is removed when complete. It prevents multiple invocations of nmake from simultaneously processing the same makefile. When possible, stale lock files from terminated processes are detected and removed. The lock file contains the pid and logid of the process that created it and the host machine on which the process was run. The format of the lock file is as follows:

pid<tab>logid@host

The User Makefile

As we have already noted, before running nmake, you must first write specifications in a text file called a makefile. The components of a makefile constitute a rich language, designed to support the construction and maintenance of complex software and documentation systems, or indeed, of any system that consists of related files that are to be combined in a consistent way. Conceptually, a makefile describes the components of the product and how these components are assembled. Subsequent chapters will describe this concept in more detail.

How a Makefile Defines a Product Structure

Generally, a makefile defines a product structure as an inverted tree. This inverted tree may have many dependency relationships. A dependency relationship in nmake is expressed as an assertion. Usually, an assertion is comprised of a single target that is to be built, a list of prerequisites for building the target, and specific steps or actions that must be performed to build the targets.

For example, suppose the product (or root target) is a book. Then the tree (or dependency relationship) might describe the book as consisting of a set of chapters. Each chapter, in turn, is defined as consisting of a collection of sections. The sections may then be further subdivided, and so on. Sections of the book may also refer to tables or illustrations that are stored separately.

The idea behind nmake is that when there is a change to an illustration, every section containing the illustration must be rebuilt; every chapter containing those sections must be rebuilt; then, finally, the book is rebuilt. The specific steps (or actions) are specified in the makefile as part of the assertions. In this case, the actions specified in the assertions would contain specific instructions for nmake to process the text using a text formatter.

In nmake's vocabulary, illustrations are prerequisites for sections; sections are prerequisites for chapters; and the chapters are prerequisites for the book.

All other aspects of building makefiles are designed to deal with more complex variations of this simple model, for example,the location of product components in multiple directories, the imposition of product or organizational standards, the maintenance of different versions, and so on.

How nmake Makes a Target: An Example

At the beginning of this chapter, we presented a high-level view of nmake processing in terms of the files and processes involved. The following section describes this processing in more detail by means of a concrete example.

Suppose a makefile, named Makefile, contains the following:

```
convert :: main.c init.c
```

(This one line constitutes a complete and valid makefile.)

This statement, which is called an assertion, specifies that the program convert is built from main.c and init.c. The file on the left-hand side of the source dependency assertion operator (::) is the *target* . The files on the right-hand side of the source dependency assertion operator are the *prerequisites*.

Because this assertion contains no other information about how to compile the source code to build convert, nmake uses the predefined rules in the default base rules to build convert.

As nmake executes, each shell script that is used to build the target is executed and echoed on the standard error (stderr) by the shell, allowing the user to follow the build procedure as it occurs. For example, if we use nmake to build this makefile, by entering the command:

```
$ nmake convert
```

we will get the following output from the shell:

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
      pp/B49DF4E0.bincc -I- -c main.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
      pp/B49DF4E0.bincc -I- -c init.c
+ cc -O -o convert main.o init.o
```

The results of the execution are:

- two intermediate object files, main.o and init.o
- the product named convert
- the compiled makefile, Makefile.mo, and the statefile, Makefile.ms.

nmake stores infomation about what it has done and the state of the various source files, intermediate files, and product files in the Makefile.mo and Makefile.ms files. In this example, nmake saves the fact that if the init.c file changes or any files that init.c includes changes, init.c needs to be recompiled, building a new init.o. The same information about the relationship between main.c, its header files, and main.o is saved. The information about the relationship between init.o, main.o, the linked libraries, and convert is also saved. These relationships are called *dependency* relationships. The collection of all assertions for a given target is called a *dependency rule*.

In general, the action block of each assertion in a makefile contains a set of instructions that specify how to combine the prerequisite files to produce a target; nmake sends these instructions to the shell. But in our example, we did not provide any instructions for compiling the source files to produce a product; that is, we did not specify that .c files may have header files, and we did not specify where nmake should find the source files. All this information is part of the source dependency assertion operation, which is defined in the default base rules.

As we have seen, the default base rules are defined in a makefile called Makerules.mk. that contains built-in knowledge you can use when you are writing your makefile. nmake reads Makerules.mk before it reads your makefile.

The makefile we have been discussing makes use of the following information from the default base rules:

- the definition of the source dependency assertion operator (::)
- how to compile source files
- how to scan files with a .c suffix for included header files.

Officially, language scanning patterns are defined in a file called Scanrules.mk, and are included by Makerules.mk. However, they are typically considered to be part of the default base rules, since there are dependencies between them.

You can also define your own global makefile that further expands or modifies the set of default rules. nmake is capable of reading many global makefiles before reading your specific makefile.

When a makefile is processed, nmake uses the value of the CC variable to determine which compiler to use (the default is cc). nmake uses the probe configuration files (generated whenever a compiler is first used by nmake) to generate the commands for the shell. The probe tool is described in *Appendix A*.

In the previous example, nmake used the -Qpath /nmake3.x/lib flag to force cc to pick up the nmake version of cpp (these flags may be different for different compilers). nmake’s cpp contains special flags that allow nmake to have greater control over the location of #include files.

How nmake Makes a Target: An Algorithm

This section provides still greater detail about nmake processing by describing the steps nmake takes in the process of making a target. They are:

1. Determine the type and the properties of the target. This process is called binding. (See Chapter 5, *Using Atoms and Special Atoms,* for more information on binding.)

2. See whether the target has ever been made.

   If so, check whether the target is active (i.e., the action is executing to build the target), then block until the action completes before returning. nmake may execute a number of target actions concurrently. This technique is referred to as *parallel processing*. The jobs command-line option can be used to specify the number of actions that can be performed concurrently. For more information concerning the jobs command-line option, see the *Alcatel-Lucent nmake Product Builder Reference Manual.*

3. Check statefile consistency.

   The current time, prerequisites, and action are compared with those recorded in the statefile. Any differences force the target to be rebuilt.

4. Complete the set of prerequisites.

   There are Special Atoms that may be used to add other prerequisites to the list of prerequisites. These Special Atoms are .INSERT, .INSERT.*pattern*, .APPEND and .APPEND.*pattern*. (See Chapter 5, *Using Atoms and Special Atoms,* for more information on these Special Atoms.)

5. Make the explicit prerequisites.

   The explicit prerequisites are made (recursively) from left to right. The most recent prerequisite time is saved in order to check whether the target is out-of-date.

6. Make the implicit prerequisites.

   nmake has rules for scanning numerous programming languages. (See Chapter 8, *Defining Custom Scan Rules*, for details.) When nmake scans a file, any implicit prerequisites are noted, and made if necessary, recursively from left to right.

   For example, suppose a file called x.c includes a file called x.h, which includes a file called y.h. When nmake scans x.c, it recognizes that x.h and y.h are implicit prerequisites of x.c; therefore, nmake makes them if they are generated files and out-of-date. The most recent time-stamp of any implicit prerequisite (in this case, x.h and y.h) is saved by nmake. These time-stamps are then used to check whether the scanned file (in this case, x.c) is out-of-date.

7. Compare the time-stamp on the newest prerequisite with the current target date.

If the prerequisite is newer, the target action is triggered to build the target.

8. Synchronize the statefile information.

   When the target has been built (i.e., the action, if any, has completed), its time, action, and prerequisites are saved in the statefile in preparation for the next invocation of nmake. If necessary, a makefile can force statefile synchronization by using the .SYNC Special Atom. (For more information on the .SYNC Special Atom, see the *Alcatel-Lucent nmake Product Builder Reference Manual*.)

Contents

Understanding Makefiles

2

The Elements of a Makefile

The elements that can appear in a nmake makefile are described in the following sections.

Assertions

Assertions define dependency relationships between targets and prerequisites; they are the core components of the makefile. The target of an assertion is the thing that nmake will build (i.e., a program or library), while the prerequisites are the things (i.e., source files) that nmake will use to build the target. Everything else in a makefile is designed to expand the applicability, generality, and portability of assertions.

Assertions may contain *targets*, *prerequisites*, *assertion operators*, and *actions*. The following is an example of an assertion containing a target (program), a built-in assertion operator (:), and two prerequisites (init.c and prog.c), but no actions:

```
program : init.c prog.c
```

This assertion tells nmake to build the program program using the source files init.c and prog.c. We will see later that nmake could actually build program without any further information about how to do so by using information available to it in the environment.

nmake provides a variety of assertion operators that may be used to establish dependency relationships between targets and prerequisites. It is also possible for users to define their own assertion operators.

The actions in an assertion may be implicit (e.g., defined in the default base rules) or explicit (e.g., defined in the makefile). They can be written in either the nmake programming language or in shell programming constructs.

See Chapter 6, *Using Assertion Operators*, for a description of how to use assertion operators and the *Alcatel-Lucent nmake Product Builder Reference Manual* for manual pages for the assertion operators provided by nmake.

Variables

Both string and integer variables may be defined in a makefile. The let keyword is used to define an integer variable. Variables that are not defined using the let keyword are strings. As in the shell, there is a difference between defining a variable and referencing it. For example, a string variable can be defined in a statement like X=string and referenced by writing $(X). When a variable is referenced, it is *expanded*; that is, it is replaced by its contents.

Other types of variables that may appear in a makefile are:

- Built-in Engine Variables

  These variables have names and values that are known in advance and can be used without additional definition.

- Automatic Variables

  These are notational variables that stand for either different components of an assertion, or specific options of the current nmake execution. The values of automatic variables can be edited using the edit operators.

See Chapter 3, *Using Variables*, for a description of how to use variables and the *Alcatel-Lucent nmake Product Builder Reference Manual* for manual pages for them.

Edit Operators

The edit operators provided by nmake form a very powerful editing language that can be used to edit the contents of a variable. This language offers an extraordinary degree of control over how a variable string is to be interpreted and manipulated.

See Chapter 4, *Using Edit Operators*, for a description of how to use edit operators and the *Alcatel-Lucent nmake Product Builder Reference Manual* for manual pages for them.

Special Atoms

Special Atoms provide fine control over assertions, actions, and nmake processing. There are nearly 100 of them.

See Chapter 5, *Using Atoms and Special Atoms*, for a description of how to use atoms and Special Atoms and the *Alcatel-Lucent nmake Product Builder Reference Manual* for manual pages for the Special Atoms.

Command-Line Options

The nmake command-line options provide control over nmake execution. They are also part of the makefile language. The set *option* statement allows you to include command-line options in the makefile. However, any of the options that are set in the makefile can be overridden using options on the command line.

See the *Alcatel-Lucent nmake Product Builder Reference Manual* for manual pages for the command-line options.

The Makefile Programming Language

A makefile is written in its own programming language, which allows for the inclusion of other makefiles, the definition and setting of numeric as well as string variables, and the ability to test for conditions and establish iterative loops. It also allows shell statements and *assertions* that define the relationship between targets and prerequisites. Variable definitions and assertions may be subject to conditional testing.

See Chapter 7, *Programming in Alcatel-Lucent nmake*, for a description of the programming statements that are available with nmake.

Using # Directives

Makefiles should not contain # directives. State variables and programming constructs should be used instead.

NOTE:
nmake programming statements provide every feature that is available with cpp. Using cpp directives is possible, but is not encouraged. Use a state variable instead of a #define directive and use the include programming statment instead of a #include directive.

Commenting Conventions

The commenting conventions used for makefiles are similar to those used for writing C language programs.

The basic convention is that all characters between /* and */ are considered commentary (that is, the whole comment is stripped out by cpp's scanner) and are ignored by nmake. As in C language programs, comments cannot be nested.

The # character is a valid character in a makefile and it is possible to use it to start a comment. When this is done, the comment extends until the end of the line.

CAUTION:
We do not recommend using the # character to set off a comment.

WARNING:
Do not use # comments in the first column of the first kilobyte (1024 characters) of the makefile. If nmake *finds a # character in the first column of the first kilobyte and it is not part of a directive,* nmake *will treat the makefile as a* make *makefile and use* /bin/make *to process it. See the example below.*

Example:

```
#this will cause make to process the build
convert.o : init.o main.o
	/* specify final target */
	ld  -r -o convert.o init.o main.o
```

Formatting a Makefile

The first target name in any assertion must start in column one; the targets and the assertion operator must be on the same line. The list of prerequisites can be extended over multiple lines by terminating a line with a backslash (\). One or more whitespace characters must either precede the backslash or start the next line.

Succeeding lines that start with a tab are considered part of the action block. If lines start with one or more spaces, they are also considered part of the action block, but that usage is discouraged and a warning message is given, as the resulting makefile loses some portability.

Names in the target and prerequisite lists must be separated by either spaces or tabs.

Naming a Makefile

By convention, makefiles are named either Makefile, makefile, or *makefile_name*.mk where *makefile_name* can be any name that you choose; however, this naming convention does not have to be followed. You can name your makefiles using any naming convention that you desire.

The makefile has two associated files that are created when nmake is executed: an *object file* (compiled makefile) and a *statefile*.

CAUTION:
The object file and statefile are created in the directory where nmake *is executed; you should not move them, because they are used for subsequent invocations of* nmake*. Both files are stored using a binary format, and should not be modified.*

Using a Makefile

The command line to execute nmake is:

```
$ nmake [arguments]
```

The *Alcatel-Lucent nmake Product Builder Reference Manual* contains manual pages for all the command-line options.

```
$ nmake [-f makefile_name] [target]
```

The -f command-line option above allows you to specify the *makefile_name* that is to be used as the makefile. If you do not specify a makefile name on the command line by using the -f option, nmake searches in the directory where nmake was invoked for the following names:

- Makefile or
- makefile.

The search occurs in the given order. Any makefile name other than Makefile or makefile must be explicitly named (using the -f command-line option); otherwise nmake displays an error message and processing halts.

Any target in a makefile can be referred to as an *entry point*. The user can specify the target that is to be built on the command line . If no target names are specified on the command line, nmake will build the first target in the input makefile that is not marked as .FUNCTIONAL, .OPERATOR, or .REPEAT, and is neither a state variable nor metarule.

Suppose that in the example below, the makefile name is files.mk. The variable FILES contains the list of files that are to be used in the build. The target that is to be built is called filelist. The nmake command line and the output from the shell are shown in the following example.

Contents of files.mk

```
FILES = a.c b.c tot.c
filelist :
    /usr/5bin/ls $(FILES)
```

Run nmake

```
$ nmake -f files.mk filelist
```

Output

```
+ /usr/5bin/ls a.c b.c tot.c
a.c
b.c
tot.c
```

In the output listing for this example, the line that starts with a + is actually the shell echoing the shell scripts it received. The lines that do not start with a + are the output from the shell scripts that have been processed.

The Makefile Environment

The following sections describe the environment in which the user’s makefile exists. This environment provides a wealth of predefined information that both simplifies the individual makefile and greatly extends its power. Understanding this environment is important if the user is to make the most of nmake.

The Default Base Rules File

nmake_install_root/lib/make/Makerules.mk, the default base rules file provided with Alcatel-Lucent nmake, is a makefile that defines a collection of options, operators, common actions, metarules, and variables suitable for use in the UNIX system programming environment. It includes predefined rules for a variety of transformations. Thus it includes rules for compiling programs in different languages, rules for creating backup files, rules for cleaning up intermediate files, and so on.

Your organization may use these rules, add to them, or replace them. However, you should not modify the default base rules file or the files that it includes. If you want to modify the default base rules, you should place the modification in a global rules file (see the section *Global Rules*, in this chapter). If you want to replace the default base rules, you should create your own base rules file, compile it, and then use the rules nmake programming statement, which specifies the base rules file that will be used in the makefile, in the first line of the makefile. The format is:

rules ["*base-name-of-base-rules-file*"]

If the rules statement is used in the makefile, the default base rules are not used; instead, the base rules specified on the rules line are used. If the rules statement is omitted from the makefile, the default base rules file provided with Alcatel-Lucent nmake is used.

The base rules file has to be compiled before it is used by nmake. The output file name is created by changing the base rules file suffix to .mo. For example, if the name of the base rules file is User-Makerules, the compiled file name would be User-Makerules.mo.

NOTE:
nmake searches for the compiled file using the directories of .SOURCE.mk. (See the section *Scan Rules*, in this chapter, for a discussion of search directories.)

The following command line is used to compile the base rules file. (See the -b and -c command-line options in the *Alcatel-Lucent nmake Product Builder Reference Manual* for more details.)

$ nmake -bc -f User_Makerules.mk

The following statement (which must be the first line in the makefile) could then be used to include the rules contained in the compiled version:

rules "User_Makerules"

The guidelines for using base rules are:

- The rules statement must be the first line of your makefile.
- If you do not use the rules statement on the first line, rules "makerules" (the default base rules provided with nmake) is implied.
- rules with no file name specified indicates that no base rules file is to be used. For example, Makerules.mk uses this statement.
- There can be only one rules statement in a makefile. Specific assertions and assignments within a makefile can override a rule specified in the base rules file.

If only a few modifications to the default base rules are needed, you may put the modified rules in your project-wide (global rules) makefile described in the following section instead of defining your own set of base rules.

In summary, nmake rules can be defined and redefined by the user. The order of precedence of rules that nmake uses in the build process is shown in Figure 2-1. (For information about metarules, see the section *Metarules*, later in this chapter.)

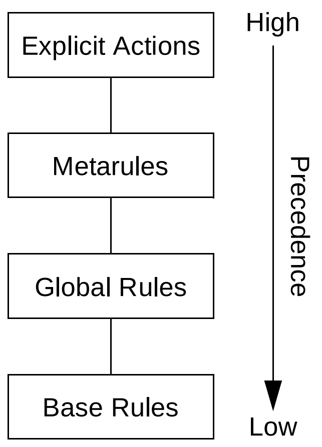


Figure 2-1. nmake Rule Precedence

The following sections deal with varous elements of the nmake environment that are defined in the default base rules file.

Predefined Source Directory Rules

Source directory rules tell nmake what directories to look in to find various types of source files. The following rules provide an example of source directory rules:

```
.SOURCE : .src
.SOURCE.h : /usr/include
.SOURCE.a : /lib /usr/lib
.ATTRIBUTE.%.c : .SCAN.c
```

The .SOURCE rule specifies that source files can be found in the src directory. The .SOURCE.h rule specifies that header files can be found in /usr/include. The .SOURCE.a rule specifies that libraries can be found in /lib and /usr/lib. The .SCAN.c attribute specifies that files with suffixes of .c and .h will be searched for header file prerequisites. See the *Alcatel-Lucent nmake Product Builder Reference Manual* for a discussion of the .ATTRIBUTE.*pattern* Special Atom.

In practice, the *probe* tool determines the path for the location of standard header files (.h) and standard libraries (.a) for each compiler. See the *probe* manual page and Appendix A for more information about the *probe* tool.

Predefined Assertion Operators

As we have noted, assertion operators establish dependency relationships between the prerequisite(s) and target(s) in an assertion. All the predefined assertion operators, with the exception of the dependency assertion operator (:), are defined in the default base rules and have the following syntax:

:*identifier*:

where *identifier* can be any string of characters (A–Z, upper or lower case). The identifier can also be null, which is the case for the source dependency assertion operator (::). (The dependency assertion operator (:) is defined in the nmake engine.)

All the predefined assertion operators provided with nmake are described in the *Alcatel-Lucent nmake Product Builder Reference Manual*. For information about defining your own asserton operators, see Chapter 6, *Using Assertion Operators*.

All assertions that use the :*identifier*: assertion operators are “macro-like” and are expanded to assertions with the dependency assertion operator at compile time. For example, the predefined source dependency assertion operator (::) generates several dependency assertions that can then be understood by the nmake engine; this process is transparent to the user.

You should use these “macro-like” assertion operators in your assertions whenever possible. Occasionally, you may want to define your own operator or use the dependency assertion operator to define a product-specific rule.

Although they are similar in appearance, there is an important difference between the dependency (:) and the source dependency (::) assertion operators:

- The dependency assertion operator is the nmake primitive assertion operator. It is normally used when the action block explicitly describes how to build a target. When it appears in an assertion without an action block, the dependency assertion operator only sets up a dependency relationship between the target list and the prerequisite list.
- The source dependency assertion operator is predefined in the nmake default base rules. It can be used without an action block in assertions that contain source files (e.g., C language files and troff files) as prerequisites

The advantage derived from using the source dependency assertion operator is that the information on how to build a target is already predefined for nmake and therefore does not have to be specified in the makefile. This operator uses its abstraction to hide complicated actions, thereby allowing the user to write concise makefiles.

Metarules

A series of metarules (also called transformation rules) is provided In the default base rules. These metarules describe how to build a file having a name of one pattern (e.g., prog.o) from a file having a name of a different pattern (e.g. prog.c).

A metarule typically identifies a target/prerequisite assertion by the pattern of the target name and the pattern of the prerequisite names. For example, in the default base rules, the rule for making a .o file from a .c file is:

```
%.o : %.c (CC) (CCFLAGS)
      $(CC) $(CCFLAGS) -c $(>)
```

Here the % symbol is a wild-card character (called the *stem*) that matches any character string of a file name. If the prefix and/or suffix are specified, the stem matches the file name without the prefix or suffix. In this example, the suffix is specified (.c); the stem is the basename of the file. The object file generated as the target is dependent on the CC variable (which contains the name of the C compiler) and on the CCFLAGS variable (which contains the compile options), both of which appear as state variables in the assertion. (See Chapter 3, *Using Variables*, for more information about state variables.) This example also demonstrates the use of string variables within the action block; these variables allow the metarule to be generic, that is, to work regardless of the user's C compiler.

As can be seen from this example, metarules help to reduce the number of details that have to be specified in the makefile.

nmake uses the metarules defined in the default base rules as follows:

- If a file having a particular pattern is needed (e.g., x.o), and there is no explicit rule in the makefile for making that file (e.g., x.o : x.c),
- and the file pattern suffix matches the left-hand side of some metarule (e.g., %.o),
- and there exists a file with the succeeding suffix (e.g., %.c) that matches the right-hand side of the metarule (e.g., %.o : %.c),

nmake then infers this rule to make the needed file.

The metarules in the default base rules have been defined for a variety of applications. In most cases, the actions are defined with variables that allow user makefile definitions of CC and CCFLAGS, for example, to be applied. You can also define your own metarules in your makefile.

Metarules are chained as necessary. Given the metarules %.z : %.m and %.m : %.a and the source file x.a, the target file x.z will be generated by first applying %.m : %.a to x.a to build x.m, and then applying %.z : %.m to x.m to build x.z.

Using the default metarules and the source dependency assertion operator supplied with nmake, a one-line makefile can be written for building a target. For example, a target named doio from the prerequisite source files main.c, input.c, process.c, and output.c: could be created in a makefile like:

```
doio :: main.c input.c process.c output.c
```

You can also provide your own implicit rules for use in conjunction with the predefined rules, as in the following examples:.

Contents of Makefile

```
%.tr : %.pr
      grap $(>) | pic | tbl | eqn > $(<)
chap : sec1.tr sec2.tr
      troff -mm -Tpost $(*) | dpost | lpr
```

Run nmake

```
$ nmake
```

Output

```
+ eqn
+ grap sec1.pr
+ pic
+ tbl
+ 1> sec1.tr
+ eqn
+ grap sec2.pr
+ pic
+ tbl
+ 1> sec2.tr
+ troff -mm -Tpost sec1.tr sec2.tr
+ lpr
+ dpost
```

Here the user-defined metarule generates a .tr file from a .pr file by processing the prerequisite file (.pr) through the grap, pic, tbl, and eqn preprocessors. The $(<) variable is an automatic variable that represents the current target. The $(>) variable is another automatic variable that represents the first %-prerequisite, also known as the primary metarule prerequisite. Note that $(>) has a different meaning in non-metarule actions. (Chapter 3, *Using Variables*, contains more information about the use of automatic variables.)

The target chap is dependent on sec1.tr and sec2.tr. All explicit file prerequisites of chap are processed by troff and sent to a printer.

Contents of Makefile

```
INFORMIXDIR = /tools/informix
ESQL=esql
ESQLFLAGS=-e

.SOURCE.h : $(INFORMIXDIR)/incl
.ATTRIBUTE.%.ec : .SCAN.sql

%.c : %.ec .TERMINAL
        $(ESQL) $(ESQLFLAGS) $(>)

t :: a.c
```

Run nmake

```
$ nmake
```

Output

```
+ esql -e a.ec
+ cc -O -Qpath /sabteam/nmake/lib -I-D/ \
    sabteam/nmake/lib/probe/C/pp/ \
    18AA9BAErucb cc -I/tools/informix/incl -I- \
    -I/tools/ informix/incl -c a.c
+ cc -O -o t a.o
```

Here the user-defined metarule (%.c : %.ec) describes how to generate a .c file from a .ec file. (The .TERMINAL Special Atom is required because of the definition of :: in the default base rules.) Therefore, if a.c is needed and a.ec exists, nmake uses the above metarule to infer the following dependency rule:

```
a.c : a.ec
    $(ESQL) $(ESQLFLAGS) $(>)
```

Once a.c is generated, nmake applies the predefined metarule %.o : %.c to compile a.c and the action specified in the source dependency assertion operator (::) definition to generate the target t.

In addition, you can tell nmake to search for .ec files in certain directories by creating a .SOURCE.ec rule in your makefile.

Common Actions

The base rules also define a group of common actions, which are actions that are frequently used by projects. An example is file cleanup. Common actions are specified on the command line, for example:

```
$ nmake clean
```

If clean is not defined in the local makefile, the nmake engine translates the command-line argument into .CLEAN, which is a target defined in the base rules. .CLEAN removes all intermediate generated files.

Any of the nmake common actions can be redefined in the local makefile. For example, if the common action clean performs some action inappropriate for your project, you can define a target called clean (or .CLEAN) in the local makefile. When nmake sees the command-line argument clean, it interprets clean as a target and performs the locally defined actions instead.

There are approximately 30 common actions defined in the default base rules provided with Alcatel-Lucent nmake. You do not need to specify these actions in the makefile unless you want to define your own rule.

The table below describes the common actions that are used in the sample session below.

Common Action	**Description**
clean	All generated intermediate files are removed. Executable and archive files are retained.
clobber	All generated files are removed, except the output of the cpio, pax, and tar common actions.
clobber.install	Removes all of the files installed by the install common action.
cpio	All source files are copied to the cpio(1) archive named *main-target*.cpio, where *main-target* is the first named target in the makefile.
install	All target files of the assertions with the predefined assertion operators ::, :INSTALL:, :INSTALLDIR:, and :INSTALLMAP: are installed in the directory hierarchy rooted at $(INSTALLROOT). INSTALLROOT is a predefined nmake variable in the default base rules.
list.generated	Lists the files that have been generated by nmake.
list.install	Lists the full paths of all files to be installed under $(INSTALLROOT) by the install common action.
list.source	Lists the source files.
cc-|+	Assumes the existence of one or more cc-opt (cc+opt) subdirectories, where opt is any valid CCFLAGS value (ex. g, p, pg, etc.). When nmake is invoked with the cc, cc-opt or cc+opt common action, it builds the executables for each of these compiler options, or the specific one specified by opt and places them in the corresponding directory.

In the following example, the install common action installs the target (convert) and all the manual pages specified in the makefile.

Consider the following makefile:

```
convert :: init.c main.c -lld convert.1
```

When the install common action is specified on the nmake command line, the install common action builds the product convert, places it in the bin directory, and places the manual page convert.1 in the man/man1 directory.

Files that contain suffixes that are not specified in the metarules are treated as miscellaneous files and are used in such common actions as cpio and install.

The following command line could be used to invoke the clean common action:

```
$ nmake clean -f makefile
```

where *makefile* is the same makefile used above. In this case, the clean common action removes the intermediate files that were generated, i.e., init.o and main.o.

Sample Session Using nmake with Common Actions

The following sample session provides an extended example of the use of common actions. It begins by describing how a product is built with nmake and continues by illustrating how common actions are used.

1. Create a directory structure similar to the one shown in Figure 2-2. The $CWD/src directory contains the following files: Makefile, main.c, sub1.c, and sub2.c

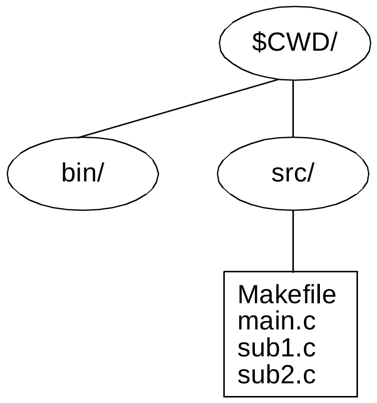


Figure 2-2. Contents of the $CWD Directory Structure

The contents of each subdirectory are explained below.

Contents of $CWD/src/Makefile

```
INSTALLROOT = ../bin
target :: main.c sub1.c sub2.c
```

INSTALLROOT is the location of the root installation directory used in the install common action.

The executable target is defined to be dependent on the files main.c, sub1.c, and sub2.c. The actions of the source dependency assertion operator (::) are defined in the base rules.

2. Change directory to the directory that contains Makefile:

   ```
   $ cd $CWD/src
   ```

3. Execute nmake by issuing the following command line:

   ```
   $ nmake
   ```

   nmake then builds the target target and displays the following output.

   ```
   + cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/\
       C/pp/B49DF4E0.bincc -I- -c main.c
   + cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/\
       C/pp/B49DF4E0.bincc -I- -c sub1.c
   + cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/\
       C/pp/B49DF4E0.bincc -I- -c sub2.c
   + cc -O -o target main.o sub1.o sub2.o
   ```

   The contents of the directory structure resulting from the build are shown in Figure 2-3.

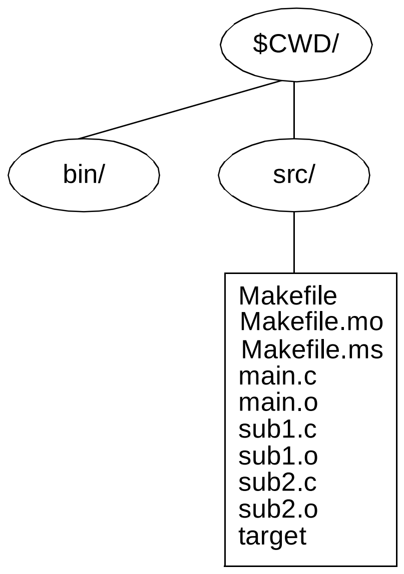


Figure 2-3. Contents of the $CWD Directory Structure after Build

4. Execute nmake with the install common action by issuing the following command line:

```
$ nmake install
```

nmake builds the target target and copies it to the installation directory ($INSTALLROOT/bin). Since the previous nmake execution status is saved in the statefiles, nmake knows that target was previously built (and up-to-date) and therefore only does the installation work.

The output from nmake is shown below.

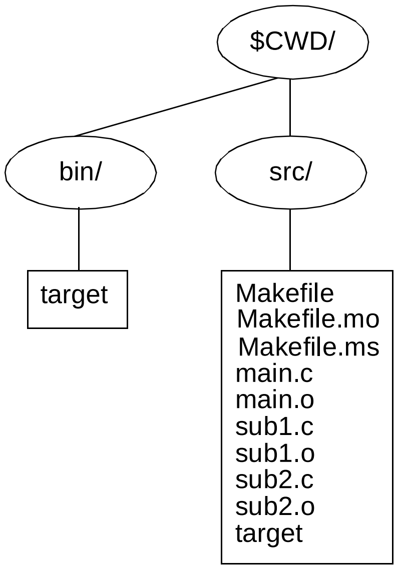
```
+ ignore cp target ../bin/target
```

The result of the installation is shown in Figure 2-4.

Figure 2-4. Contents of the $CWD Directory Structure after Installation

The target file can now be executed from the $CWD/bin directory.

5. Execute nmake with the list.generated common action to list the generated files:



```
$ nmake list.generated
```

The output from nmake is shown below:

```
../bin/target
target
main.o
sub2.o
sub1.o
Makefile.mo
Makefile.ms
```

6. Execute nmake with the list.install common action to list all the files that were installed under $(INSTALLROOT) by the install common action:

```
$ nmake list.install
```

The output from nmake is shown below:

```
bin
bin/target
```

7. Generate a list of the source files that were used in the build by executing nmake with the list.source common action:

```
$ nmake list.source
```

The output from nmake is shown below:

```
main.c
sub1.c
sub2.c
Makefile
```

8. Generate a cpio(1) archive by executing nmake with the cpio common action:

```
$ nmake cpio
```

The output from nmake is shown below:

```
+ cpio -o
+ tr \012
+ echo main.c sub1.c sub2.c Makefile
+ 1> target.cpio
1 blocks
```

The result of the cpio common action is shown in Figure 2-5.

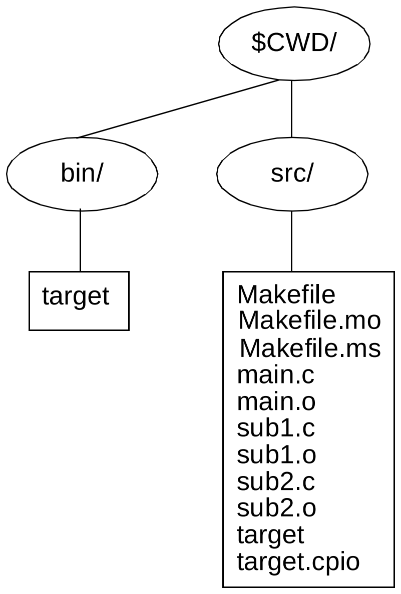


Figure 2-5. Contents of the $CWD Directory Structure after the cpio Common Action

9. Remove the files that were installed by nmake in the installation area by executing nmake with the clobber.install common action:

```
$ nmake clobber.install
```

The output from nmake is shown below:

```
+ ignore rm -f -r ../bin/target
```

The result of the clobber.install common action is shown in Figure 2-6.

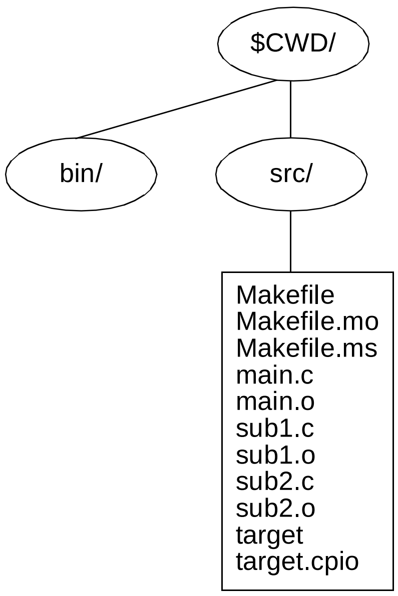


Figure 2-6. Contents of $CWD after the clobber.install Common Action

10. Remove all the files that were generated by nmake in the $CWD/src directory by executing nmake with the clobber common action:

```
$ nmake clobber
```

NOTE:
Note: cpio and installed files are not removed.

The output from nmake is shown below:

```
+ ignore rm -f target main.o sub2.o sub1.o Makefile.mo \
    Makefile.ms
```

The result of the clobber common action is shown in Figure 2-7.

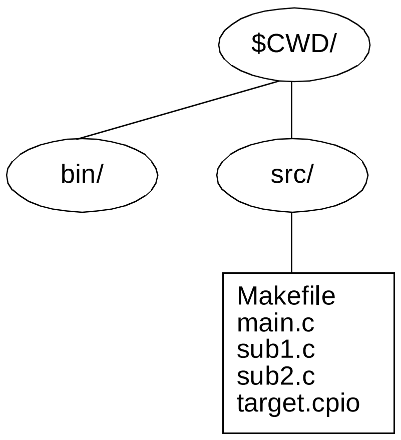


Figure 2-7. Contents of $CWD after the clobber Common Action

11. Regenerate all the files:

```
$ nmake
```

The output is the same as in Step 3 and Figure 2-3.

12. Remove all the intermediate (.o) files:

```
$ nmake clean
```

The output from nmake is shown below.

```
+ ignore rm -f main.o sub2.o sub1.o
```

The result of the clean common action is shown in Figure 2-8.

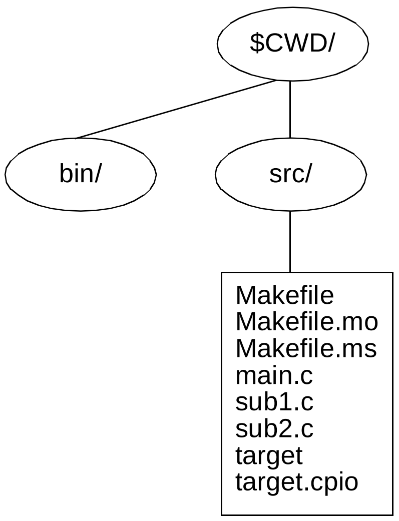


Figure 2-8. Contents of $CWD after the clean Common Action

Scan Rules

nmake has a built-in dependency scanner that searches for dependencies between source and header files. The rules below describe the search procedure the scanner uses to look for include files contained in source files written in C language.

nmake first looks for include files in the current directory; thereafter, the search order is determined by how the include file is defined in the source file. If it is defined using quotation marks (that is, #include "name.h"), the search order is as follows:

- the current directory
- the directories of .SOURCE.c
- the directories of .SOURCE
- the directories of .SOURCE.h.

If the include file is defined by using angle brackets (that is, #include <name.h>), the search order is as follows:

- the current directory
- the directories of .SOURCE.h.

nmake uses the first file it finds that has the name for which it is searching. If the file is not found, nmake issues an error message and exits.

For other types of include files, nmake searches for those files using the directories of .SOURCE.*x* (if present), where *x* is the suffix of the include file, then the directories of .SOURCE.

nmake cpp

nmake's cpp search mechanism for finding included files uses rules similar to those of the standard cpp (provided by your UNIX vendor). The directory of the including file is *not* searched first when the nmake preprocessor searches for a header file; instead, the directories are searched in the order stated by the -I options. nmake searches "" include files in all –I directories listed before -I- and searches <> include files only in the –I directories listed after –I– (this differs from the standard C preprocessor). The standard include directory is always searched last for both include file forms.

NOTE:
The search rules for the standard cpp and nmake cpp are basically the same, except that the standard cpp always searches the directory of the including file first to find an included file, regardless of the user-specified order for searching the directories. Because it is possible to tell nmake where to search for files, it was important to modify the search rules for cpp slightly to allow viewpathing (see Chapter 10, *Building with Alcatel-Lucent nmake*) to operate correctly. Therefore, Alcatel-Lucent nmake uses its own version of cpp. Refer to the cpp manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information.

Consider the following makefile:

```
.SOURCE : quote_hdrs
.SOURCE.h : angle_hdrs
```

and the following C source file segment:

```
#include "quote.h"
#include <angle.h>
#include <stdio.h>
```

nmake generates the following command line to compile this source file:

```
+ cc -O -Qpath /nmake3.x/lib \
    -I-D/nmake3.x/lib/probe/C/pp/B49DF4E0.bincc \
    -Iquote_hdrs -I- -Iangle_hdrs -c mark.c
+ cc -O -o target mark.o
```

The -Qpath option causes cc to pick up nmake cpp. The -Iquote_hdrs option causes cpp to search for quote.h in quote_hdrs; the -Iangle_hdrs option causes cpp to search for angle.h in angle_hdrs; /usr/include is not specified and is always searched last.

Like cpp, nmake displays an error message and exits if it cannot find a header file. Therefore, if a header file is missing, nmake detects the error before the command to compile the source file is given.

CAUTION:
Do not use the following construct in C source files that are compiled with nmake*:*

```
#define X "Y"
      #include X
```

This construct should not be used because include statements requiring macro substitution are not recognized by nmake's scan strategies.

Global Rules

Project-wide rules intended to extend or modify the base rule definitions may be placed in a makefile known as the global rules file. If there are name conflicts in rule definitions between the global rules and the base rules, the rules defined in the global rules makefile override the base rules.

There are two ways to specify a global rules file, either on the nmake command-line or in the makefile, itself.

Specification at the command-line level is as follows:

$ nmake -g *name_of_global_rules_file* **[-f** *name_of_makefile***]**

At the makefile level, you put a statement at the top of the file as follows:

include "*name_of_global_rules_file***"**

You can specify more than one global rules file by using as many include statements as needed.

As an example of the use of a global rules file, suppose that a user has files with a .uc suffix and wants to convert them all to files with a .c suffix, retaining the original files. This may be done as follows. The variable FILES contains the list of files that are to be used in the build. The target that is to be built is called target. On the first invocation of nmake, the target is not built because nmake does not know what to do with files with the .uc suffix. A metarule is then defined in the global rules file called global.mk. The metarule simply copies the files with a .uc suffix to files with a .c suffix. The % symbol is the *stem*. The source dependency assertion operator (::) uses the files specified in FILES to generate the target.

Contents of Makefile

```
FILES = iomodule.uc procmod.uc
target :: $(FILES)
```

Run nmake

```
$ nmake
```

Output

```
make: don't know how to make target
```

Contents of global.mk

```
%.c : %.uc
        $(CP) $(>) $(<)
```

Modify Makefile to include global.mk

```
include "global.mk"
FILES = iomodule.uc procmod.uc
target :: $(FILES)
```

Run nmake again

```
$ nmake
```

Output

```
+ cp iomodule.uc iomodule.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/\
      lib/probe/C/pp/B49DF4E0.bincc -I- \
      -c iomodule.c
+ cp procmod.uc procmod.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/\
      lib/probe/C/pp/B49DF4E0.bincc -I- \
      -c procmod.c
+ cc -O -o target iomodule.o procmod.o
```

Contents

Contents

Using Variables

3

nmake variables play an important part in constructing correct and efficient nmake makefiles. The following types of variables are used in nmake:

- Variables defined by the user,
- Automatic variables,
- State variables,
- Environment variables, and
- Base rules variables.

The *Alcatel-Lucent nmake Product Builder Reference Manual* contains manual pages for all the automatic variables and base rules variables, as well as listings of the environment variables, the base rules variables that can be used in action blocks, and the state variables that are defined in the default base rules.

What is a Variable in nmake?

An nmake variable is an element of a makefile consisting of an alphanumeric name that can hold an integer value, a string, or or other nmake variables. For information about the let programming statement, which is used to designate a variable as an integer, see Chapter 7, *Programming in Alcatel-Lucent nmake*. This chapter discusses only string variables.

String variables can be used:

- In a target list
- In a prerequisite list

- In an action block
- To define other variables.

As an example of the use and value of variables, consider the following makefile fragment:

```
convert : init.c main.c /usr/lib/libld.a
          cc -o convert init.c main.c /usr/lib/libld.a
```

This can be simplified by introducing a variable called FILES, as shown in the following example. The use of the FILES variable increases the readability of the makefile.

```
FILES = init.c main.c /usr/lib/libld.a
convert : $(FILES)
          cc -o convert $(FILES)
```

In general, the use of variables eliminates redundancy and makes makefiles more readable.

This chapter expands on this example, showing you:

- How variables are defined and used
- How variables are expanded
- What automatic variables are, and how they are defined
- How a variable's value can be manipulated during expansion
- How state variables can be used as prerequisites of targets.

Defining a Variable

Assignment statements are used to define nmake variables. The syntax of an assignment statement is:

variable_name   assignment_operator   value

where:

- *variable_name* is the name of a variable, which can be made up of a sequence of letters, digits, underscores, and dots
- *assignment_operator* can be any one of the following: =, :=,==, +=, or &=
- *value* can be a string or list of strings, such as the file names in the example above. Values in the list can be separated by spaces or tabs, but the *assignment_operator* does not have to be set off by separators.

The format $(*variable_name*) may be used to represent *value* after *variable_name* has been defined. $(*variable_name*) is expanded to *value* at certain times during nmake processing, depending on the context and the assignment operator that was used. Variable expansion times are discussed later in this chapter.

A simple example of a variable definition is:

FILES = init.c main.c

NOTE:
To avoid conflict with variable names defined in the default base rules, a *variable_name* created by the user should *not* consist of a . (dot) followed by upper-case characters.

The following table lists the functionality of each of the five assignment operators that may be used to define a variable:

Table 3-1. Assignment Operators

variable = *value*	Every time the variable is referenced, it is replaced by its *value*. If a variable is on the right-hand side of the = operator, it is expanded when the variable on the left-hand side of the operator is evaluated. If the variable on the right-hand side is later changed, later expansions of the left-hand side variable will use the changed variable value.
variable := *value*	If *value* contains another variable, this variable is expanded at the time of assignment only. Later changes to any variable's value do not affect *variable*.
variable += *value*	This operator appends *value* list to the existing value of the variable. If *value* contains another variable, this variable is expanded and appended to the value of *variable*. Changes to this variable do not affect *variable*.
variable == *value*	This operator is the same as =, except that the variable is declared as a candidate implicit *state variable* (described later in this chapter). A state variable has its value and modification time preserved in the statefile; therefore, it can be a prerequisite or target of an assertion.
variable &= *value*	This operator is used to add an auxiliary value to a variable to which a primary value has already been assigned with the =, :=, or += assignment operator. The variables on the right-hand side are expanded before the assignment. The auxiliary value is not saved in the statefile.

Undefined Variables

nmake does not report on undefined variables. If you use a variable that has never been defined, it is not considered to be an error. The variable is expanded to the null string (the string with no characters).

Undefining a Variable

To undefine a defined variable, use the variable in an assignment statement without a value. For example, the following statement cancels the default value (-O) of CCFLAGS, thereby turning off optimization:

```
CCFLAGS =
```

The Scope of a Variable

The scope of a variable is global (i.e., the variable applies to all assertions in the makefile), unless the user specifies that it is local (i.e., the variable applies only to the current assertion) by using the local programming statement. For information about the local programming statement, see Chapter 7, *Programming in Alcatel-Lucent nmake*.

Variables assigned in a prerequisite list are defined while the action is expanded for the associated target and do not affect other targets in the same makefile. In the following example -g is only applied when generating x.o.

```
x.o : CCFLAGS=-g
```

Variable Precedence

Variable assignments come from many sources. You can define nmake variables in your shell environment, on the nmake command line, or in the makefile. If you do not define a variable on the nmake command line, nmake uses the definition found in the makefile. If the variable was not defined either on the command line or in the makefile, the environment variable definition is used. The precedence of these assignments, from highest to lowest, is shown below

.

Table 3-2. Variable Precedence

	Variable	**Description**
High	Automatic	Variable value maintained by nmake.
	Dynamic	Variable assignments done while building targets.
	Command line	Assignments specified on the command line.
Precedence	Import variables	Colon-separated environment variables listed in the value of the MAKEIMPORT variable (see the *Alcatel-Lucent nmake Product Builder Reference Manual*).
	Makefile	Variable assignments in a user makefile.
	Environment	Variables defined in the environment.
	Global rules	Variable assignments in a global makefile.
Low	Base rules	Variable assignments in a base rules makefile

Command-Line Variables

Command-line variable definitions override variable definitions in makefiles and the shell environment. They are included on the command line as an argument to the nmake command. For example:

```
$ nmake CC=icc
```

This command line redefines the variable CC (which is predefined in the default base rules) to have the value icc. When the variable CC is used in an action block, as in the following example,

```
main.o : main.c
        $(CC) $(CCFLAGS) -c main.c
```

the icc compiler will be used.

By using the variable to hold the name of the compiler, you can easily change which compiler is being used without modifying the rules to make your product or, as in the above example, without modifying the makefile.

All the assignment operators can be used when defining command-line variables, but care should be taken to quote any characters that are interpreted by the shell (i.e., &=).

Variable Expansion

This section discusses when variables are expanded, how to control when variables are expanded, and the expansion of different types of variables. It also provides some examples to illustrate the importance of controlling the time of expansion.

Expansion Times

The following terms must be defined before a discussion of variable expansion can be understood.

Compile Time	After a makefile has been read and parsed (top to bottom, left to right), all the information gathered is saved in binary form in a compiled makefile. In this process, the values of certain variables are expanded once and saved in the compiled makefile; these variables are considered *frozen*. The process of building the compiled makefile is called compile time.
Execution Time	After compilation, nmake makes the targets. The process of building the targets is called execution time.

During *compile time*, nmake reads the makefile and expands once the variables used in both targets and prerequisites but not in actions. At *execution time*, variables used in the action block are expanded. An illustration of this process is shown in Figure 3-1.

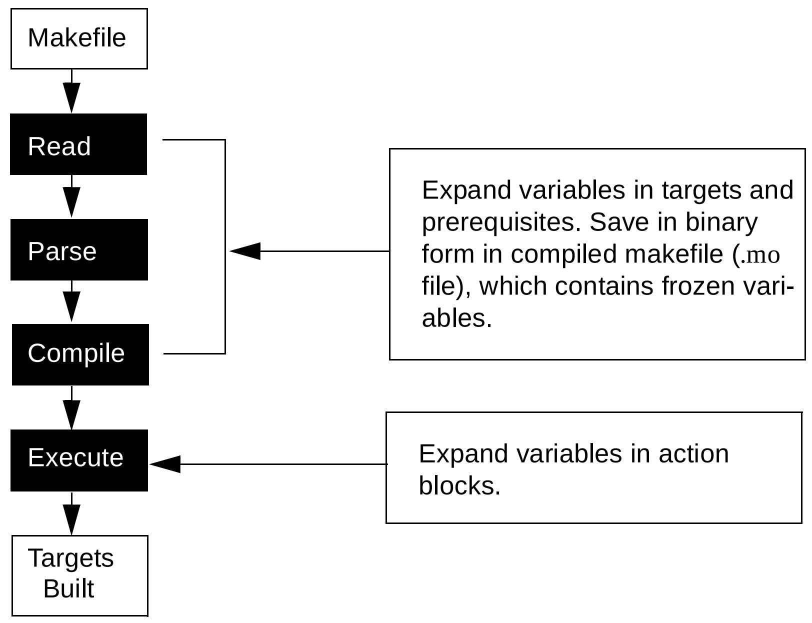

Makefile
Read
Parse
Compile
Execute
Targets Built
Expand variables in targets and prerequisites. Save in binary form in compiled makefile (.mo file), which contains frozen variables.
Expand variables in action blocks.

Figure 3-1. Variable Expansion Times

Delaying Expansion

It is possible to delay the expansion of a variable by inserting extra $s in front of the $(*variable_name*). The number of times the expansion of the variable is delayed corresponds to the number of $s inserted. For example, $$(*variable_name*) expands to $(*variable_name*) the first time it is expanded, which then expands to its value on the second expansion.

The following example is taken from the default base rules.

```
.INSTALL.%.COMMAND : $$(BINDIR)
```

The expansion of variable BINDIR is delayed once. When the makefile is parsed $$(BINDIR) is expanded to $(BINDIR) in the compiled makefile. Now BINDIR can be defined affter this assertion and it expands correctly during execution.

Expansion Types

There are three types of expansion:

- $(*var*),
- $("*string*"), and

- First non-null value.

They are explained in the following sections.

$(*var*) Expansion

All $(var) variables that occur in a target list or in a prerequisite list are expanded at compile time. Variables in action blocks are not expanded until execution time. Variables on the right-hand side of the = operator are expanded when the variable on the left-hand side of the operator is evaluated. Variables on the right-hand side of the := , += , and &= assignment operators are expanded immediately before the assignment. The following table provides a summary of this information.

Table 3-3. Variable Expansion Times

Variable	Expansion Time
target : $(*var*)	Compile time
$(*var*) : *prerequisite_list*	Compile time
x := $(*var*)	Immediately before the assignment
x += $(*var*)	Immediately before the assignment
x &= $(*var*)	Immediately before the assignment
x = $(*var*)	Expanded when x is evaluated
$(*var*) in action block	Execution time

$("*string*") Expansion

The following expression can be used anywhere in a makefile and is extremely useful in action blocks and assignment statements: $("*string*") expands to *string* where *string* is the literal string.

In the following example, drawn from Makerules.mk, the literal string is placed in an edit operation of an assignment statement. The current directory is represented as ".".

```
.CLOBBER. = $(".":L=*.(ii|l[hn])) core
```

The L edit operator is applied to the current directory to list all files with the extension .ii, .lh,or .ln. See Chapter 4, *Using Edit Operators,* for an explanation of variable editing and the L edit operator.

First Non-Null Value Expansion

nmake has a built-in construct for evaluating a list of token strings to the first non-null value. The following format:

$(*Var1* | *Var2* | *Var3* | ... |*VarN*) will evaluate to the first non-null variable in the list of variables *Var1* through *VarN*.

This construct can be used to ascertain that a particular variable has not been set, in which case an error condition may exist. For example, the following line:

```
echo $(MAJOR_PRODUCT | "error_condition")
```

evaluates to the value of MAJOR_PRODUCT, if that variable is non-null, or to the immediate value error_condition, if that variable is null.

Examples of Expansion

Example of the order of variable substitution:

```
FILES = init.c main.c
LIST = $(FILES)
FILES += error.c
convert :: $(LIST)
```

convert ends up with three prerequisites init.c, main.c and error.c because when $(LIST) is expanded, the value of FILES is init.c main.c error.c.

An example of the difference between = and := is:

```
FILES = init.c main.c
LIST := $(FILES)
FILES += error.c
convert :: $(LIST)
```

This example is the same example as above, except that := is used instead of = in the definition of LIST. In this case, convert has two prerequisites, main.c and init.c because the definition of LIST calls for immediate expansion of $(FILES) and FILES has as its value init.c main.c at expansion time.

All nmake variables are expanded before the action blocks are sent to the shell for execution. In the following example of a problematic makefile, nmake expands $(FILES) to convert.c in both action blocks. This set of assignments causes an inconsistency between the prerequisites of convert1 and what is used in the action block. See the next section for a discussion of automatic variables, which resolve this problem.

Consider the following problematic example:

```
FILES = init.c main.c
convert1 : $(FILES)
        cc -o convert1 $(FILES)
convert2 : $$(FILES)
        cc -o convert2 $(FILES)
FILES = convert.c
```

In this example, at compile time, nmake defines the prerequisites of convert1 to be init.c main.c . However, nmake does not know what the prerequisites of convert2 are because expansion of FILES has been delayed until execution time by using $$(FILES) instead of $(FILES). The last line in the makefile changes the value of FILES to convert.c. At execution time, convert1 has the prerequisites init.c main.c, and convert2 has the dependency $(FILES), which expands to convert.c.

CAUTION:
The order of the variable assignment and the use of the variable is important. Normally, you won't encounter this problem if you define all your variables first and then follow with your assertions and action blocks.

Automatic Variables

The base rules file defines a group of automatic variables that can be used in the action blocks of makefiles. These variables are expanded at execution time and provide a useful shorthand notation for parts of an assertion. Two of the most commonly used automatic variables are:

$(<)	expands to the name of the current target. The current target is the target currently being built (at any time there is only one target).
$(*)	expands to the list of all explicit file prerequisites of the current target.

For example, the makefile from the previous section can be rewritten as follows:

```
FILES1 = init.c main.c
FILES2 = convert.c
convert1 : $(FILES1)
        cc -o $(<) $(*)
convert2 : $(FILES2)
        cc -o $(<) $(*)
```

The most common automatic variables are shown in Table 3-4. The *Alcatel-Lucent nmake Product Builder Reference Manual* contains manual pages for all of them.

Table 3-4. Common Automatic Variables

Name	Expanded Value
$(<)	Current target
$(>)	Explicit file prerequisites younger than current target or new, or in a metarule the primary metarule prerequisite
$(*)	All explicit file prerequisites of current target

Table 3-4. Common Automatic Variables—*Continued*

Name	Expanded Value
$(~)	All explicit prerequisites of current target
$(!)	All implicit and explicit file prerequisites of current target
$(&)	All implicit and explicit state variable prerequisites of current target

The following is an example of the use of metarules and automatic variables to build a book. (Metarules are discussed in Chapter 2, *Understanding Makefiles*.)

```
%.tr : %.pr
        grap $(>) | pic |tbl | eqn > $(<)
book: chap1 chap2
        troff -Tpost $(*) | dpost | lp -dpost
chap1 : section1a.tr section1b.tr
        cat $(*) > $(<)
chap2 : section2a.tr
        cat $(*) > $(<)
```

This is how to interpret the example:

- The assertions are defined:

  The book is composed of chapters; each chapter is composed of sections. The sections are the .tr files.

- When a .tr file needs to be created or updated, the metarule is applied. The metarule rule is to run the .pr file, in turn, through the graphics processor, grap and its corresponding pic processor; put the result through the table processor tbl and then the equation processor, eqn. The result is the .tr file.

The book target is not physically created; rather, it is generated as output through *troff* and then physically output to a printer.

When this makefile (called test.mk) is executed using the following command line:

```
$ nmake -f test.mk
```

the execution output looks like the following:

```
+ grap section1a.pr
+ pic
+ tbl
+ eqn
+ 1> section1a.tr
+ grap section1b.pr
+ pic
+ tbl
+ eqn
+ 1> section1b.tr
+ cat section1a.tr section1b.tr
+ 1> chap1 + grap section2a.pr
+ pic
+ tbl
+ eqn
+ 1> section2a.tr
+ cat section2a.tr
+ 1> chap2 + troff -Tpost chap1 chap2
+ dpost
+ lp -dpost
request id is post-5890  (standard input)
$
```

Automatic variables can also be used in the form $(*c target_name*), where *c* is any automatic variable. For example, the following lines could be added to test.mk:

```
.DONE :
        : $(*book)
```

These lines would yield the following output:

```
+ : chap1 chap2
```

When an automatic variable is applied to a token list, the result is the union of the variable applied to each element of the list:

```
list = a.o b.o
targ :
        : star $(*$(list))
a.o : a.c
b.o : b.c

$ nmake
+ : star a.c b.c
```

Applying $(<) to a joint target will return all the joint target siblings of the specified target:

```
javafiles = a.java b.java c.java d.java
cls = a.class
targ :
        : javafiles $(<$(javafiles:D:B:S=.class))
a.class a1.class: a.java .JOINT
b.class b1.class b2.class : b.java .JOINT
c.class : c.java .JOINT
d.class : d.java

$ nmake
+ : javafiles a.class a1.class b.class b1.class b2.class c.class d.class
```

Empty Expansions

A rebuild may result in empty $(>) expansions in non-metarule actions. As an example of this situation, consider the following makefile (this line was taken from the previous example):

```
chap1 : section1a.tr section1b.tr
        cat $(>) > $(<)
```

The $(*) automatic variable from the previous example was replaced with the $(>) automatic variable. The $(>) automatic variable expands to explicit file prerequisites younger than the current target or are new prerequisites. If the chap1 target was touched (therefore changing the time-stamp), on subsequent invocations of nmake, the $(>) automatic variable would be empty because there would not be any explicit prerequisites younger than the current target. The empty $(>) variable would cause an empty file to be written to chap1. Note in metarules $(>) is never empty.

State Variables

While the prerequisites of a target are most commonly files, they may also be variables. Variables that are used as prerequisites are called *state variables*. State variables are stored in the makefile statefile (*makefile*.ms) so that their values and modification time are preserved between nmake invocations.

State variables are frequently used as prerequisites of an assertion when the target is dependent on the last modification time as well as the value of the variables. Typically, the variables defining the name of the compiler, the default preprocessing definitions, and the flags for the link loader are used as prerequisite state variables for each atom representing an object module. Occasionally, a state variable is a target atom whose value is determined after its action has triggered.

By default, nmake scans .c files for implicit file prerequisites (header files) and for implicit state variables. Therefore it is normally not necessary to list implicit state variables in a prerequisite list.

A variable is declared to be a state variable in one of the following ways:

- By enclosing the variable in parentheses and using it as a prerequisite, for example:

  ```
  targ :: targ.c (X)
  targ1 : (X)
  ```

- By assigning the .STATE Special Atom to the variable, for example:

  ```
  .STATE : X
  ```

- By assigning a value to the variable with the == operator, for example:

  ```
  X == 100
  ```

Implicit State Variables

If a state variable is an implicit prerequisite of a target and its value changes, the target is rebuilt. The preprocessor uses the -D flag to define the variable and put the flag into the compilation command line (using CCFLAGS). CCFLAGS is discussed in more detail in the next section.

Consider the following example.

Contents of Makefile

```
MAX == 100
convert :: init.c main.c
```

Contents of **main.c**

```
main()
{
        init();
}
```

Contents of **init.c**

```
init(){
   printf("MAX is %d\n", MAX);
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -DMAX=100 -c init.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C\
/pp/B49DF4E0.bincc -I- -c main.c
+ cc -O -o convert init.o main.o
```

Run nmake again

```
$ nmake MAX=50
```

Output

```
+ cc -O -Qpath /nmake3.x/lib \
-I-D/nmake3.x/lib/probe/C/pp/B49DF4E0.bincc -I- -DMAX=50 \
-c init.c
+ cc -O -o convert init.o main.o
```

MAX is declared as a state variable in the makefile using the == assignment operator, and is assigned a value of 100. In addition, MAX is used in init.c. When nmake scans init.c and finds that the MAX state variable is used there, nmake automatically considers MAX as an implicit prerequisite of init.c and includes -DMAX=100 on init.c's compilation line. Since main.c does not use MAX, nmake does not consider MAX as a prerequisite and will not include

-DMAX=100 on main.c's compilation line. On further invocations of nmake, if the value of MAX has changed, init.c will be recompiled.

NOTE:
The .PARAMETER Special Atom can be used to suppress the generation of -D flags for certain state variables. If you do not want MAX expanded with -D in CCFLAGS, you can add the following line in the makefile:
(MAX) : .PARAMETER (See the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information about the .PARAMETER Special Atom.) Also, note that parentheses are required when using a state variable on the left-hand side of an assertion.

State Variables with Default Values

Certain state variables (including some which correspond to UNIX system commands) are defined in the default base rules file. See the *Alcatel-Lucent nmake Product Builder Reference Manual* for a complete list of the variables that are assigned default values in the base rules file.

By convention, a UNIX system command is stored in a variable called *command* (for example, cc is stored in a variable called CC), and flags to that command are stored in another variable called *commandflags* (for example, -O is stored in a variable called CCFLAGS). Also by convention, in the rule to make a target, the default base rules use $(*command*) and $(*commandflags*) in action blocks, with (*command*) and (*commandflags*) as prerequisites of the target. For example, consider a metarule that is defined in the default base rules:

```
CC = cc
CCFLAGS = -O
%.o : %.c (CC) (CCFLAGS)
        $(CC) $(CCFLAGS) -c $(>)
```

Here CC and CCFLAGS are state variable prerequisites of the target and $(CC) and $(CCFLAGS) appear in the action block. In this example, changing the value of CC or CCFLAGS causes the product to be rebuilt.

Contents

Contents

Using Edit Operators

4

Editing Variables

As we have seen in the preceding chapter, a variable can be assigned a list of values (e.g., FILES = a.c b.c c.c). This list of values is called the *value_list*. nmake provides approximately 60 edit operators that are used to manipulate a variable's *value_list* when the variable is expanded. This manipulation is called *variable editing*. Each string, separated by white space, contained in the *value_list* for the variable is called a *token*. Variable editing can use pattern-matching schemes to manipulate the tokens. For example, if a variable has a list of file names for its value (each file name separated by white space), when that variable is expanded, each token that matches a pattern can be selected, leaving out those tokens that do not match the pattern. This variable-editing capability constitutes an important part of the nmake programming language called the *edit operators*.

Edit operators can be placed in a contiguous sequence to form arbitrarily complex editing operations. Conceptually, this sequencing is similar to using a pipeline at a UNIX shell prompt, because the output of earlier edit operators serves as the input for later edit operators.

This chapter describes the syntax and rules for using edit operators and provides examples of the use of four important types of edit operator. For examples of the use of other edit operators, see the *Alcatel-Lucent nmake Product Builder Reference Manual,* which contains manual pages for all the edit operators provided with nmake. In addition, the default base rules make extensive use of edit operators. You are encouraged to examine the default base rules to see how the edit operators are used.

Edit Operation Syntax

The general syntax for editing a variable is:

$(*variable_name***:***edit_operation***:***edit_operation***...)**

The general syntax for the *edit_operation* is:

[@]*edit_operator***[***separator argument***]**

where *edit_operator* is a token representing the name of a valid edit operator.

In the following example, the :N edit operator applies shell pattern matching and returns the matching tokens of variable FILES.

Contents of Makefile

```
FILES = a.c b.y c.c e.s
targ :
      : $(FILES:N=*.c)
```

Run nmake

```
$ nmake
```

Output

```
+ : a.c c.c
```

Edit Operation Rules

The following rules apply to edit operations associated with the syntax above:

- Double quotes (""), single quotes (’’), or backslash (\) characters used to enclose strings containing spaces are considered part of the token and are not removed.
- If an at sign (@) immediately precedes an edit operator name, the entire value is treated as a single token for that edit operator.
- The separator (*separator*) is usually an equal sign (=). Where appropriate, some edit operators support other arithmetic comparison signs such as not equal (!=), less than (<), less than or equal (<=), greater than (>), and greater than or equal (>=).
- *argument* is the argument for the edit operator, if applicable.
- Some operators select tokens by returning the token value as the result. Nonselected tokens produce an empty string (null).

- The edit operators, each preceded by a colon (:), are applied in order from left to right to each token in the expanded variable value. The edit operators form a token pipeline in which the output (returned or selected tokens) of any edit operator becomes the input for the next edit operator. The *newline* is treated as a separate token.
- The edit operator delimiter (:) can be redefined using a backtick (`) followed by the desired delimiter character. If there is more than one edit operator it must be redefined at the first operator and the new delimiter used for the subsequent operators. This is useful when a colon (:) appears somewhere in the edit string.

  ```
  var = ab c:d ef g:h ij
  $(var`;N=*:*)
  ```
- The expansion algorithm recursively expands the *variable_name* to determine the tokens in the *value_list* for that variable.

  As an example, consider the following assignment statements:

  ```
  var = a b $(c)
  c = d e f
  ```

  When the var variable is recursively expanded, the *value_list* for var contains the following: a b d e f.
- The operator expressions (*separator argument*) are then expanded before the edit operators are applied. The ultimate expansion is formed by applying each edit operator to each token, separating adjacent results by a single space character.

  As an example, consider the following example, which uses the X edit operator. The syntax of the X edit operator is as follows:

  ```
  $(S:X=$(cross))
  ```

  The X edit operator computes the cross product of the tokens that are directory names in variable S and the tokens in the expanded value of cross.

  In the example below, the action block contains the statement: $(start:X=$(dirs)), where *separator* is = and *argument* is $(dirs). This statement expands to /usr/project/include /usr/project/lib

Contents of Makefile

```
start = /usr
dirs = project/include project/lib

targ :
      : $(start:X=$(dirs))
```

Run nmake

```
$ nmake
```

Output

```
+ : /usr/project/include /usr/project/lib
```

Using File Component Edit Operators

The most commonly used edit operators are file component edit operators. With the string /mygroup/myname/src/convert.c as an example, the file components are as shown in Table 4-1

Table 4-1. File Component Edit Operators

Edit Operator	Meaning	Value
D	Directory name	/mygroup/myname/src
B	Basename	convert
S	Suffix name	.c

NOTE:
With abbreviations for libraries, such as -lm or -lld, nmake assumes that there is only a base name and no directory or suffix; therefore, do not use these operators when specifying libraries.

In the previous section, we mentioned that edit operators form token pipelines. The file component edit operators (D, B, and S) are an exception to this rule; they do not form token pipelines because nmake treats the file component edit operators as a group. The pipelines for the D, B, and S edit operators are different.

Suppose that you want to determine the basename of the parent directory from the variable PWD. $(PWD:D) evaluates to the full path of the parent directory. $(PWD:B) evaluates to the basename of the current directory, which we don't want. $(PWD:D:B) evaluates to the full path of the current directory, which again we don't want.

As shown, the edit operations do not work as desired; therefore, the B evaluation must take place on the results of the D evaluation. The proper specification for this example is $(PWD:D::B) where the :: forms a separate pipeline to the previous edit operation from the next. This case can be better understood by looking at the following example.

In this example, the current working directory, that is, the directory that contains the makefile and from where nmake is executed is /usrs1/A/new.p/chk.dir.

Contents of Makefile

```
targ:
    : D = $(PWD:D)
    : B = $(PWD:B)
    : S = $(PWD:S)
    : D:B = $(PWD:D:B)
    : D:B:S = $(PWD:D:B:S)
    : D::B = $(PWD:D::B)
    : D::S = $(PWD:D::S)
    : D:S = $(PWD:D:S)
    : D::B:S = $(PWD:D::B:S)
    : D::B::S = $(PWD:D::B::S)
```

Run nmake

```
$ nmake
```

Output

```
+ : D = /usrs1/A/new.p
+ : B = chk
+ : S = .dir
+ : D:B = /usrs1/A/new.p/chk
+ : D:B:S = /usrs1/A/new.p/chk.dir
+ : D::B = new
+ : D::S = .p
+ : D:S = /usrs1/A/new.p/.dir
+ : D::B:S = new.p
+ : D::B::S =
```

As mentioned previously, the desired file suffix can be specified as the argument of the edit operation. Consider the following example of file component editing:

```
FILES = init.c main.c
convert : $(FILES:B:S=.o)
        cc -o $(<) $(*)
```

In this makefile, as $(FILES) is expanded, the basename of all the strings is kept, and the suffix of all the strings is changed to .o. Therefore, nmake expands $(FILES:B:S=.o) to init.o main.o.

The following example illustrates a method of file backup and restoration using file component edit operators.

Two variables, BACKDIR and BACKLOG are defined to hold the backup directory name and the backup log file name. The backup target has (backup) as a prerequisite. This is a *state variable*; its state is saved in the statefile. The time of last update is preserved in the statefile and governs the backup action. The backup action block copies to the backup directory all of its prerequisites that have changed since the last time backup was built.

$(>) expands to the list of file prerequisites that caused the current target to be out of date. The action block then logs the backup in the backup log.

To restore the files from backup, all of the source files are copied from the backup directory to the current directory. A log is again entered in the backup log. silent is a shell script included in the nmake package that causes the command line not to be echoed to the terminal as the command is executed.

Contents of Makefile

```
FILES = init.c main.c
BACKDIR = BAK
BACKLOG = backup.log

backup : (backup)

(backup) : $(FILES)
      $(CP) $(>) $(BACKDIR)
      silent echo `date` ": $(>) backed up" >> $(BACKLOG)

restore : $(FILES:D=$(BACKDIR):B:S)
      $(CP) $(*) .
      silent echo `date` ": $(*:B:S) restored" >> $(BACKLOG)
```

Run nmake

```
$ nmake backup
```

Output

```
+ cp init.c main.c BAK
```

Change time-stamp

```
touch init.c
```

Run nmake

```
$ nmake backup
```

Output

```
+ cp init.c BAK
```

Run nmake

```
$ nmake restore
```

Output

```
+ cp BAK/init.c BAK/main.c .
```

The file backup.log from the previous session contains the following output that was generated by nmake:

```
Thu Dec 9 09:39:06 EST 1993 : init.c main.c backed up
Thu Dec 9 09:39:30 EST 1993 : init.c backed up
Thu Dec 9 09:39:43 EST 1993 : init.c main.c restored
```

Using Pattern Recognition Edit Operators

The edit operators for pattern matching are L, M, and N. The following sections explain their functioning and give examples of their use.

The L Edit Operator

The syntax of the L edit operator is:

L [<>]= *pattern*

This edit operator lets you select files from a list of directories. Each token is considered to be a directory name and the outcome of the token manipulation is a list of files in the token(s) that match the *pattern*. The optional use of the < and > signs allows you to specify that the file list is to be sorted in ascending order (<) or descending order (>). The default is unsorted.

Using < or > without the = sign returns only the first token from the list; that is, either the first token (<) or the last (>).

Consider the following example. In this example, the .SOURCE Special Atom defines two directories (dir1 and dir2) that are to be searched for source files. The DIRS variable contains the explicit prerequisites of .SOURCE, which are dir1 and dir2. The line: FILES = $(DIRS:L<=*.c) produces a list of C files that exist in all of the directories listed in $(DIRS).

Contents of Makefile

```
.SOURCE  : dir1 dir2
DIRS= $(*.SOURCE)
FILES = $(DIRS:L<=*.c)

tst :
        : Directories to be searched are: $(DIRS)
        : C files found are: $(FILES)
```

Run nmake

```
$ nmake
```

Output

```
+ : Directories to be searched are: . dir1 dir2
+ : c files found are:  a.c b.c c.c d.c e.c main.c
```

The M Edit Operator

The syntax of the M edit operator is:

M [!]= *pattern*

All tokens that match (=) the pattern, or do not match (!=) the pattern, are selected. The patterns are egrep(1)-style regular expressions.

Consider the following example. This makefile checks whether the current directory starts in /usrs1 and ends in directory tst. If it does not, the string *not in correct directory* is displayed.

Contents of Makefile

```
dir_verify:
      if [ "$(PWD:M=^/usrs1/.*/tst$))" = "" ]
      then
            $(SILENT) echo not in correct directory
      fi
```

Determine present working directory

```
$ pwd
```

UNIX System Response

```
/home/dev
```

Run nmake

```
$ nmake
```

Output

```
+ [  =  ]
      not in correct directory
```

The N Edit Operator

The syntax of the N edit operator is:

N [!]= *pattern*

All tokens that match (=) the pattern, or do not match (!=) the pattern, are selected. Patterns are generated using the same conventions used by the shell for file name generation (see the ksh(1) manual page for information on file name generation).

For example:

```
FILES = init.c main.c Makefile convert.1
convert : $(FILES:N=*.c)
            cc  -o $(<) $(*)
```

In this makefile, the variable FILES lists all the files associated with convert but only the files matching the pattern *.c are the prerequisites of convert. So the matched files are init.c and main.c.

Using the Pattern Substitution Edit Operator

Earlier, we saw the capability of token selection using predefined patterns, such as file name suffix, directory, and so on. nmake also supports a sed-like facility for pattern substitution. The edit operator is C. The general form is:

$(*variable***:[@]C***<del>old<del>new<del>***[g])**

or

$(*variable***:[@]***/old/new/***[g])**

The edit operation is applied to each token. The string specified as *old* is replaced by the string specified as *new*. If a trailing g is specified, all occurrences of the string that is specified as *old* are replaced by the string that is specified as *new*. The character following C is used as the delimiter (<*del*>). Any character can be specified as <*del*> to be used as the delimiter. C/ can be abbreviated as / as shown in the second line of the syntax described above.

Like sed, the substitution applies to all instances of *pattern* when the g is present; otherwise, it applies to the first instance only. The *pattern* is any ed(1) pattern-matching expression and just as in ed, the backslash (\) is used as an escape character.

Consider the following example. In this example, the .SOURCE Special Atom defines two directories (dir1dir and dir2dir) that are to be searched for source files. The DIRS variable also contains this list of directories. The remaining lines perform string substitutions on the directory list contained in DIRS. The output is explained below.

Contents of Makefile

```
.SOURCE : dir1dir dir2dir
DIRS = $(*.SOURCE)

tst :
      : DIRS = $(DIRS)
      : NEWDIR0 = $(DIRS:C/dir/new/)
      : NEWDIR1 = $(DIRS:/dir/new/)
      : NEWDIR2 = $(DIRS:C/dir/new/g)
      : NEWDIR3 = $(DIRS:C?dir?new?g)
      : NEWDIR4 = $(DIRS:@/dir/new/)
      : NEWDIR5 = $(DIRS:@/dir/new/g)
```

Run nmake

```
$ nmake
```

Output

```
+ : DIRS = . dir1dir dir2dir
+ : NEWDIR0 = . new1dir new2dir
+ : NEWDIR1 = . new1dir new2dir
+ : NEWDIR2 = . new1new new2new
+ : NEWDIR3 = . new1new new2new
+ : NEWDIR4 = . new1dir dir2dir
+ : NEWDIR5 = . new1new new2new
```

Let's examine the output in detail.

- Output line: + : DIRS = . dir1dir dir2dir

  The DIRS variable contains the list of directories, which includes the current directory and the directories dir1dir and dir2dir.

- Output line: + : NEWDIR0 = . new1dir new2dir

  Only the first occurrence of the string dir in each token was changed to the string new.

- Output line: + : NEWDIR1 = . new1dir new2dir

  Only the first occurrence of the string dir in each token was changed to the string new because :C/ can be abbreviated to /.

- Output line: + : NEWDIR2 = . new1new new2new

  Every occurrence of the string dir in each token was changed to the string new because the trailing g is specified.

- Output line: + : NEWDIR3 = . new1new new2new

  Every occurrence of the string dir in each token was changed to the string new because the trailing g is specified. This line shows that the delimiter can be any character.

- Output line: + : NEWDIR4 = . new1dir dir2dir

Only the first occurrence of the string dir in the first token was changed to the string new because @ is used to cause $(DIR) to be treated as a single token.

- Output line: + : NEWDIR5 = . new1new new2new

  Every occurrence of the string dir in each token was changed to the string new because the trailing g is specified.

Another example of the use of pattern substitution is:

```
LIBRARY = libc.a
LIB = $(LIBRARY:C/lib/-l/:B)
```

This substitution results in LIB = -lc. The substitution transforms libc.a into -lc.a; then the B edit operator retains only the base file name, dropping the .a and leaving -lc.

Using the Type Recognition Edit Operator

Tokens can be recognized in terms of *type* by the edit operator T. The general format of this edit operator is:

$(*variable***:T=***type***)**

Table 4-2 contains a small sample of variable types.

Table 4-2. Variable Types

Type	What it Matches
D	Matches a state variable (expanded using state variable definition)
F	Matches a file name
G	Matches a file that has already been built or generated.

Consider the following example. In the output, the T=D edit operator returns the cc(1)-style command-line flag definition.

Contents of Makefile

```
.SOURCE : src
Files = a.c (var)
var = b.c
tst:
        : $(Files:Q)
        : $(Files:T=F)
        : $(Files:T=D)
```

Run nmake

```
$ nmake
```

Output

```
+ : a.c (var)
+ : src/a.c
+ : -Dvar=b.c
```

The Q edit operator is needed in the first statement to protect the parentheses from the shell, which would attempt to interpret them. The directory name is included in the second statement because F returns the bound file name.

Contents

Contents

Using Atoms and Special Atoms

5

This chapter discusses both atoms and Special Atoms in Alcatel-Lucent nmake. A description of atoms is presented first, providing the foundation that is required for understanding Special Atoms.

Special Atoms are used to control nmake's processing and are a primary programming resource under nmake. There are nearly 100 Special Atoms defined in nmake. This chapter discusses a few of the most commonly used Special Atoms and provides examples to demonstrate their use. Manual pages for all the Special Atoms appear in the *Alcatel-Lucent nmake Product Builder Reference Manual*.

Atoms

An atom is any element that appears on either the left-hand side or the right-hand side of an assertion operator in an assertion, i.e., an atom can be in the target position or the prerequisite position of an assertion. In addition, implicit and explicit prerequisites are considered to be atoms, as are state variables.

The components of an atom are listed and described in the following table.

.

Component	**Description**
Unbound Name	The name that appears in the assertion statement in the makefile.
Bound Name	The full path that is determined during the binding process if the atom can be bound to a file. The binding process is described in the following section.
State Value	The value that is assigned to a state variable. (This component applies only to state variables.)
Time-Stamp	The last modification time the atom was bound by nmake.
Action	The action specified in the assertion.
Attribute List	The Special Atoms that are assigned to an atom by nmake or by an assertion based on the pattern of the file name (prefix, stem, suffix), position in an assertion (left-hand side or right-hand side), binding status (bound or unbound), etc. This list of Special Atoms describes the properties inherited by the atom. Only a subset of Special Atoms can become properties of an atom. The terms *Special Atom* and *attribute* are used synonymously when referring to the Special Atoms that can appear in the *attribute list*.
Prerequisite Atom List	The list of implicit and explicit prerequisites including files and state variables. Special Atoms are not included in the *prerequisite atom list*. Each of the atoms in the prerequisite atom list is subject to nmake processing (e.g., include files).

Some of the components of an atom may be null. For example, the prerequisite atom list is null for an atom that does not depend on any other atom.

Binding

At compile time, nmake treats all atoms as character strings. At execution time, nmake determines the *object type* (described in the following section) for each of the atoms in an assertion. (For a disucssion of compile time and execution time, see the section *Variable Expansion* in Chapter 3, *Using Variables*.) In addition, nmake assigns other attributes to the atom depending on the atom's object type. These attributes are stored in the *attribute list* component of the atom. The assigned attributes are used internally by nmake. The process of determining an atom's object type and assigning attributes is called *binding*. Once bound, an atom remains bound until nmake exits, unless it is explicitly unbound.

Atom Object Types

nmake asssigns one of the following four object types to atoms used in assertions; *regular file*, *state variable*, *virtual atom*, and *label*. The following sections describe each of these types.

Regular File Object Type

nmake assigns the *regular file* object type to any atom that is either a pre-existing file or a file that was built during the execution of nmake. (The file must not be a directory or a special file.) The regular file object type is the most common atom object type used by nmake. After binding, the atom's bound name is the full path to the regular file. The atom's time-stamp is the last modification time of the file.

State Variable Object Type

nmake assigns the *state variable* object type to any atom that is a state variable. . State variable atoms are frequently used as prerequisites of an assertion in which the target is dependent on the value as well as the last modification time. Typically, the name of the compiler used and the name of the compiler and its flags are used as prerequisite state variables for each atom representing an object module. Occasionally, a state variable is a target atom whose value is determined after its action has triggered. Once the state variable is bound, the *state value* component is the value that was assigned to the state variable. The atom's *time-stamp* component is the last modification time that a new value was assigned to the state variable.

CAUTION:
When a state variable with no prerequisites is used as a target, the entire assertion is ignored.

Virtual Atom Object Type

nmake assigns the *virtual atom* object type to atoms that have been assigned the .VIRTUAL Special Atom in an assertion. This object type does not represent a regular file, a state variable, or a label (described below).

A target atom marked by the .VIRTUAL Special Atom is not made and is never bound to a file. This property allows a list of prerequisites to be maintained even when the target is not built.

For example, consider the following assertion:

```
targ : prereq1 prereq2 .VIRTUAL
        : (targ is never generated)
```

In this example, after the binding process is done, the .VIRTUAL Special Atom is included in the attribute list for the atom, targ. The .VIRTUAL Special Atom is a Special Atom that causes the target to have the virtual atom object type. The *unbound name* (i.e., the name specified in the makefile) and the last modification time of a virtual atom are stored in the statefile and are thus available for the next invocation of nmake.

Label Object Type

nmake assigns the *label* object type to any atom that does not represent a regular file, a state variable, or a virtual atom. This type of atom is always in the target position of the assertion. An atom that is in the target position represents a label if the assertion has no prerequisites but has an action block, or if the assertion has prerequisites but no actions and the target is not a regular file, virtual atom, or state variable.

For example, consider the following assertion:

```
targ :
        : Echo this string. Making $(<)
```

The targ target in the assertion has the label object type after the binding process is done.

The time-stamp of the label is stored in the statefile but is ignored by nmake; therefore, labels are always made on subsequent invocations of nmake.

How nmake Binds an Atom

When nmake parses a makefile for the first time, atom object types have not yet been determined; therefore, the atoms are in an unbound state and assigned the .UNBOUND Special Atom. In addition, if an atom is a target of an assertion, then the atom is also assigned the .TARGET Special Atom. At execution time, nmake must bind all the atoms that have a .UNBOUND Special Atom. Once the atom's object type has been determined, the .UNBOUND Special Atom is removed and other attributes related to the object type are assigned. As the binding status of the atom changes, so will the atom's *attribute list*.

The following sections describe the process of binding a target or prerequisite atom to each of the four object types.

Binding a Target/Prerequisite to Regular Files

An atom is bound to a regular file if the *unbound name* of the atom can be found as a regular file (not a directory or a special file). If the file exists, the atom is assigned the .REGULAR Special Atom to describe its object type. The last modification time of the file becomes the atom's time-stamp component. The full path to the regular file is assigned as the bound name component of the atom.

In the process of binding an atom, nmake searches the current directory first by default. If a makefile contains the .SOURCE.*pattern* Special Atom, where *pattern* is the specified file suffix, the directories that are specified as prerequisites to the .SOURCE.*pattern* Special Atom are searched when nmake looks for files with that suffix. If a makefile contains the .SOURCE Special Atom, the directories that are specified as prerequisites to .SOURCE are searched when nmake looks for files with any suffix. See the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information concerning the .SOURCE.*pattern* and .SOURCE Special Atoms.

Win32 platforms have some special binding rules. On win32 the .exe filename suffix is implicit so atoms will bind to file *filename*.exe if it exists or *filename* otherwise. Case insensitivity is also supported on win32.

Consider the following makefile:

```
.SOURCE : src1 src2
targ :: main.c input.c
```

When nmake processes this makefile, the following events occur, though not necessarily in the order in which they are listed:

- At compile time, the targ atom is assigned the .TARGET Special Atom because it is in the target position. The .UNBOUND Special Atom is assigned to targ, main.c, and input.c because their object types are not yet known.
- During nmake execution, the current directory and the src1 and src2 directories are searched when nmake tries to bind the unbound atoms targ, main.c and input.c to regular files.
- At execution time, as nmake attempts to make the targ target, nmake realizes that the atoms main.c and input.c have not yet been bound; therefore, nmake must bind these atom before continuing. In attempting to bind these atoms, nmake realizes that the atoms main.c and input.c are not virtual atoms since they do not have the .VIRTUAL Special Atom. nmake also knows that they are not state variables since they do not appear in either the *(variable)* format or the *variable* == *value* format. nmake will therefore attempt to bind the atoms to regular files whose bound names are assigned as full path names to main.c and input.c.

- If any of the prerequisite files that nmake looks for do not exist, then the atom's *unbound name* is compared to the implied metarules (many metarules are predefined in the default base rules) to see whether the atom can be made from an implied prerequisite. If there are no rules defined to make this atom, nmake stops processing. For example, if the file main.c cannot be found or made from an implied prerequisite, nmake stops processing and then displays the following message:

  don't know how to make: targ : main.o : main.c

  Notice in the message that the dependency information generated by nmake is displayed for the main.c prerequisite.

- If the file that nmake is looking for exists, the atom is bound. When an atom is bound, the .UNBOUND Special Atom is removed and the .REGULAR Special Atom is assigned. In addition, the .EXISTS Special Atom is applied to the atom to inform nmake that this atom binds to an existing file. This information is needed by nmake for the cases where a target is dependent on a prerequisite, which is dependent on another prerequisite.

- When the atom is found (as a file or is made by applying a matching metarule), the atom is bound to a regular file and the attribute list for the atom is checked for a .SCAN Special Atom. This attribute specifies the scan strategy that should be used in case the source file has any implied prerequisites, such as header files. In our example, the main.c and input.c atom are assigned the .SCAN.c Special Atom by the following rule defined in the base rules:

  .ATTRIBUTE.%.c : .SCAN.c

  This rule states that any file that matches the file pattern %.c is automatically assigned the .SCAN.c Special Atom.

  Any atom that has a .SCAN Special Atom in its attribute list must be scanned for implicit prerequisites before any other action can be done. For example, if main.c includes a header file main.h, main.h becomes an implied prerequisite of the atom main.c. After the scanning process has been completed, the atom main.c is assigned the .SCANNED Special Atom to inform nmake that this atom has already been scanned for the implied prerequisites.

- nmake will generate a state atom whose name corresponds to the *unbound name* where the state atom's name is prefixed with either paired parentheses or an unpaired parenthesis. For example, a state atom that is prefixed with paired parentheses would appear as follows: ()*atom* where *atom* is the name of the prerequisite atom. In our example, nmake will generate a state atom called ()main.c for the prerequisite atom main.c The state atom contains all the information that is to be saved in the statefile. The only way to access this information from within a makefile is by using the T=SR edit operator, such as in $("main.c":T=SR). The .BUILT Special Atom is assigned to state atoms that were built in either the previous or current execution of nmake. For example, to see whether an atom has been built, the evaluation can be specified as $("main.o":T=SR:A=.BUILT), which evaluates to the name of the state atom for main.o if the main.o target has been made. (See the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information concerning the T=SR primary state name edit operator.)

- The binding process may continue if special operations have been specified in the global makefile or the local makefile by using the .ATTRIBUTE Special Atom. These special operations may include assigning attributes to atoms whose names match the file pattern specified in the .ATTRIBUTE.*pattern* Special Atom.

  For example, the following assertion could be specified in the makefile to create .C_src as a user-defined attribute:

  ```
  .C_src : .ATTRIBUTE
  ```

  (Defining your own attributes is described in greater detail at the end of this chapter and is presented here only to show how attributes can be added to an atom's attribute list.)

  Once .C_src is declared as a user-defined attribute, the following assertion could then be specified to append .C_src to the *attribute list* for each of the atoms whose names match the file pattern %.c.

  ```
  .ATTRIBUTE.%.c : .C_src
  ```

The nmake interactive tracer can be used to display the attributes that have been assigned during nmake execution. The interactive tracer is discussed in the section *Invoking the Interactive Tracer*, later in this chapter.

The time-stamp for a file can be displayed by nmake. For example, the evaluation of $("main.c":T=R:F=%T) would return the time-stamp from the main.c atom, which is its last modification time.

NOTE:
An atom is not the same thing as a variable. Trying to specify $(main.c) would evaluate to null.

Binding Library Prerequisites

When nmake binds a library prerequisite to a regular file the current directory is searched first, then the .SOURCE.a directories and then the .SOURCE directories. Note that the .SOURCE.a directories are searched for both archive and shared libraries, there is no need to define a .SOURCE.*pattern* for shared libraries.

If the library prerequisite is specified as -l*nnn,* nmake will first bind to a shared library if it exists, otherwise to an archive library. The search scheme for a corresponding file is: lib*nnn*$(CC.SUFFIX.SHARED), *nnn*$(CC.SUFFIX.SHARED), lib*nnn*$(CC.SUFFIX.ARCHIVE), *nnn*$(CC.SUFFIX.ARCHIVE). On win32 platforms *nnn.suffix* is searched before lib*nnn.suffix*.

For +l*nnn* prerequisites, nmake will first bind to an archive library if it exists, otherwise to a shared library. This only affects library *nnn* and not any required libraries of *nnn*.

When binding a library nmake will check for optional required libraries which will also be bound and included as prerequisite libraries. There are two methods for which required libraries may be specified, the required-libs file, and the CC.REQUIRE.*name* probe variable. For bound library $(LIBDIR)/libnnn.a*,* the corresponding required-libs file would be $(LIBDIR)/lib/nnn and would contain one library per line. The required-libs file is generated by the :LIBRARY: assertion operator when a target library is made. See the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information concerning :LIBRARY:. The corresponding probe variable for libnnn.a would be CC.REQUIRE.nnn and would be defined to a list of libraries to include when -lnnn is specified as a prerequisite. See *Appendix A* for more information on probe.

Binding a Target/Prerequisite to State Variables

A target or a prerequisite specified as (*var*) is considered a state variable; therefore, nmake retains the state variable's time-stamp of last modification (last time that a value was assigned) and the current value (called the *current state*) in the statefile.

If the *unbound name* matches the pattern (*var*) where *var* is a string, this unbound atom will be assigned the .STATEVAR Special Atom. The same thing occurs to a variable that is assigned a value with the == assignment operator. In addition, nmake considers the variable a state variable and therefore will assign the .SCAN.STATE Special Atom to this atom.

When an atom is bound to a state variable and the state variable atom does not have the .TARGET Special Atom, nmake notes the *state value* of the variable as the value assigned in the makefile or on the command line and assigns the atom the .EXISTS Special Atom. Otherwise, if the state variable atom has been assigned the .TARGET Special Atom, the state value of the variables is not determined until the atom is made. During the binding process, nmake also records in the time-stamp component when the state variable was actually bound.

Occasionally it is necessary for state variable atoms to be assigned a state value as the result of an action block being triggered. In such cases, nmake does not determine the state value of the state variable atom until after the action has triggered.

Binding a Target/Prerequisite to Virtual Atoms

An atom is considered to be a virtual atom if the atom has the .VIRTUAL Special Atom. An atom can be assigned the .VIRTUAL Special Atom through an assertion statement in the makefile.

When the assertion specification is compiled, the atom is assigned the .UNBOUND, .VIRTUAL, and .TARGET Special Atoms. When nmake binds the atom, the .UNBOUND Special Atom is removed from the attribute list.

Binding a Target to a Label

A target is considered a label if it is not bound as a regular file, a virtual atom, or a state variable. At compile time, a target is assigned the .UNBOUND and .TARGET Special Atoms. When nmake binds the atom, the .UNBOUND Special Atom is removed from the attribute list.

Special Atoms

Special Atoms are used to provide fine control over assertions, actions, and nmake processing. Any atom that has a Special Atom in its attribute list is treated differently during compilation or execution.

A Special Atom is similar to an atom in the respect that a Special Atom may appear on either the left-hand side or the right-hand side of an assertion operator in an assertion statement. However, Special Atoms cannot be categorized as being any one of the four atom object types previously mentioned and typically do not have all the components of an atom. Special Atoms can be assigned dynamically by nmake or be specified in makefiles by the user.

A Special Atom may or may not be a property of another atom. For example, the .VIRTUAL Special Atom may be assigned to the attribute list of an atom, thereby becoming a property of that atom, but the .SOURCE Special Atom (described previously) can never become a property of another atom. Its function is simply to specify the location of source files.

The Special Atoms defined by nmake all have the form *.ID*, where *ID* is any string of capital letters or dots, but users may give their own Special Atoms any names they choose.

NOTE:
The name of a user-defined Special Atom should always have at least one lower-case letter in the string to distinguish it from *the* Special Atoms defined by nmake.

Types of Special Atoms

There are nine types of Special Atoms in nmake. They may be categorized by their position in the makefile as follows:

- Special Atoms in the Target Position
  - Dynamic List Atoms
  - Immediate Rule Atoms
  - Pattern Association Atoms
  - Sequence Atoms
- Special Atoms in the Prerequisite Position
  - Action Rule Atoms
  - Assertion Attribute Atoms
  - Dynamic Attribute Atoms
- Special Atoms in the Action Block
  - Readonly Attribute Atoms
  - Readonly List Atoms

A few of the Special Atoms (e.g., .MAKE, etc.) may belong to more than one of these types and so may appear in either the target or prerequisite positions. But, as would be expected, they behave differently depending on the position in which they appear.

For example, in the followng assertion,

```
.MAKE : x
```

.MAKE is an Immediate Rule Atom, and indicates that the prerequisite, *x,* is to be made as soon as nmake reads the assertion, while in the following assertion,

```
x : .MAKE
```

.MAKE is a Dynamic Attribute Atom, and indicates that the action block associated with the target, *x*, should be interpreted as containing nmake statements and not shell commands.

The following sections, organized acording to the position of the Special Atom in the makefile, describe a few of the most commonly used Special Atoms and give examples of their use.

Special Atoms in the Target Position

The Special Atoms discussed in this section are specified on the left-hand side of assertion operators, as if they were targets. They include:

- Immediate Rule Atoms

  An Immediate Rule Atom causes nmake to perform an action as soon as it reads the assertion.

- Dynamic List Atoms

  A Dynamic List Atom is used to control nmake actions on other targets. The atom applies to all prerequisites of the specific assertion.

- Sequence Atoms

  A Sequence Atom is made only at specific times during nmake execution.

Saving Prerequisite Values in the Statefile

The .RETAIN Special Atom causes the prerequisites of an assertion to be interpreted as variable names whose values are to be retained in the statefile. The time-stamps are not saved.

In the following example, the .RETAIN Special Atom forces COUNT to be saved in the statefile. Therefore, each time this makefile is executed, the count is incremented by 1. Without the .RETAIN atom, COUNT would default to 0 at each nmake invocation.

Contents of Makefile

```
let COUNT = 0
.RETAIN : COUNT
target : .MAKE
let COUNT = COUNT + 1
error 0 $(COUNT)
```

Run nmake

```
$ nmake
```

Output

```
1
```

Run nmake

```
$ nmake
```

Output

```
2
```

Specifying Directories to Search During Binding

The .SOURCE Special Atom is used to specify a list of directories that should be searched to find required files during the binding process. For example, consider the following assertion:

```
.SOURCE.x : src
```

This assertion specifies that the src directory should be searched for files whose name has a suffix of .x. By default, nmake always searches the current directory first.

Controlling the Sequence of Actions

The .MAKEINIT Special Atom, if present, is executed just after the statefile has been loaded. The base rules typically use this Special Atom to initialize nmake. The .INIT Special Atom is executed immediately after .MAKEINIT. It is typically used to initialize the conditions for a user makefile.

Similarly, the .DONE and .MAKEDONE Special Atoms initiate actions after all other processing has been completed. These atoms are used to tie up loose ends before nmake terminates.

Specifying Initial Actions

This example shows a .INIT action that is always made first.

Contents of Makefile

```
target :
        : Making target

.INIT :
        : Making .INIT

.DONE :
        : Making $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ : Making .INIT
+ : Making target
+ : Making .DONE
```

Specifying Concluding Actions

The .DONE Special Atom is made after all other targets have been made and has no effect on the update status of the targets. It is made unconditionally, regardless of the completion code of previous targets. The commands in the action following this atom are always executed in the foreground shell.

This example shows the order in which processing is done as a result of the .DONE Special Atom.

Define the .DONE action

```
.DONE :
        : Processing has finished.
```

Define the target nexttolast

```
nexttolast :
        : Made before .DONE
```

Run nmake

```
$ nmake
```

Output

```
+ : Made before .DONE
+ : Processing has finished.
```

Special Atoms in the Prerequisite Position

The Special Atoms discussed in this section are specified on the right-hand side of assertion operators, as if they were prerequisites. They include:

- Action Rule Atoms

  Action Rule Atoms are used for nmake control. They are usually invoked at decision points in the nmake execution to override default execution.

- Assertion Attribute Atoms

  Assertion Attribute Atoms are provided to control how assertions are compiled by nmake. An assertion with the .INSERT Special Atom (for example, targ : .INSERT y) causes nmake to insert y into the beginning of the prerequisite list for targ.

- Dynamic Attribute Atoms

  Dynamic Attribute Atoms are used to describe characteristics of how an atom is used in the software construction process. For example, an atom with the .DONTCARE Special Atom causes nmake to ignore the atom if it cannot be made. Dynamic attributes are assigned to atoms through the assertion specifications and, once assigned to the atom, remain in effect throughout the software construction process.

Dynamic Attribute Atoms are specified in the same way as prerequisites are in assertions. The A=attribute attribute selection edit operator can be used to test for dynamic attributes. For information about the A attribute selection edit operator see the *Alcatel-Lucent nmake Product Builder Reference Manual*.

Invoking the Interactive Tracer

When nmake builds targets from assertions that have the .QUERY Special Atom, the interactive tracer is invoked, allowing the user to obtain a trace listing of nmake activity. The trace listing is useful in the early, indiscriminate stages of tracing a makefile. The information it provides includes full atom and state information for all the prerequisites.

You can also invoke the interactive tracer by specifying the .QUERY Special Atom on the command line, as follows:

```
$ nmake -n .QUERY
```

or, more commonly,

```
$ nmake -n query
```

The noexec command-line option (-n) causes shell actions to be traced and printed but not executed (see the *Alcatel-Lucent nmake Product Builder Reference Manual* for details). The noexec option also inhibits makefile compilations; the execution status is not saved in the statefile.

When the interactive tracer is invoked, a prompt is displayed (make>), at which time you can enter queries and new assertions. Any assertions that use intermediate rules such as .MAKE : targ can be entered as make targ at the make> prompt. You can also use the error or print nmake statement to print variable values. You continue to be prompted until you enter q followed by a carriage return or press the CTRL and D keys simultaneously.

The .QUERY Special Atom cannot be tested using the A attribute selection edit operator.

Sample Session Using the Interactive Tracer

The following sample session elaborates on the discussion of the interactive tracer. Two files are used in this sample session that include a makefile and a C source file. The contents of both these files are shown below. The C source file contains a candidate implicit state variable that is defined as a state variable in the makefile.

1. Create the Makefile shown below:

```
MAX == 100
tst :: x.c
.DONE : .QUERY
```

The .DONE Special Atom is made after tst has been made and before nmake terminates execution. The interactive tracer is invoked when .DONE is made.

2. Create the C source file (named x.c) shown below:

```
#include <stdio.h>
main()
{
  printf("%d\n",MAX);
}
```

3. Run nmake by entering the following command line:

```
$ nmake
```

4. Output similar to the following will be displayed (the time-stamp component values will be different when you run this example):

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -DMAX=100 -c x.c
+ cc -O -o tst x.o
make>
```

The interactive tracer 's prompt is displayed as:

```
make>
```

5. At the prompt, enter MAX to display the state of the MAX variable.

```
make> MAX
```

Output similar to the following will be displayed:

```
MAX [ free scan ] = 100

(MAX) : [Dec 10 07:25:14 1997] .SCAN.STATE statevar \
    compiled EXISTS
state: 100
```

The output indicates the following:

- The line containing MAX [scan] = 100 indicates that MAX is a candidate state variable and will be scanned by the .SCAN scanning strategy.

The next two lines in the output indicate the following:

- The date and time (last modification time) when the atom was bound (this information is recorded as the time-stamp component value).
- The state value component of MAX is set to 100, which is the current state of MAX.

MAX has the following attributes in its attribute list:

NOTE:
The attribute names that appear in the interactive tracer's output may not be the same case as the attributes that can be specified by the user or assigned by nmake (i.e., some attributes may appear in upper-case and others in lower-case letters) In addition, some of the names that appear may not be prefixed with a dot (.).

- The .SCAN.STATE Special Atom indicates that nmake should search for the MAX candidate state variable when the source files are scanned.
- The statevar attribute indicates that the atom is a state variable.
- The compiled attribute is an internal memory allocation attribute that indicates that this information came from a compiled makefile.
- The EXISTS attribute indicates that the atom was successfully bound by nmake.

6. At the prompt, enter x.c to display the status of the x.c atom.

```
make> x.c
```

Output similar to the following will be displayed:

```
x.c : [Dec 10 07:25:15 1997] .SCAN.c must=1 terminal \
    compiled regular scanned EXISTS

()x.c : [Dec 10 07:19:56 1997] .SCAN.c event=[Dec 10 \
    07:25:15 1997] compiled scanned
     prerequisites: (MAX) /usr/include/stdio.h==stdio.h
```

There are two pieces of information displayed in the output; the first describes the status of the x.c atom, the second the status of the ()x.c state atom.

There are three different times displayed in square brackets in this output; the first is the most recent update time of x.c or its prerequisites, the second the most recent update time of x.c, and the third the *event* time, the time when nmake notices that the current update time is different from the last time.

The attribute list for the x.c atom contains the following attributes:

- The .SCAN.c Special Atom informs nmake to use the .SCAN.c scanning strategy that is defined in the default base rules when looking for implied prerequisites.
- The must=1 attribute is an internal attribute that indicates the number of reasons why this target is out of date (in this case there is 1 reason).
- The terminal attribute indicates that this atom is a terminal node of a directed graph that was constructed by nmake.
- The regular attribute indicates that this atom is bound to a regular file.
- The scanned attribute indicates that this source file has been scanned for implicit prerequisites.

The compiled and EXISTS attributes in the attribute list were described previously, as were all the attributes in the attribute list for the ()x.c atom.

The prerequisite list for the ()x.c state atom contains two prerequisites: (MAX) and /usr/include/stdio.h==stdio.h (the == is tracer syntax and has nothing to do with state variables). MAX is the candidate state variable defined in the makefile and referenced in the x.c source module. The stdio.h header file was included in the x.c source module.

7. At the prompt, enter x.o to display the status of the x.o object module.

```
make> x.o
```

Output similar to the following will be displayed:

```
x.o : [Dec 10 07:25:16 1997] .OBJECT must=5 implicit \
     target compiled regular triggered EXISTS
 prerequisites: x.c (CC) (CCFLAGS)

()x.o : [Dec 10 07:25:22 1997] .OBJECT event=[Dec 10 \
     07:25:22 1997] force built compiled
 prerequisites: x.c (CC) (CCFLAGS)
```

The attribute list for x.o contains the following attributes:

- The .OBJECT Special Atom informs nmake that this atom is an object module.
- The implicit attribute informs nmake that an implicit metarule was used to make this target; specifically, the %.c to %.o metarule defined in the default base rules.

- The target attribute indicates that this atom is a target.
- The triggered attribute indicates that the action for x.o has been placed in a queue for nmake execution.

The remaining attributes in the attribute list have been previously described. These attributes include must, implicit, compiled, regular, and EXISTS.

8. At the prompt, enter tst to display the status of the tst target.

```
make> tst
```

Output similar to the following will be displayed:

```
tst : [Dec 10 07:25:22 1997] must=7 command target \
compiled regular triggered EXISTS
 prerequisites: x.o .COMMAND.o (CCLD) (CCFLAGS) \
(F77FLAGS) (LDFLAGS)

()tst : [Dec 10 07:25:25 1997] event=[Dec 10 07:25:25 \
1997] force built compiled
 prerequisites: x.o .COMMAND.o (CCLD) (CCFLAGS) \
F77FLAGS) (LDFLAGS)
```

The only attribute in the attribute list that has not been described previously is the command attribute. The command attribute indicates that this target atom is an executable.

The prerequisite list for the ()tst state atom contains the object module x.o, the COMMAND.o rule, as well as several state variables. The COMMAND.o rule is defined in the default base rules and is used for linking object modules to create an executable. The state variables include (CC), (CCFLAGS), (F77FLAGS), (LDFLAGS), and (LDLIBRARIES); they simply define the environment that is to be used during the build.

The debug command-line option is similar to the .QUERY Special Atom in that it is used to provide a trace listing of nmake activity. However, the debug command-line option does not allow the user to debug makefiles interactively. A multilevel trace listing can be obtained by using the debug command-line option simply by specifying the debugging level.

The debugging level can be set at the interactive tracer prompt. For example, the following command line sets the debugging trace level to 1 (do not put spaces around the =).

```
make> set debug=1
```

A trace level of 1 traces targets, prerequisites, and triggered actions. See the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information concerning the debug command-line option.

9. Exit the interactive tracer by pressing the CTRL and D keys simultaneously. Output similar to the following will be displayed:

```
make: debug: time(.DONE) = not found [No such file or \
directory]
make: debug: jobs 2 user 1.39s sys 3.55s
make: debug: normal exit [No such process]
```

   This output is related to nmake internals. The last line containing normal exit indicates that nmake finished processing without any errors.

This concludes the sample session with the interactive tracer.

Forcing Processing when a Target Can't Be Built

Normally if nmake cannot make a target, nmake discontinues work on the target's parent and siblings. However, you may want to continue building for the parent and siblings even if the target could not be built. (See the *Alcatel-Lucent nmake Product Builder Reference Manual* for information regarding the related keepgoing command-line option.)

In the following example, two files, a.f and b.f, and the bin directory exist in the current working directory. The defined rule %.p %.m : %.f does not always generate files with a .m suffix from files with a .f suffix. In this example, the rule generates only the file b.p from the file b.f. The file b.m is not created. The .DONTCARE Special Atom prevents nmake from insisting on having b.m be part of the prerequisite list of pminstall.

In the example, nmake is run twice, with and without the following statement:

```
.ATTRIBUTE.%.m : .DONTCARE
```

Contents of Makefile

```
PFILES = a.p b.p
MFILES = $(PFILES:B:S=.m)

pminstall : $(PFILES) $(MFILES)
        cp $(>) bin
.ATTRIBUTE.%.m : .DONTCARE

%.p %.m : %.f
        cp $(>) $(%).p
        if [[ $(%) = a ]]
        then
                cp $(>) $(%).m
        fi
```

Run nmake

```
$ nmake
```

Output

```
+ cp a.f a.p
+ [[ a == a ]]
+ cp a.f a.m
+ cp b.f b.p
+ [[ b == a ]]
+ cp a.p b.p a.m bin
```

Comment Out Line

```
 /* .ATTRIBUTE.%.m : .DONTCARE */
```

Clean Up

```
$ nmake clobber
+ ignore rm -f b.p a.m a.p Makefile.mo \
Makefile.ms
```

Run nmake again

```
$ nmake
```

Output

```
+ cp a.f a.p
+ [[ a == a ]]
+ cp a.f a.m
+ cp b.f b.p
+ [[ b == a ]]
+ cp a.p b.p a.m b.m bin
cp: b.m: No such file or directory
make: *** exit code 1 making pminstall
```

Using Actions in More Than One Assertion

The .USE Special Atom allows the user to specify a series of actions that can then be used in any number of assertions. The following example demonstrates its use.

Contents of Makefile

```
FILES = a.c b.c c.c

program : prog1 prog2
          : make complete
          ln $(>) $(HOME)/lib

prog1 : link main.c $(FILES)

prog2 : link x.c

link : .USE
          : prerequisites = $(>),targets = $(<)
          /* Variables will be expanded in \action blocks */
          $(CC) $(CCFLAGS) -c $(>)
```

```
                $(AR) $(ARFLAGS) $(<) $(>:B:S=.o)
```

Run nmake

```
$ nmake
```

Output

```
+ : prerequisites = main.c a.c b.c c.c,\
targets = prog1
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib\
        /probe/C/pp/B49DF4E0.bincc -I- -c main.c a.c b.c c.c
main.c:
a.c:
b.c:
c.c:
+ ar r prog1 main.o a.o b.o c.o
ar: creating prog1
+ : prerequisites = x.c,targets = prog2
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/\
        probe/C/pp/B49DF4E0.bincc -I- -c x.c
+ ar r prog2 x.o
ar: creating prog2
+ : make complete
+ ln prog1 prog2 /home/u1/user/lib
```

Special Atoms in the Action Block

The current state of an atom is described by its *Readonly Attributes*, which remain in effect for various lengths of time. An atom's Readonly Attributes are changed by nmake as the state of that atom changes. For example, a target atom whose action block failed is assigned the .FAILED Special Atom only after nmake exits the action with a nonzero error code. You can test for readonly atoms by using the A attribute selection edit operator.

Two kinds of Readonly Attributes are maintained by nmake: Readonly List Atoms and Readonly Attribute Atoms.

Listing Global Makefiles

In this example, making the target causes the list in .GLOBALFILES to be printed.

Define the target

```
target :
        : $(*.GLOBALFILES)
```

Run nmake

```
$ nmake -g glob1.mo -g glob2.mo -g   glob3.mo
```

Output

```
+ : glob1.mo glob2.mo glob3.mo
```

Listing Atoms Whose Actions Have Been Triggered

The .TRIGGERED Special Atom marks atoms whose actions have been triggered during the current nmake execution. An action that has been triggered has been queued for nmake execution but is not necessarily executing yet.

This example shows how the triggered attribute is assigned to targets whose actions have begun.

Initialize the variable

```
TARGETS = a b c d e
```

Define the targets

```
a : b
        touch $(<)

b : c
        touch $(<)

c :
        touch $(<)

d : e
        touch $(<)
e :
        touch $(<)
```

Define the action for .DONE to list all triggered targets

```
.DONE :
        : $(TARGETS:A=.TRIGGERED)
```

Run nmake

```
$ nmake
```

Output

```
+ touch c
+ touch b
+ touch a
+ : a b c
```

Listing State Atoms that Have Been Built

The .BUILT readonly Special Atom marks state atoms that were built in the previous or current execution of nmake. A state atom holds the state of an atom at the time nmake last built it. It is updated each time an atom is built.

Consider the following example where the state rule atoms, ()a.o and ()b.o, are tested to see whether they have been built.

Assign the state rule atoms to variables

```
VAR = ()a.o
VAR1 = ()b.o
```

Define the targets all, a, b, and target

```
all : a target
a :: a.c
b :: b.c
target :
            : $("$(VAR)":A=.BUILT:Q)
            : $("$(VAR1)":A=.BUILT:Q)
```

Run nmake

```
$ nmake all
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/\
        lib/probe/C/pp/B49DF4E0.bincc -I- -c a.c
+ cc -O -o a a.o
+ : ()a.o
+ :
```

The a.o object file was built during execution; therefore, the value of variable VAR has the .BUILT Special Atom and is displayed in the output. The b.o object file was never built; therefore, the value of the variable VAR1 does not have the .BUILT Special Atom and is not displayed in the output.

Defining Your Own Special Atoms

You may define your own Special Atoms to describe atoms based on their names, locations, or last time of modification. To do so, you must write an assertion in which the name of the new Special Atom appears as the target and the .ATTRIBUTE Special Atom apopears as the prerequisite. A Special Atom defined in this way is known as a named attribute, and nmake supports up to 32 named attributes for each invocation of nmake. Thus, the collection of all named attributes from the base rules, the global makefile, and the user makefile cannot exceed 32.

For example, the following assertion:

```
.SCCS : .ATTRIBUTE
```

defines .SCCS as a Special Atom or named attribute.

nmake associates a named attribute with an atom by declaring the named attribute as a prerequisite for the atom. For example, an SCCS file named s.main.c is assigned the named attribute .SCCS by the assertion:

```
s.main.c : .SCCS
```

nmake recognizes the prerequisite as a Special Atom or named attribute and completes this assertion by giving the target this attribute.

Explicitly associating target atoms with named attributes would be tedious and redundant if the list of targets were lengthy. Therefore, nmake allows the association to be made using file name patterns. A named attribute can be specified for all atoms matching a given pattern with the association being made during the binding process. Using the named attribute we have just defined, the assertion:

```
.ATTRIBUTE.s.%.c : .SCCS
```

instructs nmake to associate the .SCCS named attribute with all atoms whose names match s.%.c. The % is the stem and follows the metarule name association semantics *<prefix><base><suffix>*, where % matches the *base* name.

Assigning a Special Atom to a State Variable

To assign a Special Atom or named attribute to a state variable, the user must write an assertion in which the state variable appears as a target and the named attribute appears as the prerequisite. Consider the following makefile, in which we assign the .MYSTATE Special Atom to the state variables USER_STATE_VARIABLES and LOCAL_STATE_VARIABLES.

```
/* Declare .MYSTATE as a named attribute. It
 * will be used as an attribute for state
 * variables declared in this makefile.
 */
.MYSTATE : .ATTRIBUTE
/* Declare the following variables as candidate
 * state variables.
 */
LOCAL_STATE_VARIABLES ==
USER_STATE_VARIABLES ==
/* This assertion assigns .MYSTATE as an
 * attribute for the state variables in this
 * makefile.
 */
(LOCAL_STATE_VARIABLES) : .MYSTATE
(USER_STATE_VARIABLES) : .MYSTATE
```

Testing an Atom for Named Attributes

Atoms are frequently assigned named attributes for the purpose of grouping together atoms that have the same attribute. Using the N shell file match edit operator, nmake can select tokens from a variable that match the specified pattern. The use of the N edit operator is best shown through the following example. (See the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information concerning the N edit operator.)

Consider the makefile specifications for building a program procr from the source modules main.c, input.c, lexical.l, and yacc.y. Further suppose that the source modules needed a named attribute specifying the type of source, such as .YACC_src or .C_src. The makefile would be given as:

```
/*
 * Create the named attributes .C_src .YACC_src
 * .LEX_src and .LOCAL_src., remembering that
 * multiple targets of a single assertion are
 * treated as if each target - prerequisite
 * appeared on a line by itself.
 */
.C_src .YACC_src .LEX_src .LOCAL_source : .ATTRIBUTE
/*
 * Assign to the variable SOURCE_FILES the
 * complete list of all source modules needed.
 */
SOURCE_FILES = main.c input.c lexical.l yacc.y
/*
 * Use an assertion to assign the named attribute
 * .LOCAL_source to all the source files.
 */
$(SOURCE_FILES) : .LOCAL_source
/*
 * Use the :N=<pattern> in-line edit operator to
 * select all tokens of $(SOURCE_FILES) which
 * match the specified pattern. This will enable
 * us to choose appropriate source modules to have
 * the attribute .C_src. In this example,
 * $(SOURCE_FILES:N=*.c) evaluates to main.c and
 * input.c, which are multiple targets with the
 * .C_src attribute as a prerequisite.
 */
$(SOURCE_FILES:N=*.c) : .C_src
/*
 * Now do the same thing using the pattern *.y to
 * select all yacc source modules.
 */
$(SOURCE_FILES:N=*.y) : .YACC_src
/*
 * Now select all tokens matching the pattern
 * *.l which in our example evaluates to
 * lexical.l.
 */
$(SOURCE_FILES:N=*.l) : .LEX_src
```

Occasionally it may be necessary to unassign a named attribute that has previously been assigned to a target. This may be done by placing a minus sign before the named attribute prerequisite.

For example, in the example above, the assertion

```
$(SOURCE_FILES) : .LOCAL_source
```

assigned the named attribute .LOCAL_source to all the source modules. But suppose that we wanted any yacc files in our list of source modules to have only the named attribute .YACC_src. We would then have to change the assertion

```
$(SOURCE_FILES:N=*.y) : .YACC_src
```

to

```
$(SOURCE_FILES:N=*.y) : .YACC_src -LOCAL_source
```

The prerequisite .YACC_src is equivalent to +YACC_src; both have the effect of assigning the named attribute to the target. The prerequisite –LOCAL_source unassigns the LOCAL_source named attribute from the target atoms, if they exist.

Contents

Contents

Using Assertion Operators

6

Assertion operators define dependency relationships between targets and prerequisites. The dependency assertion operator (:) is the only built-in assertion operator that is defined in the nmake engine. Other assertion operators are defined in the default base rules and have the following syntax:

:*identifier*:

where *identifier* can be any string of characters (A–Z, upper- or lower-case). The identifier can also be null, as it is in the source dependency assertion operator (::). You can also define your own assertion operators in a makefile. The *Alcatel-Lucent nmake Product Builder Reference Manual* contains manual pages for all the predefined nmake assertion operators.

Of the predefined assertion operators, this chapter discusses only the dependency assertion operator (:), which defines the dependency assertion, and the source dependency assertion operator (::), which defines the source dependency assertion. The difference between the dependency and source dependency assertion operators is:

- The dependency assertion operator is the nmake primitive assertion operator. Dependency assertions are normally used when the means of building a target are explicitly provided in the action block. Without an action block, the dependency assertion operator only sets up a dependency relationship between the target list and the prerequisite list.
- The source dependency assertion operator is predefined in the nmake default base rules. Source dependency assertions can use source files such as C files and troff files as prerequisites. Many predefined rules in nmake's default base rules are applied; however, explicit actions can also be specified for the target.

The advantage of using the source dependency assertion operator is that the information on how to build a target is already predefined and does not have to be specified in the makefile. This property results in concise makefiles.

The Dependency Assertion

The dependency assertion is used when the action must be explicitly specified. This type of assertion is also used to establish dependency relationships.

The dependency assertion has the following syntax:

target **:** *prerequisites*
    [*action block shell script* **]**

The examples that follow show different ways of building the same target. Subsequent examples are simpler, more efficient and less error prone.

The following makefile is used to build and maintain the convert target. The makefile also maintains the directory where convert is built.

NOTE:
In this example, all files excluding stdio.h and libld.a are assumed to be in the same directory as the makefile.

```
convert : init.c main.c defines.h\
        /usr/include/stdio.h /usr/lib/libld.a
        cc -o convert init.c main.c /usr/lib/libld.a
my_clean :
      rm -f init.o main.o
my_clobber : my_clean
      rm -f convert
```

The first rule specifies that convert's prerequisites are the source files init.c and main.c, the header files included by init.c and main.c, and the ld library. Once nmake has built convert, it will build it again only if one or more of these prerequisites are modified.

The my_clean target is used to remove the intermediate object files, init.o and main.o. The my_clobber target is used to remove the intermediate object files and then also remove the product convert. Note that the my_clobber target depends on the my_clean target. Because prerequisites must be brought up to date before a target can be brought up to date, first the object files init.o and main.o will be removed by the my_clean assertion; then convert will be removed.

Another possible makefile for maintaining convert is as follows:

```
convert : init.o main.o /usr/lib/libld.a
        cc -o convert init.o main.o /usr/lib/libld.a
init.o : init.c defines.h
        cc -c init.c
main.o : main.c defines.h /usr/include/stdio.h
        cc -c main.c
my_clean :
        rm -f init.o main.o
my_clobber : my_clean
        rm -f convert
```

Although longer, this makefile is more efficient than the previous makefile, because the product convert is now dependent on intermediate object files and the ld library, rather than on the source files. If only init.c is modified, only init.o is rebuilt and then linked with the existing, up-to-date object main.o and the ld library.

NOTE:
Whenever possible, makefiles should be written so that compilations of source files are kept distinct from one another. This method minimizes the amount of compilation being done each time the product is built.

The previous example can be considerably simplified without losing any of the increased efficiency by allowing nmake to infer header file dependency information. Using nmake would also be more reliable because you might change the #include lines in your source file, which would make the corresponding assertion invalid.

nmake knows by means of the predefined scan rules that header files can be included in C language source files. These header files are therefore prerequisites of the C language source file. Files that have a .h suffix can be included in the source file using C language #include directives. nmake scans all prerequisite files having a .c suffix for #include lines. nmake then searches for the included file and infers that whatever target has the source file as a prerequisite also has the header file as an implicit prerequisite. This concept also applies to nested header files. See Chapter 8, *Defining Custom Scan Rules*, for more information regarding Scan Rules.

By allowing nmake to infer header file dependency information, the previous makefile can be shortened as follows:

```
convert : init.o main.o /usr/lib/libld.a
        cc -o convert init.o main.o /usr/lib/libld.a
init.o : init.c
        cc -c init.c
main.o : main.c
        cc -c main.c
my_clean :
    rm -f init.o main.o
my_clobber : my_clean
    rm -f convert
```

nmake also knows by means of its default base rules how to make a .o file from a .c file. If init.o is needed and init.c exists and you have not specified how to make init.o, nmake infers a dependency rule for building init.o.

By allowing nmake to infer how .o files are built, the makefile can be further simplified as follows:

```
convert : init.o main.o /usr/lib/libld.a
    cc -o convert init.o main.o /usr/lib/libld.a
my_clean :
    rm -f init.o main.o
my_clobber : my_clean
    rm -f convert
```

The Source Dependency Assertion

The source dependency assertion automatically provides implicit actions commonly used in makefiles, thereby saving you from having to write them. The source dependency assertion has the same syntax as the dependency assertion.

The syntax is shown below:

target **::** *prerequisites*

There is no update action block. nmake applies the predefined rules when processing a source dependency assertion, eliminating the need to state the action explicitly. The makefile discussed previously can be shortened considerably by using a source dependency assertion and relying on the clean and clobber common actions that are also predefined in the default base rules.

The previous makefile could be written using the source dependency assertion operator as:

```
convert :: init.c main.c libld.a
```

(The my_clean and my_clobber targets are handled by the nmake common actions clean and clobber, respectively.) The source dependency assertion causes nmake to use file suffixes to determine different types of source files used to build the target. In addition, this assertion can also recognize option flags in the prerequisite list.

Consider the following assertion (which assumes that all of the prerequisites and the makefile are in the same directory):

```
foo :: foo.c foobar.y foo.1 libab.a
```

In this assertion, foo is the target to be made. Note that different file suffixes appear in the prerequisite list.

This makefile can be written in a different way by using the -l option as shown below:

```
foo :: foo.c foobar.y foo.1 -lab
```

In this example, the -l option links in the shared library version of the libab.a library if it exists; otherwise, libab.a gets linked in statically.

The table below summarizes the actions nmake takes for the various file suffixes..

File Suffix	Action
foo.c	nmake knows that it must compile files that have a .c suffix. Once the file is compiled, an object file is produced that has the same file basename with a .o suffix. In addition, suppose that foo.c includes a header file called foo.h. If foo.h is ever modified, nmake knows that foo.h is an implied prerequisite and therefore will recompile foo.c even though foo.h is not explicitly listed in the prerequisite list.
foobar.y	When nmake processes a file that has a .y suffix, it knows to process the file with yacc. After the file has been processed with yacc, a file called y.tab.c is produced. nmake knows that it must rename this file (in this case, the file is renamed to foobar.c).
foo.1	nmake treats a file with a .1 suffix as a manual page; therefore, nmake recognizes that foo.1 is a manual page and does not use the file when building the foo target.
libab.a	nmake treats a file that has a .a suffix as a library; therefore, nmake knows to link libab.a statically at the end of compilation.

Defining Your Own Assertion Operators

nmake allows you to define your own assertion operators. This capability is particularly useful if you want to design global rules files that can be applied to multiple makefiles.

The syntax for defining an assertion operator is:

```
":operator_name:" : .OPERATOR
        defined_actions
```

where *operator_name* can be any identifier name that you choose. The *operator_name* must be enclosed by quotes.

Once an assertion operator is defined, whenever you have an assertion of the form:

```
target_list :operator_name: prerequisite_list
        specific_actions (optional)
```

the *defined_actions* are used. If the optional *specific_actions* are given they are used only if referenced in *defined_actions* by the $(@) variable.

When defining your assertion operator, you should use the automatic variables to control the actions. The automatic variables most useful in this context are:

$(<)	Use this automatic variable in the *defined_actions* definition; when the assertion is acted on, it is interpreted as the *target_list* of the assertion.
$(>)	In the same way, this automatic variable is interpreted as the *prerequisite_list* of the assertion.
$(@)	When the assertion is processed, this automatic variable refers to the *actions* specified in the action block of the *target_list:operator_name: prerequisite_list* assertion.

In the following example, we assume that the variable TRANSMIT has been defined to be your organization's command string to send files to another machine.

```
":GETFILES:" : .OPERATOR
            $(@)       /*take specific_actions first*/
            $(TRANSMIT) $(>) $(<)
```

You can then use this definition in an assertion with no specific actions similar to the following:

```
machine1 :GETFILES:  $(FILE_LIST)
```

This is a very oversimplified example.

Here is another example, extracted from the default base rules file, Makerules.mk. It consists of two pieces: the first defines a .USE update action block that compares the lhs file with the rhs file and copies the rhs file to the lhs file if they are different. The next part defines :COPY: as an assertion operator that contains one assertion line as its action block. This assertion uses the actions defined in .DO.COPY.

```
.DO.COPY : .USE
        $(CMP) -s $(*:O=1) $(<) || $(CP) $(*:O=1) $(<)

/*
* lhs is a copy of rhs
*/
":COPY:" : .MAKE .OPERATOR
        $(<:V) : .SPECIAL .DO.COPY $(>:V)
```

Contents

Contents

Programming in Alcatel-Lucent nmake

7

nmake provides a large degree of programmatic control over the entire makefile. This control includes the ability to define both string and integer variables and to specify overall flow-control statements.

As we have seen, makefile assertions may contain action blocks; the action blocks may contain either nmake programming statements or shell statements. (nmake also allows nmake programming statements outside of action blocks.) The following sections describe these two kinds of statements.

Shell Statements

This section describes the shell flow-control statements that may be used in action blocks.

Special Shell Scripts

The Alcatel-Lucent nmake package includes two special shell scripts that implement an ignore statement and a silent statement. The ignore statement causes nmake to ignore the exit code of the following command. For example,

```
ignore mkdir xyz
```

returns a zero exit code (indicating success), even if directory xyz exists and the mkdir command fails. The silent statement causes the following command not to be echoed. For example,

```
silent mkdir xyz
```

suppresses output of the + mkdir xyz line.

Shell Constructs

You can use shell constructs directly within any action block. These include if/fi constructs, case/esac, and for/do/done constructs. For example,

```
installmac : psmacs semacs ymacs
        if [ ! -d "$(ROOT)/lib" ]
        then
                /bin/mkdir $(ROOT)/lib
        fi
        if [ ! -d "$(ROOT)/lib/tmac" ]
        then
                /bin/mkdir $(ROOT)/lib/tmac
        fi
        chmod 664 $(*)
        /bin/mv ymacs $(ROOT)/lib/tmac/xmacs
        /bin/mv psmacs semacs $(ROOT)/lib/tmac
```

The action block moves the file ymacs into one directory and then changes its name to xmacs and then moves the files psmacs and semacs into the directory tmac. However, before any movement takes place, a test is made to see whether the directories exist and, if they do not, the directories are created.

Shell Exit Codes

Use the false command to force a shell action to exit with an error. When the exit shell command is used directly in an action block the return code is lost and nmake does not see an error when the code is greater than 0.

CAUTION:
Use false, not exit, to exit a shell action with an error.

Error codes from shell actions are reported in the output. The following example indicates that the shell action exited with an error code of 23 while making target main.o. Code 23 is not an nmake error code, it is the error code from the tool running in the shell action.

```
make: *** exit code 23 making main.o
```

nmake Programming Statements

nmake programming statements include flow control, variable assignment, and expressions using arithmetic and logic operators. nmake programming statements are interpreted by nmake, not the shell. As a consequence, when they appear in action blocks, the .MAKE Special Atom must be included in the prerequisite list. The .MAKE Special Atom ensures that the nmake programming statements are interpreted, rather than passed to the shell. For example:

```
myinit : .MAKE
        FILES = a.c b.c
```

The action block in this example is an nmake programming statement that assigns a.c and b.c to the nmake variable FILES; FILES is not a shell variable because .MAKE is specified. (See the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information regarding the .MAKE Special Atom.)

nmake programming statements or keywords must be the first word on a line. Any user-defined names matching any of the keywords must be quoted to avoid conflicts when used as the first word on a line. For example, suppose a target has been named if, which is an nmake keyword, as in the following example:

```
if : x
        shell action
```

This will cause an error because the target if will be interpreted as an operator; to work properly, the example would have to be changed as follows.

```
"if" : x
        shell action
```

Expressions

The programming statements refer to the use of *expressions*. Each expression can be a combination of arithmetic and string expressions where the string expression components must be quoted, i.e., "*string*".

Pattern Matching in Expressions

Strings enclosed with double quotes use shell pattern matching scheme (for example, "$(VAR)" == "*pattern*"). Empty strings evaluate to 0 in arithmetic expressions and non-empty strings evaluate to nonzero. Expression components can also contain optional variable assignments. All computations are done using signed long integers. For example:

```
if "$(VAR)" == "string"
        action
end

if v=2 < 4
        action
end
```

The string comparison requires both items on either side of the relational operator (e.g., ==) to be enclosed in double quotations. However, when one of the items is a string and the other is a variable in its unexpanded form (say, VAR == "*string*"), nmake applies an implicit string conversion rule on VAR to enable string comparison. So, for the expression, VAR == "*string*" nmake essentially converts to "$(VAR)" == "*string*" for the comparison.

The integer variable v is both assigned and compared on a single line (see *Integer Variables -- The let Statement*, below).

NOTE: The general rule for arithmetic and string expressions is;

```
if var
        true /* where var is non-null and !=0*/
else
        false /* where var=0 or var is null */
end

if "$(var)"
        true /* where var is non-null */
else
        false /* where var is null */
end
```

Integer Variables -- The *let* Statement

In addition to the string-variable facility we have seen in Chapter 3, *Using Variables*, , nmake also supports integer variables. The let statement is used for evaluating integer expressions. The syntax is as follows:

let *variable=expression*

An integer variable can be assigned the results of any integer expression by preceding the assignment with the let keyword. A variable assigned a new value without the let keyword is assumed to be of type string. An integer variable can be used within an integer expression as in the following:

```
let index = 5
let count = index * 5
```

which assigns the integer value 5 to the integer variable index, and assigns the evaluation of index * 5, or 25, to the integer variable count. The integer variable name is used in expressions, such as index * 5; no $ is used. When complex expressions are to be used, parenthesize according to standard mathematical evaluations.

Scope of Variables -- The *local* Statement

String and integer variables can have both local and global (default) scope. Local scope is restricted to the current assertion; global scope refers to all assertions. The syntax of the *local* statement is as follows:

local *var1 var2 ...*
local *name=value*

Consider the following example. On the first line in the makefile, a global integer variable called var is defined. The value of the global variable is 5. In the action block for the targ1 target, a local variable called var is declared. The value of the local variable is 100.

Once the makefile is processed, the output verifies that in fact, two different variables exist.

Contents of Makefile

```
let var = 5   /* Global Variable */

all : targ1 targ2

targ1 : .MAKE
        local var     /* Local Variable */
        let var = 100
        print Local Variable is $(var)

targ2:
        : Global Variable is $(var)
```

Run nmake

```
$ nmake
```

Output

```
Local Variable is 100
+ : Global Variable is 5
```

In a .MAKE action block, only nmake statements are valid. The print statement must be used to echo a string (see *The print Statment*, below).

NOTE:
There are also other forms of the *local* statement which when used in a function with the automatic variable $(#), causes $(#) to be expanded to the actual number of arguments in the function call. The syntax of these forms is:

local -*[n] arg* ...
local (*formal_args* ...) *actual_args*

The following example will return the number of actual arguments, 3 and 2:

```
F1 : .FUNCTION
        local (a b c) $(%)
        print $(#)

F2 : .FUNCTION
        local - a b
```

```
                print $(#)

        t :
                $(F1 1 2 3)
                $(F2 1 2)
```

Arithmetic, Logic, and Bitwise Operators

The arithmetic operators supported by nmake are listed below.

Operator	Description
!	Logical not
!=	Not equal
%	Modulo
&	Bitwise AND
&&	Logical AND
*	Multiplication
+	Addition
-	Subtraction
/	Division
<	Less than
<<	Bit shift left
<=	Less than or equal to
==	Logical equal
>	Greater than
>=	Greater than or equal to
>>	Bit shift right
?:	C-style operator (see below)
^	Bitwise exclusive OR
|	Bitwise inclusive OR
||	Logical OR
~	Bitwise one’s compliment

The ?: C-Style Operator

The ?: C-style operator can be used in integer expressions. The syntax is:

expression ? *true_expression* : *false_expression* such that if *expression* is true, then *true_expression* is evaluated. If *expression* is false, *false_expression* is evaluated.

For example, suppose a makefile contains:

```
let counter = 1
let index = (counter > 0 )? 1 : 0
```

nmake evaluates counter to the integer value 1, and index to the integer value 1, since the expression counter > 0 evaluates to true.

The if/else Statement

The if/else statement is an nmake construction and uses a syntactical form that is similar to shell if/else statements.

The syntax is:

```
if expression_1
        statement_1
elif expression_2
        statement_2
else
        statement_3
end
```

where *statement_n* is an nmake statement or statement block.

Consider the following example. We will be composing a book from two chapters, each consisting of one or more sections, where the sections are dependent on tables and pictures. Note that the if/ end block occurs before the composition rules. Listed below is the output that was created when DFLAGS was set equal to post on the command line.

Contents of Makefile

```
CHAP1 = sect1a.tr
CHAP2 = sect2a.tr sect2b.tr

if "$(DFLAGS)" == "post"
        POST = dpost
elif "$(DFLAGS)" == "i300"
        POST = dimpress
elif "$(DFLAGS)" == "aps"
        POST = daps
end

%.tr : %.pr (DFLAGS) (POST)
        pic -T$(DFLAGS) $(>) | tbl > $(<)

book : $(CHAP1) $(CHAP2)
```

```
            cat $(*) > $(<)
            troff -T$(DFLAGS) macros $(<) | $(POST) | \
                    lp -d$(DFLAGS)
```

Run nmake

```
$ nmake DFLAGS=post
```

Output

```
+ pic -Tpost sect1a.pr
+ tbl
+ pic -Tpost sect2a.pr
+ tbl
+ pic -Tpost sect2b.pr
+ tbl
+ cat sect1a.tr sect2a.tr sect2b.tr
+ troff -Tpost macros book
+ dpost
+ lp -dpost
request id is post -9960  (standard input)
```

Listed below is the output that was created when DFLAGS was set equal to i300 on the command line.

Run nmake again

```
$ nmake DFLAGS=i300
```

Output

```
+ pic -Ti300 sect1a.pr
+ tbl
+ pic -Ti300 sect2a.pr
+ tbl
+ pic -Ti300 sect2b.pr
+ tbl
+ cat sect1a.tr sect2a.tr sect2b.tr
+ troff -Ti300 macros book
+ dimpress
+ lp -di300
request id is imagen -9960 (standard input)
```

The for Statement

The for statement is used to define iterative actions. The syntax is as follows:

for *variable patterns*
        statement
end

The statements between for and end are executed with *variable* assigned to the name of each atom matching one of the shell *patterns*.

The following example illustrates the use of the for statement in conjunction with the use of local nmake integer variables. The .INIT Special Atom ensures that the action block is executed before making any targets.

Define .INIT
```
.INIT : for-ex
```

Define for-ex
```
for-ex : .MAKE
```

Define FACT and TOTAL as integer variables
```
        let FACT = 1
        let TOTAL = 0
```

Define for loop
```
        for COUNT 1 2 3
                let FACT = $(FACT) * $(COUNT)
                WHAT += $(FACT)
                let TOTAL = TOTAL + FACT
        end
```

Define target
```
target :
        : $(WHAT)
        : $(TOTAL)
```

Run nmake
```
$ nmake target
```

Output
```
+ : 1 2 6
+ : 9
```

WHAT is not defined as an integer variable; consequently, it is treated as a string variable. Thus, when the += operator is used, the string values 1, 2, and 6 are concatenated to become the value of WHAT. The += operator is not an arithmetic operator in nmake, so the running value of TOTAL must be computed by the last let statement. Both variables must be printed as string values.

The while Statement

The while statement controls conditional iteration. Its syntax is:

```
while expression
        statement 1
                .
                .
                .
        statement n
end
```

Consider the following example in which the while programming statement is used.

Contents of Makefile

```
let INCR = 1
let BOUND = 1
while BOUND <= 3
        let BOUND = BOUND + INCR
        PREQ += prog$(BOUND).c
end

prog :: $(PREQ)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/\
lib/probe/C/pp/B49DF4E0.bincc -I- -c prog2.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/\
lib/probe/C/pp/B49DF4E0.bincc -I- -c prog3.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/\
lib/probe/C/pp/B49DF4E0.bincc -I- -c prog4.c
+ cc -O -o prog prog2.o prog3.o prog4.o
```

Again, even though BOUND is an arithmetic variable, it must be referenced as a string variable for non-let statements.

The break Statement

A break statement may be used to break out of the **for** and **while** loops and resume execution after the **end** statement.

The error Statement

The syntax is:

error *level* "*message*"

where *message* is output as an nmake error message with severity level *level*, and *level* may be any one of the levels shown below.

Level	**Description**
< 0	Debug trace—output only if the absolute value of *level* is <= debug
0	Information

Level	Description
1	Warning
2	Error—no exit
≥ 3	Error—exit with code (*level-2*)

The print Statement

The syntax is:

print **–n –u[***0-9***] –f** *format* **[+–]o** *file* **- -** *data*

where *data* is an output string. The optional arguments are as follows:

Argument	Description
-n	adds a trailing null character at the end of the output string
-u[0-9]	sends output to the specified file descriptor (e.g., **0** is stdin, **1** is stdout, **2** is stderr)
-f *format*	specifies the output *format*, where *format* is a C printf-like format string
-o *file*	open *file* for writing
+o *file*	open *file* for appending
- -	ends the arguments, anything after **- -** is considered *data*
data	the output string data

The eval/end Statement

The syntax is:

```
eval
...
end
```

The variables between eval and end are expanded an additional time. eval/end pairs can nest to cause more than one additional expansion.

This example uses the V edit operator to prohibit expansion of variables inside STD1, STD2 and STD3 so we can see how the eval/end paris affect the variables.

Contents of Makefile

```
ABC = abc
XYZ = xyz

eval
STD1 = $(ABC) $$$(XYZ)
```

```
eval
STD2 = $(ABC) $$$(XYZ)
eval
STD3 = $(ABC) $$$(XYZ)
end
end
end

t : .MAKE
        print =1= $(STD1:V)
        print =2= $(STD2:V)
        print =3= $(STD3:V)
```

Run nmake

```
$ nmake
```

Output

```
=1= abc $$(XYZ)
=2= abc $(XYZ)
=3= abc xyz
```

The include Statement

The syntax is:

include **[–]** *path*

This statement Includes the file specified by *path*. If – is present, the file is optionally included, that is, no warning is generated if the file is not found.

NOTE:
This statement should be used instead of the #include cpp directive in a makefile for faster makefile compilation. Although nmake recognizes cpp(1) directives in a makefile, nmake programming statements should be used instead for faster compilation.

The read Statement

The syntax is:

read -i *file variable*

This statement reads input from the file, *file* and assigns the contents of the file to *variable*.

The return Statement

The syntax is:

return [*expression*]

This statement forces a return from the action with results specified by *expression*, where *expression* can be any of those listed below.

expression	**Description**
omitted:	If *expression* is omitted, return as if the action completed normally.
-1	The action failed.
0	The target exists but has not been updated.
>0	Set the target time to the value specified by expression.

The rules Statement

The syntax is:

rules ["*base-rules*"]

This statement determines the base rules context. If the rules statement is specified, it must be the first line of the first makefile. If it is of the form:

rules "*base-rules*"

base-rules.mo will be used rather than the default base rules makerules.mo If *base-rules* is not specified, no base rules are used. Omitting the rules statement is equivalent to specifying rules "makerules".

The set Statement

The syntax is:

set [no]*name*=[value]

This statement sets options using the option-by-name format.

Contents

Contents

Defining Custom Scan Rules

8

The Dependency Scanner

A source file specified as an explicit prerequisite in an assertion may have a scan strategy associated with it. A scan strategy comprises:

- a list of scan rules that are used to recognize file and preprocessor-variable references in the source files
- optional search flags
- optional binding directories.

When nmake encounters such a source file, its built-in dependency scanner scans it to determine whether that file contains implicit prerequisites. Any implicit prerequisites that are found are recorded in nmake's statefile, so changes to any of these files automatically trigger re-execution of that assertion.

nmake's dependency scanner is programmable. In fact, even rules for common languages such as C, FORTRAN, etc., are not hard-coded into nmake, but are read from a definitions file (Scanrules.mk) during nmake process initialization. You can take a look at the file *nmake_install_root*/lib/make/Scanrules.mk to see the existing scan rule definitions, where *nmake_install_root* is the directory where all the Alcatel-Lucent nmake files are installed.

If you want to use a programming language that does not appear in the table below, you must extend the predefined scan rules.

Supported Languages

The following table lists the file extension and predefined scan strategy for each of the languages in the default base rules.

Language	File Extension	Scan Strategy
C	.c	.SCAN.c
C++	.C	.SCAN.c
C++	.cc	.SCAN.c
C++	.cpp	.SCAN.c
C++	.cxx	.SCAN.c
IDL	.idl	.SCAN.idl
FORTRAN	.f	.SCAN.f
Macro Files	*	.SCAN.m4
Makefiles	.mk	.SCAN.mk
nroff	*	.SCAN.nroff
Pascal	*	.SCAN.p
RATFOR	.r	.SCAN.r
Shell Scripts	.sh	.SCAN.sh
SQL	*	.SCAN.sql
4GL, ESQL constructs	*	.SCAN.F

* For these files there is no default file extension. So if you want to use these scan strategies, you must specify what file extension you use for these languages in your makefile.

Defining New Scan Strategies

The scan rules are used to specify the syntax of all the file and preprocessor-variable references supported by your source language, to indicate the start and end of comments in your language (so that nmake can ignore them), and to identify the syntax of conditionally included portions of source text (again, so that nmake can ignore them). If your language supports any or all of these concepts and if there is no default scan strategy for your language defined in Scanrules.mk, you must define a set of scan rules so that nmake is able to look for the references in your source files and dynamically generate implicit prerequisites for your source files.

To write your own scan rules, proceed as follows:

1. Define the scan rules themselves in the makefile. For example,

```
.SCAN.x : .SCAN
        I | $ include "%" | /* this is the scan rule */
```

where .SCAN.*x* is a token representing a scan strategy, and subsequent lines define the scan rules.

2. Assert the scan strategy for the intended source files. For example,

   ```
   .ATTRIBUTE.%.pattern : .SCAN.x
   ```

   means that .SCAN.*x* scan strategy is associated with all the files whose names are of the form %.*pattern* (where % is the wildcard character).

3. If your language supports a search-flag concept, define an nmake variable that accounts for the search flags for the included files. For example, for the awk language:

   ```
   AWKFLAGS &= $$(.INCLUDE. awk -I)
   ```

   More generally,

   ```
   xxFLAGS &= $$(.INCLUDE. lang [flag] [-I-])
   ```

   This statement returns the *flag* option list for .SCAN.*lang*. The optional third argument specifies the -I- style flag used by the compile tool. If present, the specified third argument will be used to separate the local include directories from the standard include directories in the option list returned by .INCLUDE.

4. Specify the search order of the source files being scanned using the new scan strategy (binding directories). For example, for some hypothetical language X (.x source files):

   ```
   .SOURCE.%.SCAN.x : .FORCE $$(*.SOURCE.x) $$(*.SOURCE)
   ```

   This assertion means that the included file is found by searching the directories of .SOURCE.x first and then the directories of .SOURCE. If the .FORCE special atom were not used, the bindfile() routine would look through prerequisites of .SOURCE when a file wasn't found anywhere else.

nmake variables can be used within the body of .SCAN.*x*.

Scan Rules

Scan rules are built using a small language of their own. There are presently ten scan rules that fall into four general types. The following table lists each rule by type, syntax and meaning: |

Type	**Rule Syntax**	**Meaning**
Option	**O**|*option*|	Set scan **o**ption
Quote	**Q**|*str1*|*str2*|*esc*|*flags*	Strings indicating start and end of comment/**q**uote
Pattern	**D**|*patt*|	Pattern that **d**efines preprocessor macro
	B|*patt*|	Pattern that starts/**b**egins conditional section
	E|*patt*|	Pattern that **e**nds conditional section
	I|*patt*|*action*|	Pattern that references **i**ncluding/**i**ncluded file
	T|*patt*|*action*|	**T**ype or **t**okenized match
	A|*patt*|*action*|	Pattern that allows **a**rchive (aggregate) type of scanning
Boundary	**S**|*action*|	Execute action before the scanning **s**tarts
	F|*action*|	Execute action after the scanning is **f**inished

- In scan rules, the pipe symbols (|) are literal.
- *option*, *str1*, *str2*, *esc*, and *flags* are described in the sections on the option type and quote type, below.
- The pattern, *patt*, is a character string. An arbitrary sequence of characters in a pattern may be indicated by an asterisk (*). Any character that has to be used literally in *patt* and also has a special meaning for nmake must be escaped with a backslash character (\).
- *action* can be null or of the form *action1*[|. . .| *actionN* |].

Scan Rules - *action*

As shown in the table above, scan actions are only applicable to pattern and boundary type rules. For the pattern type, if *patt* is found, *action* is evaluated. Whereas for the boundary type, *action* is governed by the start(S) and finish(F) rules. A valid scan *action* can be any combination of:

a. **A***attribute-list*

attribute-list is a space-separated list of Special Atoms or user-defined atoms that can be assigned with the .ATTRIBUTE special atom that are assigned to a file being scanned or referenced (see the *Alcatel-Lucent* *nmake Product Builder Reference Manual* for more details about .ATTRIBUTE; see Example 4, below, for use of *attribute-list* and the A (attribute) action).

When used in the pattern and boundary rules, it has the following meaning:

[TA]|*patt*|A*attribute-list*	If *patt* is found, *attribute-list* is assigned as implicit prerequisite to the file being scanned.
I|*patt*|A*attribute-list*	If *patt* is found, *attribute-list* is assigned as implicit prerequisite to the file referenced, i.e., the included file -- NOT the file being scanned. NOTE: The file referenced/included is also an implicit prerequisite to the file being scanned and inherits the latter's scan strategy.
[SF]|A*attribute-list*	*attribute-list* is assigned as implicit prerequisite to the file, prior to it being scanned (the S type) or to the file after it has been scanned (the F type).

b. **R***stmt*

stmt is a valid stand-alone nmake statement that is read and evaluated by the nmake engine (see Example 3, below, for the use of *stmt* and the R (read) action).

[ITA]|*patt*|R*stmt*	If *patt* is found, *stmt* is read and evaluated by the nmake engine. NOTE: For the **I** rule, the file referenced is still the implicit prerequisite of the file scanned and inhere-its the latter's scan strategy.
[SF]|R*stmt*	*stmt* is read and evaluated by the nmake engine before the file is scanned (the S type) or at the completion of scanning the file (the F type).

c. **M***expression*

expression is an nmake expression that is expanded if the pattern is found (see Example 6, below, for the use of expression and the M (map) action).

[ITA]|*patt*|M*expression*	If *patt* is found, the expanded value of *expression* becomes the implicit prerequisite of the file being scanned. NOTE1: *expression* may have multiple tokens, hence multiple implicit prerequisites. NOTE2: For the **I** type, the implicit prerequisite is **NOT** the file referenced as indicated by the **'%'** within the matched pattern, *patt*.
[SF]|M*expression*	The expanded value of *expression* is the implicit prerequisite of the file, prior to it being scanned (the S type) or to the file after it has been scanned (the F type).

d. **OX**

The OX scan language *action* explicitly disables the inheritance of a scan strategy by the implicit prerequisites (see Example 2, below).

I|*patt*|OX	Normally, (with the **I** rule) the file included/referenced becomes the implicit prerequisite of the file scanned and inherits its scan strategy (ie., of the file scanned). However, with the OX action, if *patt* is found, the implicit prerequisite of the file scanned is prevented from inheriting the scan strategy.

Option Scan Rule

The option type has only one rule. The option rule sets a scan option. The following table lists the options available and their meaning.

Option	Meaning
S	Look for candidate state variable during scanning
M	Do an m4 scan (.SCAN.m4)—all other rules except the start rule and finish rule are ignored

The S option allows recognition of state variables whose values are passed on to the language preprocessor.

Example 1: The S Option

Contents of oc.c

```
main() {
#ifdef MAX
      printf("%d\n",MAX);
#endif
}
```

Contents of Makefile

```
.SCAN.c : .SCAN
      O|S|
      .
      .
      .
MAX == 1
os :: os.c
```

In this example, the state variable MAX is declared in the makefile. When the option rule O|S| is used, the output of the program os.c is 1. If the O|S| rule were not used, the nmake dependency scanner would not look for the state variables, which would result in MAX not being defined and in no output for the above program.

Quote Scan Rule

The quote type has only one rule. The quote *rule* indicates regions of quoted source text, i.e., regions of text to ignore, while scanning for file references. This rule indicates comments and strings within the source text. The format is:

Q|*str1*|*str2*|*esc*|*flags*|

where *str1* and *str2* are the character strings that represent the start and end of the quoted region, respectively. No special characters (i.e., *, %, etc.) are allowed. *esc* represents an escape sequence which, appearing immediately in front of *str1* or *str2*, nullifies its special meaning. *str1*, *str2*, and *esc* can be null if the language does not support these concepts. *flags* can be any combination of the flags listed in following table.

Flag	Meaning
C	Quoted region is a comment
L	Newline terminates quoted region
N	Quoted regions can be nested
Q	Quoted region is a string literal
S	Single-character quoted region, for next character only
W	Beginning of line or whitespace must precede quote

Pattern Scan Rules

Define Rule

The define rule is used in conjunction with the option rule O|S|. If the S option is set, the define rule tells nmake how to recognize preprocessor definitions in the new language. nmake uses these definitions as declarations of candidate state variables (see Example 1, above). The format is:

D|*patt*|

The pattern, *patt*, is a character string followed by a percent character (%) to indicate the token being defined. Any whitespace in the string is interpreted as meaning that any sequence of spaces or tabs can be found in the source file.

For example, to tell nmake about the C language #define syntax, you would use:

D| \#define % |

The # character is escaped here because nmake would interpret the remaining content of the line as a comment (when not in the first column or line).

Begin Rule

The begin rule tells nmake's scanner that a conditional portion of text follows. The format is:

B|*patt*|

nmake's scanner does not parse the conditional expression to determine whether the conditional portion is actually included. Such a determination would require not only parsing the expression but the ability to evaluate it; nmake's scanner would have to keep track of the values for all tokens in the conditional expression. Given that the semantics of all languages vary considerably, this becomes a very difficult task.

Instead, nmake continues to scan for implicit file prerequisites in conditional portions of source text. Any references found are included in the implicit list but are marked with the .DONTCARE attribute. This attribute tells nmake not to indicate an error but to continue even if it cannot find a referenced file.

With this technique, nmake does not need to know about #else parts of conditionally included text (for the C language), or even about #elif parts. Everything from the start of conditional area (indicated in the begin rule) to the end of it (indicated in the end rule) is scanned, and all references are marked with the .DONTCARE attribute.

End Rule

The end rule allows nmake to detect the end of a conditional portion of source text. The format of this rule is:

E|*patt*|

Include Rule

The include rule specifies an included file reference. The format is:

I|*patt*|*action*|

For simple cases, *action* is null. The pattern, *patt*, is a character string that always contains a percent character to indicate the name of the file being referenced. For example, to write the scan rules for FORTRAN's include statement, you would use:

```
I| include '%' |
I| INCLUDE '%' |
```

patt can also contain an @ sign, which indicates that it could be followed by multiple files on the same line. nmake also supports non-line-oriented format, where the referenced files can be spanned over multiple lines. The general format is:

I|*begin_patt* **@ % [@** *end_patt***]**|

For example:

```
.SCAN.4gl : .SCAN
          I| include @ "%"|
          I| global @ "%" @ endglobal|
```

These include rules can scan source files containing the following:

```
include "a" "b" "c"

global
          "a"
          "b"
          "c"
endglobal
```

When *action* is not null, it can be any combination of valid scan actions as described above (for the pattern-type rules).

The A and M actions have a special meaning when used in context of the I directive:

- The A (attribute) action assigns *attribute-list* to the file matched (indicated by a percent character) as implicit prerequisites whenever a match of a *patt* occurs.

NOTE:
attribute-list is assigned as implicit prerequisites to the included file, not to the file being scanned. *attribute-list* is assigned as implicit prerequisites to the file being scanned for all other pattern and boundary rules.

- The M (map) action uses an expanded value of the following *expression* (instead of the file indicated by a percent character within the matched *patt*) to be the implicit prerequisite for the file being scanned (see Example 6, below).

Let us look at the example of the OX scan action.

Example 2: The OX Scan Language Action

Partial Contents of Makefile

```
SCAN.G : .SCAN
```

```
    .
    .
    .
    I | include % ; |OX|
    .
    .
    .
SRC = foo.G foobar.G
.MAIN : foo.incl - foo.G
```

Partial Contents of foo.G

```
include foo.incl ;
```

In this example, the file foo.incl is made before foo.G. When the foo.G target is made, nmake must scan the file using the specified scan strategy. Without the OX scan action, the implicit prerequisite, foo.incl, would normally inherit the .SCAN.G scan strategy even though it has already been bound and scanned. Since the OX scan *action* is applied to the include rule, all implicit prerequisites of $(SRC) (referenced with the include statement above and created by the include rule) do not inherit the .SCAN.G scan strategy regardless of whether the files are already bound.

Token Rule

The token *rule* matches literal patterns (as opposed to regular expressions or shell patterns). The format is:

T|*patt*|*action*|

The pattern, *patt*, is a character string. It can contain a percent character. *action* is any valid scan action as described above.

The R (read) *action* denotes that the *stmt* is read and evaluated by the nmake engine. Should the match of a *patt* occur, nmake will read and evaluate *stmt*.

Example 3: The Pattern Token Rule with the Read Scan Action

Partial Contents of Makefile

```
.SCAN.G : .SCAN
    .
    .
    .
    T| relid "%" |R$$(<) : .RELATION.ID|
                  .
                  .
                  .
```

In Example 3, the R scan *action* is read and evaluated whenever the token relid is found in the file being scanned. For this example, the *action* results in the .RELATION.ID attribute being assigned to the file being scanned.

Example 4: The Pattern Token Rule with the Attribute Scan Action

Partial Contents of Makefile

```
.REL.ID : .ATTRIBUTE
.INTERFACE : .ATTRIBUTE

.SCAN.G : .SCAN
    O|S|
    T| relid % |A.REL.ID .INTERFACE|
    .
    .
    .
```

This example shows that when the relid keyword is found in a source file being scanned, the source file is assigned the two attributes .REL.ID and .INTERFACE.

Aggregate Rule

The aggregate rule allows the user to do the archive (or aggregate) type of scanning. The format is:

A|*patt*|*action*|

The pattern, *patt*, is a string with two space-separated % characters in it. The first % matches the name of the archive member; the second % matches the time associated with it. *action* is a valid scan *action* as described above.

Boundary Scan Rules

Start Rule

The format of the start rule is:

S|*action*|

action is a valid scan *action* as described above; it is evaluated before the scanning starts.

Finish Rule

The format of the finish rule is:

F|*action*|

action is a valid scan *action* as described above; it is evaluated after the scanning is complete.

Example 5: The Boundary Rule with the Read Scan Action

Partial Contents of Makefile

```
.SCAN.G : .SCAN
      S|R$$(<) :  -RELATION|
      .
      .
      .
      F|R$$(<) : $$(!:Y/HAS_STATE/NO_STATE)|
```

In Example 5, the boundary start rule always removes the RELATION attribute (if found) from the file being scanned. Otherwise, no action is taken. After the file has been scanned, the boundary finish rule checks for all implicit and explicit file prerequisites. If any are found, the R scan *action* adds the new attribute, HAS_STATE; otherwise, the attribute NO_STATE is added to the file being scanned (*.G).

Example 6: The Boundary Rule with the Map Scan Action.

Partial Contents of Makefile

```
RELATIONS  := transfers.R newHires.R \
      displaced.R
.SCAN.G : .SCAN
      O|S|
      F|M $$(FUNCT)|
FUNCT : .MAKE .FUNCTIONAL .FORCE .REPEAT
      return $$(RELATIONS:A=$(&&))
```

In Example 6, the functional variable FUNCT is evaluated whenever nmake finishes scanning a source file (*.G). This functional variable returns the set of RELATIONS files (*.R) containing one or more state variables that were also found in the .G source files. The RELATIONS files become implicit prerequisites of the source .G file through the M scan *action*.

C/C++ Scan Rules Example

As a complete example, here is a scan strategy defined for the C/C++ languages[*]:

```
0:     CCFLAGS &= $$(.INCLUDE. c -I -I-)
1:     .LCL.INCLUDE: .ATTRIBUTE
2:     .PFX.INCLUDE: .ATTRIBUTE
3:     .STD.INCLUDE: .ATTRIBUTE
4:     .ACCEPT .IGNORE .RETAIN : .LCL.INCLUDE \
           .PFX.INCLUDE .STD.INCLUDE
5:     .SOURCE.%.LCL.INCLUDE   : .FORCE \
           $$(*.SOURCE.c) $$(*.SOURCE) $$(*.SOURCE.h)
6:     .SOURCE.%.STD.INCLUDE   : .FORCE $$(*.SOURCE.h)
7:     .SCAN.c : .SCAN
8:             O|S|
9:             Q|/*|*/||C|
10:            Q|//||\\|LC|
11:            Q|"|"|\\|LQ|
12:            Q|’|’|\\|LQ|
13:            Q|\\|||CS|
14:            D| \# define % |
15:            B| \# if|
16:            E| \# endif|
17:            I| \# include <%>|A.STD.INCLUDE|
18:            I| \# include "%"|A.LCL.INCLUDE|\
                       M$$$(.PREFIX.INCLUDE.)|
19:            I| \# pragma library "%"|A.VIRTUAL|A.ACCEPT|\
                       M.LIBRARY.$(%)|
20:
21:    .ATTRIBUTE.%.c: .SCAN.c
22:    .ATTRIBUTE.%.C: .SCAN.c
23:    .ATTRIBUTE.%.cc: .SCAN.c
```

Line 0 defines the -I flag for searching included files.

Lines 1-3 define new attributes for C/C++ language scan rule purposes. These user-defined attributes are given to the C and C++ source files based on their format (see the explanation for lines 17-18). Once listed as prerequisites for C/C++ files, we want nmake to retain them in the statefile (line 4). This way we know which attributes apply to a C/C++ file in case the corresponding header file is up to date and thus the C/C++ file is not checked.

Lines 5-6 deserve a special attention. Here, we use a slight deviation from step d on page 3. According to that rule, we should have defined the binding directories using .SOURCE.%.SCAN.c for the C language. For the C/C++ language though, we differentiate between two types of include files, namely, local include files (" ")

[*] This example is *not* taken verbatim from Scanrules.mk. Its main purpose is to demonstrate the concepts introduced in this chapter, not to concentrate on subtleties irrelevant to the chapter.

and standard include files (< >). Local include files are assigned the .LCL.INCLUDE attribute; standard include files are assigned the .STD.INCLUDE attribute by lines 17-18. Since we have two types of include files, we define binding directories for each set of files. If the include file is a local include file (e.g., #include "name.h"), the included file is found by searching the directories of .SOURCE.c first, then the directories of .SOURCE, and then the directories of .SOURCE.h (line 5). If the include file is a standard include file (e.g., #include <name.h>), the included file is found by searching the directories of .SOURCE.h (line 6).

Lines 7-19 define C/C++ scan strategy itself.

Let us look at line 10. This is the quote rule. *str1* corresponds to //, which introduces the quoted region (the beginning of the C++ style comment). *str2* in this case is null, and the escape sequence *esc* is \, not \\. The escape sequence is specified as two backslashes because it also must be escaped, since a backslash character is also a special character for nmake.

Line 18 uses the M (map) scan *action* by calling the .PREFIX.INCLUDE. function. Thus, whenever a match of a *patt* occurs, if the file name (indicated by a percent character) does not contain a prefix (directory component), the prefix from the including file (if it exists) is attached; the matched file then inherits all the implicit prerequisites of the including file.

The use of the M scan *action* in line 19 merits some elaboration. Here, *expression* evaluates to .LIBRARY.*file_name* whenever the *patt* match occurs, where *file_name* is the same as the % of the *patt* part of the include rule; the result of this evaluation becomes an implicit prerequisite for *file_name*. The $(%) part of *expression* corresponds to the % of *patt*.

Line 21 instructs nmake to apply the .SCAN.c scan strategy listed above (lines 7-19) whenever it is binding an atom whose name matches %.c. This method is how you assert the scan strategy for the C files. Likewise, line 22 instructs nmake to apply the same scan strategy for %.C files (C++). This way you write the scan strategy only once and can use it on both types of files.

Contents

Contents

Writing Makefiles

9

This chapter provides guidelines and suggestions for writing Alcatel-Lucent nmake makefiles when the software product/target is produced from multiple source files or from multiple source directories. Both nmake global makefiles and local makefiles are discussed. Also included is a discussion of how multiple source directories should be structured so that the common information can more readily be specified in the global and local makefiles. The common information may include such items as source file location, installation destination, common rules, and any third-party software packages that may be used in a build. Examples from existing software projects are discussed.

The Build Environment

A build environment consists of all the components necessary to build a software product. These components are:

- the hardware on which the product will be built
- the software used in building the product
- the directory structure
- the source files.

In order to take the most advantage of using Alcatel-Lucent nmake in building software products, you must understand these components. Specifically, you should design a directory structure with proper file partitioning. Once the directory structure is constructed, you can design the global makefiles and the local makefiles using a top-down approach.

The global makefile contains the build-environment specifications. The structure of a product directory and the partitioning of its source files can affect the design of the global makefile. The product directory-structure design also has an impact on the way viewpathing can be used.

Directory Structures

The principles of designing a directory structure for a project are:

- Make the structure scalable, in other words, design the directory structure so that it is easy to expand or shrink; and
- Partition the directory structure, which means to design subdirectories according to subsystems and/or features.

These concepts are described in greater detail in the sections that follow.

A recommended directory structure that incorporates these concepts is shown in Figure 9-1.

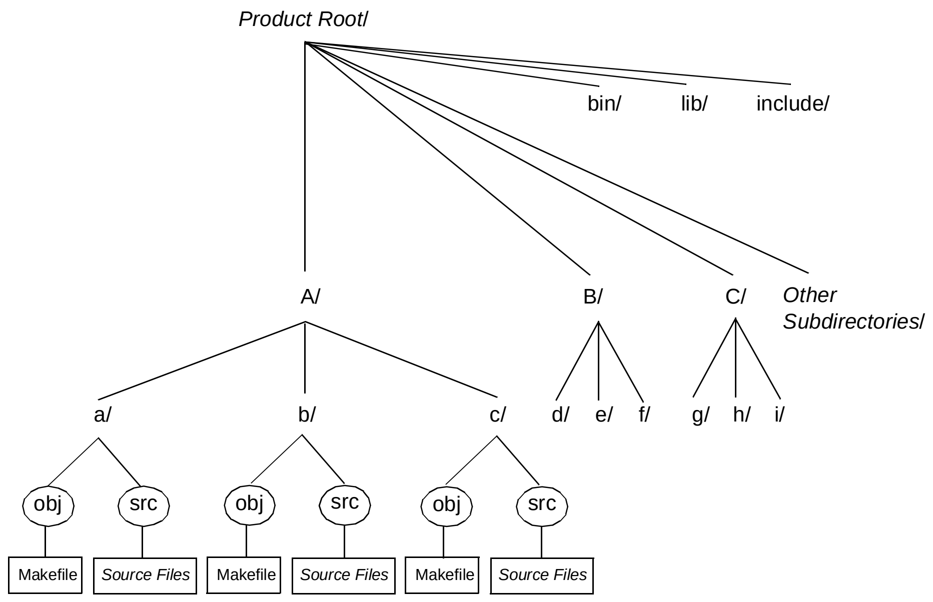


Figure 9-1. A Recommended Directory Structure

In the directory structure shown in Figure 9-1, the subdirectories A, B, and C are the subsystems of the project; and the subdirectories a to i are the features of the subsystems. The layout of each sub-directory is designed similarly. All the targets specified in Makefile under the subdirectory obj of the a to i directories are built from the source files in the ../src directories (except included files and libraries). Once the objects and targets are built by nmake, the targets are placed in the directories where the makefiles reside. All the targets will be installed (by using the install common action) into ../../../bin, ../../../lib, or ../../../include. This directory structure is both *scalable* and *partitioned*. Both of these terms are described in the sections that follow.

Scalable Directory Structure

In a scalable directory structure, each subdirectory must be designed similarly. In this way, you can easily expand or shrink a project by adding or deleting similar subdirectories. This adding or deleting of directories causes minimal impact on the information specified in the global makefile.

With the recommended directory structure, information such as the source file location and the target installation destinations can be easily specified in the global makefile because the information is the same for each of the subsystems.

For example, the various source file locations could be specified by using the following statements:

```
.SOURCE.c  : ../src
.SOURCE.h  : ../src ../../../include
.SOURCE.a  : ../../../lib
```

The various target installation destinations could be specified by using the following statements:

```
BINDIR = ../../../bin
LIBDIR = ../../../lib
INCLUDEDIR = ../../../include
```

If a similar subdirectory D is added under the *Product Root* directory, the building process for the targets under the subdirectory D can share the same common specification information that was specified in the global makefile without modification.

Partitioned Directory Structure

The directory structure should be partitioned according to the subsystems or features for each individual project. When designing a partitioned directory structure, you should also take into consideration different hardware architectures on which various portions of the project may be built. Software systems (e.g., cross-compilation systems) must also be considered. With viewpathing, a product or portions of a product can be built on one or more different hardware/software platforms. Directory structures that are partitioned properly enable viewpathing to be used more effectively

In the example shown in Figure 9-2, with viewpathing, targets can be built under the A, B, and C directories by using one architecture/compilation system. Then, with different viewpathing, other targets could be built under the C directory in the top view by using another architecture/compilation system.

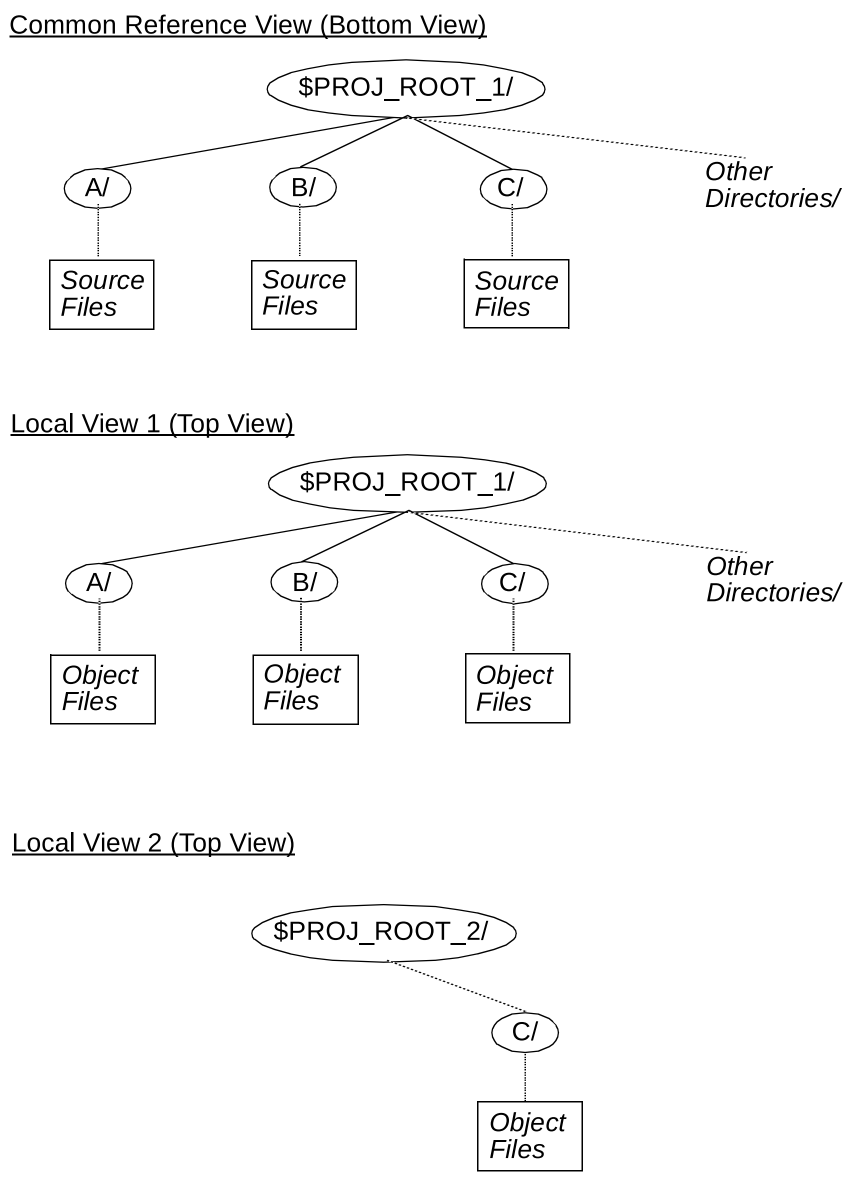


Figure 9-2. Viewpathing

The VPATH environment variable in this example could be set as follows:

```
export VPATH=$PROJ_ROOT_2:$PROJ_ROOT_1
```

where $PROJ_ROOT_2 contains the path to the area where the partial product can be rebuilt and $PROJ_ROOT_1 contains the path to the common reference area. This area can be referenced by the local area during the building process but does not have to be rebuilt. By having the directory structure partitioned, the top view ($PROJ_ROOT_2) would need only the C directory (and directories under the C branch) for a partial build. It is not necessary to create the A and B directories in the top view if they are not referenced during the building process.

File Names

Files should be partitioned properly under the directory structure according to the subsystems or features of each individual project as described in the previous section. In addition, file names can contain a suffix (e.g., *basename.suffix*). Different types of files can have different suffixes. Various rules can then be written that use a pattern-matching scheme to identify file types based on file suffixes.

Viewpathing Considerations

An improper directory structure can cause difficulties with viewpathing and with nmake global makefiles. You should avoid designing directory structures that are excessively deep. When viewpathing is used to build a portion of a product that is located deep down in the directory structure, unnecessary directories may have to be created on the top view in order to build the desired portion. An example of an excessively deep directory structure is shown in Figure 9-3.

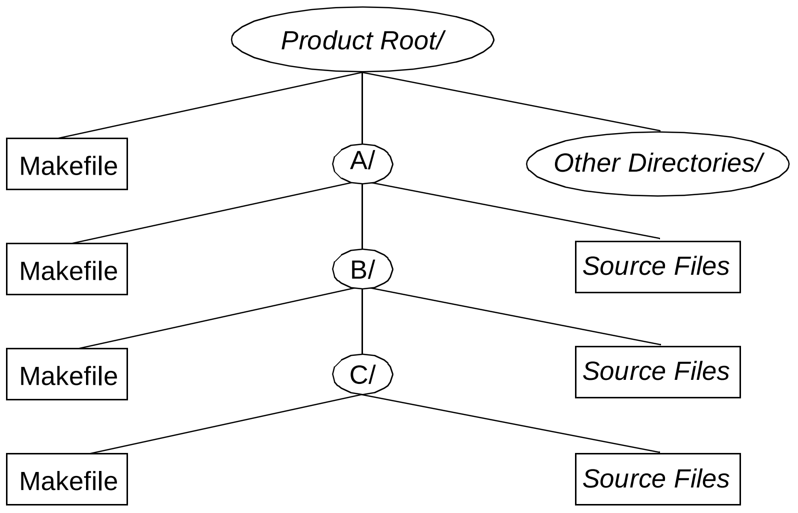


Figure 9-3. A Deep Directory Structure

In Figure 9-3, to use viewpathing to build the targets under the C directory, you must create the A and B directories as well on the top view, which may be unnecessary if those directories are not referenced during the building process.

Another reason for avoiding excessively deep directory structures is that the space may be used inefficiently and target information may be duplicated when propagating object files from lower directories to upper directories. For example, if an object file located in a lower directory is needed to make a target in a directory that is located above the location of the object file, the object file would then be duplicated in both the lower and the higher directory.

As another example, suppose a deep directory structure is designed as shown in Figure 9-4 in order to build a product using multiple architectures. In this design, viewpathing was not considered. One makefile is used to build one branch by the native compilation system and another makefile is used to build another branch by a cross-compilation system. A global makefile is not used in this example. Instead, all the specifications required for the build are contained in the two local makefiles. This type of design is not recommended.

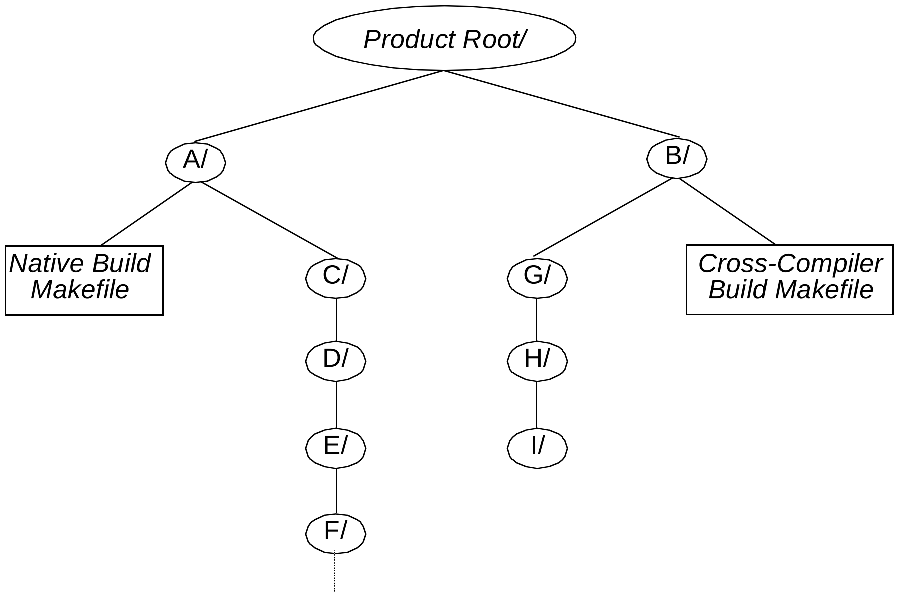


Figure 9-4. Two Architectures for One Build

With this approach, some of the specifications required in one of the makefiles must be duplicated in the other makefile. Duplication results in two large makefiles that contain redundant information. It also makes the build procedure more complex. In addition, expanding and shrinking the project can cause major changes in both of the local makefiles and make them hard to maintain.

A global makefile could be used to contain any information that is common for both architectures. Use of a global makefile would reduce the size and complexity of the local makefiles.

In fact, viewpathing could be used to build the product. If viewpathing is used, a directory structure similar to the one shown in Figure 9-1 is recommended because global and local makefiles can be easily written to simplify the build procedure as well.

You should also avoid designing a directory structure with improper file partitioning. Improper file partitioning can result in using more than one makefile to build a single target. An example of this type of improper file partitioning is shown in Figure 9-5.

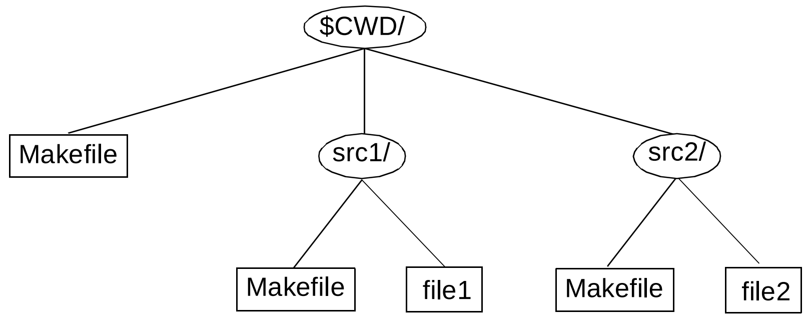


Figure 9-5. Improper File Partitioning

In Figure 9-5, the $CWD/Makefile contains the following:

```
:MAKE: src1 src2
```

The $CWD/src1/Makefile contains the following:

```
../lib.a :: file1
```

The $CWD/src2/Makefile contains the following:

```
../lib.a :: file2
```

In this example, the lib.a target is made by both the $CWD/src1/makefile and the $CWD/src2/makefile. This type of directory structure not only creates extra .mo and .ms files in both the $CWD/src1 and $CWD/src2 directories but also causes the modification time of the lib.a target to change between one nmake invocation and the next. The change in modification time can cause unnecessary rebuild actions in every nmake invocation.

To correct the situation, you can change the directory structure, repartition the files, or do both.

Global Makefiles

A global makefile contains the common specifications that can be shared when building different targets for a project. Generally, there is only one global makefile for each project.

When designing a makefile (either global or local), you should avoid using an excessive number of nmake variables and an excessive number of shell environment variables. A large number of variables makes the makefile difficult to maintain; run-time efficiency may also be affected. When writing the rules for a global makefile, you can use nmake automatic variables as well as edit operators and programming constructs. You should keep specifications of the same category together for maintenance purposes.

NOTE:
Be careful when using absolute paths in a makefile. Relative paths should be used whenever possible for the purpose of viewpathing.

A global makefile normally consists of the following specifications:

- Source file location information
- Third-party software package information
- Installation destination
- Transformation rules (also called metarules)
- Other special rules.

These specifications are discussed in the sections that follow.

Source File Location

The source file location is specified by using the .SOURCE and .SOURCE.*suffix* Special Atoms. Consider the following example.

```
.SOURCE  : ../src
.SOURCE.h : ../inc
.SOURCE.a : ../lib
```

In this example, the .SOURCE Special Atom specifies that any type of source file can be found in the ../src directory. The .SOURCE.h Special Atom specifies that source files with a .h suffix in the file name can be found in the ../inc directory. The .SOURCE.a Special Atom specifies that source files with a .a suffix in the file name can be found in the ../lib directory.

If the source file location is not specified, nmake by default always checks the current directory first.

Third-Party Software Information

Third-party software packages and their option flags that are to be used when building a product are specified by assigning the software process command names and flags to nmake variables. For example, if the INFORMIX software package is used, the INFORMIX file locations and flags can be specified as follows:

```
IESQL = $(INFORMIXDIR)/bin/esql
IESQLFLAGS = -e
```

where $(INFORMIXDIR) contains the path to where the INFORMIX software package is installed.

Installation Destination

The installation destination is the place where the product that is built by nmake is installed for public access or delivery purposes. Installation destinations can be specified by assigning the destination path names to nmake variables. If the installation destination directories are not specified, the predefined directories in the default base rules are used. For example, $(BINDIR) will be used for installing executables, $(LIBDIR) for installing libraries, $(MANDIR) for installing man pages, etc. The default values of those variables are as follows:

```
INSTALLROOT = $(HOME)
BINDIR = $(INSTALLROOT)/bin
LIBDIR = $(INSTALLROOT)/lib
MANDIR = $(INSTALLROOT)/man/man
```

The default value for INSTALLROOT is $(HOME). You can redefine INSTALLROOT to be the desired installation directory. The remaining predefined directories can also be used as installed or redefined. The predefined directory variables are listed in the *Alcatel-Lucent nmake Product Builder Reference Manual*.

Metarules

Metarules (also called transformation rules), normally included in a global makefile, use a pattern-matching scheme consisting of the following format:

```
[prefix2]%suffix2 : [prefix1]%suffix1
        action
```

A file pattern is recognized when the file name found matches the file pattern defined in *prefix1* and *suffix1*. The % matches the remaining characters in the string and is called the *stem*. Consider the y.file.c file name and the file pattern y.%.c. The prefix is y. and the suffix is .c; the stem in this case is file.

Using different suffixes for file names for different types of files can aid when writing transformation rules. For example, INFORMIX esql files always have a .ec suffix, and all the output files compiled from INFORMIX esql files have a .c as a suffix.

The transformation rule can then be specified as follows:

```
%.c : %.ec
        $(IESQL) $(IESQLFLAGS) $(>)
```

where $(IESQL) and $(IESQLFLAGS) contain the INFORMIX esql software processor name and corresponding flags. When this transformation rule is expanded, INFORMIX esql compiles the .ec source file and produce the desired .c file.

Other Special Rules

Special rules can be specified in a global makefile that are useful for building different targets. This section describes a few of the more common ones.

Scan Rules

You can specify a scan strategy that is to be used for scanning the implicit prerequisites (the included files) for a specific type of file. For example, the INFORMIX .ec files use sql style to include header files. The following assertion specifies that the sql scan strategy should be used when scanning files that contain a .ec suffix:

```
.ATTRIBUTE.%.ec : .SCAN.sql
```

where .SCAN.sql is predefined in the base rules. See Chapter 8, *Defining Custom Scan Rules*, for more information about scan rules.

.USE Rules

You can define a special task by using the .USE Special Atom. The .USE Special Atom allows the user to specify a series of actions that can then be used in any number of assertions. In the following example, a special task is defined to cat files into a file and then run that file through *troff* and also print the output.

```
.DO.CATROFF : .USE
        cat $(*) > $(<)
        troff -Tpost -mm $(<) | lp -dpost
```

When this rule is used in an assertion line, such as:

```
file1 : file2 file3 .DO.CATROFF
```

the action defined in the .DO.CATROFF rule are applied to this assertion during execution. That is, file2 and file3 will be concatenated into file1. file1 is then processed by *troff* and the output is printed.

.OPERATOR Rules

You can define an assertion operator by using the .OPERATOR Special Atom. A .OPERATOR rule uses its abstraction to hide the complicated actions in making a target file. For example, you can define a .OPERATOR rule as follows:

```
":CATROFF:" : .OPERATOR .MAKE
        NFILES += $(<)
        $(<) : $(>) .DO.CATROFF
```

NOTE:
Any name containing a colon (:) must be surrounded by double quotation marks to prevent confusion with edit or assertion operators.

When this .OPERATOR rule is used in an assertion such as:

```
file1 :CATROFF: file2 file3
```

file1 is added to NFILES. The action defined in .DO.CATROFF (described previously) is used. The abstraction of the .OPERATOR rule hides the fact that the .DO.CATROFF rule is one of the prerequisites of the target.

.FUNCTIONAL Rules

A .FUNCTIONAL rule returns a value. With the .FUNCTIONAL rule, you can define a target as an nmake variable. The value of that variable can then be obtained by a specific action of the .FUNCTIONAL rule. For example, one may want the CSRC variable to contain a list of .c files that are derived from a list of various types of source files. The .FUNCTIONAL rule to do this can be specified as follows:

```
CSRC : .FUNCTIONAL .MAKE .FORCE .REPEAT
        return $(**:N=*.c)
```

When the CSRC variable is expanded during execution, this .FUNCTIONAL rule will be invoked and the value of CSRC will be obtained from the action of this rule. For example:

```
csrc : a.c b.x c.c d.y
        : C files are $(CSRC)
```

Here, csrc has a prerequisite list containing different types of files. When $(CSRC) is expanded, the CSRC .FUNCTIONAL rule is invoked. The CSRC variable obtains the value from the action of the rule, which in this case is a.c c.c.

Special Actions

Special common actions can also be specified in global makefiles. For instance, a file backup can be defined as a special action that performs source files backups when needed. For example, the backup action can be specified as follows:

```
backup : .CPIO
        $(MV) $(output:D:B:S=.cpio) $(BAKDIR)
```

This rule packs files into one file and then moves the packed files into a backup directory, $(BAKDIR). Here, .CPIO is defined in the base rules; it packs files into a .cpio file by using cpio.

Global Makefile Example

In this section, we examine a global makefile that was used on a product developed within AT&T. The product that used this global makefile incorporated a number of different software packages, so different specifications were needed. Nevertheless, the global makefile is easy to understand. This global makefile is a typical example of a global makefile that can be used to build medium- to large-sized software products. Various nmake specification language features are used throughout this example.

The following information is specified in the global makefile:

- Source file location. The source file location relative to the local makefile location is ../src.
- Installation destination. The installation destination is three levels above the local makefiles.
- Third-party software. The software packages used by the project are awk, C5, G2, TUXEDO, and INFORMIX.
- Other information. Target files are built from different types of files. The file type is determined by the suffix of the file name.

The global makefile discussed in this section is included in its entirety at the end of this chapter. In this section, various portions of the makefile are presented. The line numbers used in this section match the line numbers that appear in the makefile at the end of this chapter.

- Lines 1–3 define the installation destination directories by assigning directory names to nmake variables. INSTALLROOT is redefined. The directory names defined in the base rules (i.e., BINDIR, LIBDIR, MANDIR, ETCDIR, INCLUDEDIR, etc.) are used; therefore, it is not necessary to redefine them.

```
1:     /*Define installation destination directories*/
2:     INSTALLROOT = $(VROOT)
3:     MASKDIR = $(INSTALLROOT)/mask
```

- Lines 5–34 specify third-party software package information; lines 8–34 define the locations of different software packages and initialize the option flags employed in using those packages.

```
5:     /* Define software package locations and their
6:     * flags. */
7:
8:     AWKDIR = /usr/add-on/awkcc /* awk package */
```

```
9:   AWKCC = $(AWKDIR)/bin/awkcc
10:  AWKCCFLAGS =
11:
12:  C5DIR = /usr/add-on/aitools /* C5 package */
13:  CC5 = $(C5DIR)/bin/cc5
14:  CC5FLAGS = -h -Z
15:
16:  G2DIR = /usr/add-on/g2 /* g2 package */
17:  G2CC = $(G2DIR)/bin/g2comp
18:  G2CCFLAGS=
19:
20:  INFORMIXDIR = /usr/add-on/informix   /* INFORMIX
21:                                               package */
22:  IESQL = $(INFORMIXDIR)/bin/esql
23:  IESQLFLAGS =
24:  ISFORMBLD = $(INFORMIXDIR)/bin/sformbld
25:  ISFORMBLDFLAGS =
26:
27:  TUXDIR = /tux/topas_units /* TUXEDO package */
28:  TUX.FTBL = $(TUXDIR)/udataobj/Usysflds
29:  BLDSRVR = $(TUXDIR)/bin/buildserver
30:  BLDSRVRFLAGS = -v
31:  MC = $(TUXDIR)/bin/mc
32:  MCFLAGS = -r 40
33:
34:  LNFLAGS = -s /* flag for symbolic link */
```

- Lines 37–41 assert the .PARAMETER Special Atom to the nmake variables, which are defined on lines 8–34, to prevent these variables from being expanded by the T=D: or T=E: edit operators.

```
37:  (BLDSRVR) (BLDSRVRFLAGS) (MC) : .PARAMETER
38:  (MCFLAGS) (CC5) (CC5FLAGS) (G2CC) : .PARAMETER
39:  (G2CCFLAGS) (AWKCC) (AWKCCFLAGS) : .PARAMETER
40:  (IESQL) (IESQLFLAGS) (ISFORMBLD) : .PARAMETER
41:  (ISFORMBLDFLAGS) (LNFLAGS) : .PARAMETER
```

- Lines 44–49 define the source file locations including the locations of header files and libraries.

```
44:  ancestor = 3
45:  .SOURCE : ../src
46:  .SOURCE.a : $(TUXDIR)/lib $(INFORMIXDIR)/lib \
47:          $(G2DIR)/lib $(C5DIR)/lib
48:  .SOURCE.h : $(TUXDIR)/include $(INFORMIXDIR)/incl \
49:          $(G2DIR)/include $(C5DIR)/include
```

- Lines 52–125 define different transformation rules and scan rules.
- Lines 52–55 specify the transformation rule for compiling awk scripts into executables. The CLEAN.TEMPS rule is asserted as one of the prerequisites of the target. The rule is defined on lines 127–128 and is used for removing any generated intermediate files.

```
52:  % : %.awk (AWKCC) (AWKCCFLAGS) CLEAN.TEMPS
53:          : "compiling awk files: $(!)"
54:          $(AWKCC) $(AWKCCFLAGS) -c $(%).c \
55:                  -o $(<) $(>)
```

- Line 58 specifies the scan rule for INFORMIX esql files (.ec files use sql style to include header files).

```
58:  .ATTRIBUTE.%.ec : .SCAN.sql
```

- Lines 60–62 specify the transformation rule to generate .c files from .ec files (the INFORMIX esql files).

```
60:  %.c : %.ec (IESQL) (IESQLFLAGS)
61:          : "ESQL prerequisites are: $(!)"
62:          $(IESQL) $(IESQLFLAGS) -e $(>)
```

- Lines 65–67 specify the transformation rule to generate .c and .h from .g files (the g2 files).

```
65:  %.c %.h : %.g (G2CC) (G2CCFLAGS)
66:          : "compiling g2 record: $(!)"
67:          $(G2CC) $(G2CCFLAGS) $(>)
```

- Lines 71–77 specify the transformation rule to generate .o files from .c5 files (the C5 files). This rule specifies linking the .c5 files to the current directory if they are not in the current directory. This linking is required for those processes that take input files only from the current directory, or generate output files only in the same directory where the source is found. The temporary linked files will be removed at the end of nmake execution since CLEAN.TEMPS is one of the prerequisites.

```
71:  %.o : %.c5 (CC5) (CC5FLAGS) (LDFLAGS) CLEAN.TEMPS
72:          : "compiling c5 files: $(!)"
73:          if [ ! -f $(>:B:S) ]
74:          then
75:                  $(LN) $(LNFLAGS) $(>) .
76:          fi
77:          $(CC5) $(CC5FLAGS) -c $(>:B:S)
```

- Lines 80–91 specify the transformation rule for generating .frm files from .per files (the INFORMIX perform files). This rule verifies whether the environment variable DBPATH is set. It also specifies linking the prerequisites (the .per files) to the current directory if the prerequisites are not in the current directory for the reasons mentioned above. The temporary files will be removed at the end of nmake execution since CLEAN.TEMPS is one of the prerequisites.

```
80:  %.frm : %.per (ISFORMBLD) (ISFORMBLDFLAGS) \
81:          CLEAN.TEMPS
82:          if [ -z "$DBPATH"]; then
83:                  : "ERROR - DBPATH must be set to \
84:                          compile PERFORM files"
85:                  false
```

```
86:          fi
87:          if [ ! -f $(>:B:S=.per) ]; then
88:                  $(LN) $(LNFLAGS) $(>) .
89:          fi
90:          : "FRMBLD prerequisites are: $(!)"
91:          $(ISFORMBLD) $(ISFORMBLDFLAGS) $(>:D:B)
```

- Line 104 specifies the scan rule for the TUXEDO mask files; .m files use the m4 preprocessor. In other words, it uses m4 style to include header files.

```
104: .ATTRIBUTE.%.m : .SCAN.m4
```

- Lines 107–125 specify the transformation rule for generating .M files from .m files. It also links the prerequisites to the current directory if the prerequisites are not in the current directory. The reason for this linking is that the mc processor builds masks in the directory where the source for the masks is found. With viewpathing, it is necessary to link the files to the current directory. Any temporary files will be removed at the end of nmake execution because CLEAN.TEMPS is one of the prerequisites. mc execution also depends on two environment variables: FIELDTBLS and FLDTBLDIR; therefore, these two variables are set in the action.

```
107: %.M : %.m (MC) (MCFLAGS) $(TUX.FTBL) CLEAN.TEMPS
108:         : "MASK prerequisites are: $(!)"
109:         for i in $(!:N=*.fld|*.m)
110:         do
111:                 case "$i" in
112:                         *.m )       if [ ! -f ‘basename $i‘ ]
113:                                     then
114:                                           $(LN) $(LNFLAGS) $i .
115:                                     fi ;;
116:                         *.fld )echo '* temporary nmake field \
117:                                     table'>‘basename $i‘
118:                                     ;;
119:                      * )  ;;
120:                 esac
121:         done
122:         FIELDTBLS=$(SET.FTBL)
123:         FLDTBLDIR=‘pwd‘
124:         export FLDTBLDIR FIELDTBLS
125:         $(MC) $(MCFLAGS) $(>:B:S)
```

- Lines 127–183 define different special rules.
- Lines 127–128 specify the removal of generated intermediate files at the end of nmake execution. The CLEAN.TEMPS rule is also one of the prerequisites of the transformation rules listed on lines 52, 60, 65, 71, 80, and 107. The .AFTER Special Atom is also a prerequisite. Therefore, the action for this rule will be executed after the completion of the action of those rules that have this rule as prerequisite. This rule also specifies that

the CYAN.CLEAN rule should be invoked at the end of nmake execution. CYAN.CLEAN is defined on lines 166–172 and is used to remove generated intermediate files.

```
127:  CLEAN.TEMPS : .MAKE .AFTER .VIRTUAL .FORCE
128:          .DONE : CYAN.CLEAN
```

- Line 132 defines a user-defined attribute called .IS.A.FTBL. This user-defined attribute is used to identify *field table* files.

```
132:  .IS.A.FTBL : .ATTRIBUTE
```

- Line 135 specifies that all files with a .fld suffix are asserted with the .IS.A.FTBL attribute to identify them as *field table* files.

```
135:  .ATTRIBUTE.%.fld : .IS.A.FTBL
```

- Line 138 specifies that the standard TUXEDO field table is also asserted with the .IS.A.FTBL attribute to identify those files as *field table* files.

```
138:  $(TUX.FTBL) : .IS.A.FTBL
```

- Lines 142–147 define a .FUNCTION rule that is executed when the target of the rule is expanded (on line 122). The .FUNCTION rule is defined in the default base rules file, Makerules.mk. It is associated with the following attributes: .USE, .ATTRIBUTE, .MAKE, .FUNCTIONAL, .VIRTUAL, .FORCE, and .REPEAT. This rule builds a list of field table files, separating each file with a comma.

```
142:  SET.FTBL : .FUNCTION
143:          local FT FL
144:          for FT $(!!:A=.IS.A.FTBL)
145:                  FL := $(FL:Y?$(FL),??)$(FT)
146:          end
147:          return $(FL)
```

- Lines 151–156 define a .USE rule used for building a TUXEDO server. The target that has this rule as prerequisite will be built by using the action defined in this rule, if the assertion of the target does not contain its own action.

```
151:  .DO.SERVER : .USE (BLDSRVR) (BLDSRVRFLAGS) (CC) \
152:          (CCFLAGS) (LDFLAGS) \
153:          ($$(<:B:S=_SERVICE)) -ltux -lfml -lPW -lgp
154:          : "BSRVR prerequisites are: $(!)"
155:          $(BLDSRVR) $(BLDSRVRFLAGS) -o $(<) -f "$(*)" \
156:                  $($(<:B:S=_SERVICE):C/^/-s/)
```

- Lines 159–163 define a .OPERATOR rule. This operator rule uses its abstraction to hide the .DO.SERVER rule (lines 151–156) as one of the prerequisites. This rule can also use the action defined in the assertion, which uses this operator rule as an alternative, instead of using the action defined in the .DO.SERVER rule.

```
159:  ":SERVER:" : .OPERATOR .MAKE
```

```
160:        eval
161:                $(<) : $(>) .DO.SERVER
162:                        $(@:V)
163:        end
```

- Lines 166–172 define another .USE rule used for cleaning up any temporary files.

```
166:  CYAN.CLEAN : .USE .VIRTUAL .FORCE .REPEAT
167:        CURDIR=‘pwd‘
168:        if  [ ${CURDIR##*/} = obj ]
169:        /*equivalent to [ "‘basename $CURDIR‘" = obj ]*/
170:        then
171:                $(RM) $(RMFLAGS) *.fld *.m *.per *.c5 *.c
172:        fi
```

- Line 177 sets the base rule clobber variable to 1. This instruction informs nmake not to move an existing installed target to a file called *target*.old in the installation directory.

```
177:  clobber = 1
```

- Lines 180–183 contain two assertions that specify the startup control. Line 180 specifies that each target should be built and then installed if necessary. Line 184 specifies that the first thing to be done during nmake execution is to clean up all the generated temporary files from the last build (if there are any).

```
179:  /* build everything and install if necessary */
180:  .MAIN : .ALL .INSTALL
181:
182:  /* clean up the directory if last build wasn't
183:    *a complete one */
184:  .INIT : CYAN.CLEAN
```

This concludes the discussion regarding global makefiles.

Local Makefiles

Local makefiles mainly contain assertions that are used to build target files from source files. Local makefiles may also contain specifications that have not been defined in the base rules or in the global rules. The following sections discuss assertions and additional specifications that can be included in the local makefile. Samples of some local makefiles are also presented.

Assertions

When you write assertions in local makefiles, the assertion operators defined in the base rules are often used, such as ::, :LIBRARY:, :INSTALLDIR:, etc. In addition, any user-defined assertion operators in the global rules may also be used.

For example, a local makefile can be as simple as:

```
SRC = a.c b.c c.c
abc :: $(SRC) main.c
xyz :LIBRARY: x.c y.c z.c $(SRC)
```

In this example, the SRC variable contains a list of files. This variable is used as one of the prerequisites for building the abc target and the xyz target. The abc target is built by the :: assertion operator, and the xyz target is built by the :LIBRARY: assertion operator.

Additional Information

Any specifications in local makefiles are of a higher precedence than the specifications defined in either the base rules or global rules. Therefore, additional specifications can be added to the base rules or global rules in local makefiles. The information specified in the local makefile may provide additional specifications or perhaps redefine the action to an existing rule.

Other information can also be defined in local makefiles, such as new rules or new variables. The specification information defined in local makefiles is local to the particular makefile. Any rules that can be shared with other targets in other local makefiles of the project should be included in the global makefile.

In summary, a local makefile may contain the following specifications:

- Additional source file location information
- Additional software packages and corresponding flag information
- Additional installation destination information
- Additional rules.

Examples of Local Makefiles

This section contains some examples of local makefiles. The rules defined in the global makefile described previously are used by these local makefiles.

Example 1

This local makefile contains assertions to do the following:

- Build a library called libg2msgs.a from .g files by using the :: assertion operator (line 15). This assertion implies that the transformation rule of generating .c and .h files from .g files (which is defined in the global makefile) is applied during the building process. It also implies that the transformation rule of generating .o from .c is also applied (which is defined in the base rules).
- Install the .h files generated from .g files by using the :INSTALLDIR: assertion operator (line 18).
- Install .g files by using the :INSTALLDIR: assertion operator (line 21).

  The contents of the makefile are shown below.

```
1:     /* The variable is local only to this makefile. */
2:     G2ETCDIR = ../../../g2etc
3:
4:     /* Define g2 source files. */
5:     G2SRC = p_eptr.g p_tptr.g p_eqd.g p_rreq.g \
6:             p_psvc.g cc.g ck.g cn.g co.g go.g hk.g hr.g \
7:             jk.g jn.g link_start.g ncd_req.g ok.g ra.g \
8:             rn.g TSIrec.g ccis_dat.g ckt_ff_dat.g
9:
10:    /* Build the library, libg2msgs.a from .c files
11:     * which are generated from g2 source files.
12:      * The metarule of generating .c files from .g
13:     * files is applied.
14:     */
15:    libg2msgs.a :: $(G2SRC)
16:
17:    /* Install .h files generated from .g files. */
18:    $(INCLUDEDIR) :INSTALLDIR: $(G2SRC:B:S=.h)
19:
20:    /* Install .g files. */
21:    $(G2ETCDIR) :INSTALLDIR: $(G2SRC)
```

Example 2

This local makefile contains assertions to do the following:

- Build a library called libmtcif.a from different types of files including .c, .ec and .g files by using the :: assertion operator (line 17). This assertion also implies that the transformation rules for generating .c from .ec files and generating .c from .g files are applied in the building process. Again, the transformation rule of generating .o from .c is also applied.
- Build a TUXEDO server called MTCIF by using a user-defined operator called :SERVER: (line 26). The user-defined operator :SERVER: is defined in the global makefile.

The contents of the makefile are shown below.

```
/* Library files */
CORE_LIBS = Glib.a libtm.a liber.a libtgp.a \
          libg2.a libdb.a prv_state.a prv_ord.a \
          libprv.a libg2msgs.a P_trace.a Glib.a

/* Source files */
MTCIF_SRC = MTCgetg2.c MTCstatmch.c MTCIFstart.c \
          MTCcircuit.c MTCcprsend.c MTCcpr_upd.c \
          MTCfailure.c MTCsuccess.c MTCcirupd.c \
          MTCupdbuff.c MTCdefsend.c MTCdef_upd.c

/* Build the library from different type
 * of files.
 * Metarule of generating .c from .ec is applied.
 * Metarule of generating .c from .g is applied.
 */
libmtcif.a :: MTCgrpprs.ec MTCpendupd.c MTCg2.g

/* Variable used in .DO.SERVER rule which is
 * used in :SERVER: operator rule */
MTCIF_SERVICE = NPRV_MTCIFsvc:MTCIFsvc

/* Build server with user defined\
 * operator, :SERVER: */
MTCIF :SERVER: $(MTCIF_SRC) $(CORE_LIBS) libmtcif.a
```

Example 3

This local makefile specifies the assertions to:

- Build a library (line 30). Note that CCFLAGS is redefined to include $$(CC.PIC) (line 3). The library can be built as a shared library by the :LIBRARY: assertion operator if $$(CC.PIC) contains the flag for generating position-independent code. In the library-building process, the transformation rule of generating .c from .ec files is applied. The transformation rule of generating .o from .c is also applied.
- Build two application executables (lines 40 and 47); both executables will be linked with the same set of libraries defined in LDLIBRARIES.

The contents of the makefile are shown below.

```
1:    /* This is only local to this Makefile.
2:    */
3:    CCFLAGS += $$(CC.PIC)
4:
5:    /* The source files for building one of the
6:     * application executables
7:     * Notice the different suffixes of the files
```

```
 */
SRC = VALutil.ec VALai_proc.ec VALc_proc.c \
        VALreg_cmp.c5 VALdb_btf.c5 VALdb_ckt.c5 \
        VALdb_grp.c VALwm_ckt.c VALwm_grp.c \
        VALmrid.c

/* Local libraries used */
LIBS1 = Glib.a P_trace.a libtm.a libaq.a libad.a \
        libgf.a libum.a liber.a formlib.a libsql.a

/* Third-party software libraries used */
LIBS2 = libsql.a libg2.a libcc5.a

/* Libraries to be linked to all executable targets */
LDLIBRARIES += -lber -ltgp -ltux -lfml -lgp -lPW

/* Build the library with :LIBRARY: operator.
 * If CC.PIC is set, the library will be built\
 * as shared library.
 * The metarule of generating .c from .ec files
 * will be applied.
 */
val :LIBRARY: VALtty_msg.c VALstats.c VALia.c

/* Build the executables, valmgr and valg2. */

/* The metarule of generating .c from .c5 and
 *  generating .c from .ec files
 * will be applied.
 * In addition to $(LIBS1) and $(LIBS2), the files \
 * listed in LDLIBRARIES will also be linked.
 */
valmgr :: $(SRC) $(LIBS1) $(LIBS2)

/* The metarule of generating .c from .g files
 * will be applied.
 * In addition to $(LIBS1), the files \
 * listed in LDLIBRARIES will also be linked.
*/
valg2 :: VALg2.g VALnoenvbuf.c $(LIBS1)
```

Example 4

The following makefile is an example of local exceptions. This makefile contains additional specifications that supplement those defined in the global makefile (discussed previously) and the default base rules. This makefile specifies additional source file location information (line 5) and also defines the transformation rule for files containing a .sh suffix (lines 8–10). This transformation rule is different from the one defined in the base rules. This makefile also triggers quite a few of the metarules defined in the global makefile previously discussed.

The contents of the makefile are shown below.

```
/* Additional source file location information is
 * appended to the global information.
 * This information is local to this Makefile only.
 */
.SOURCE : ../awk_src ../per_src ../man ../data

/* Transformation rule local to this Makefile. */
%.rpt : %.sh (CP)
        $(CP) $(>) $(<)
        $(CHMOD) u+w,+x $(<)

/* The transformation rule for compiling awk
 * scripts into executables is applied. */
dattsa2f :: dattsa2f.awk
dattsa3f :: dattsa3f.awk

/* The transformation rule of generating .M from
 * .m files is applied.
 * Install the resulting .M files.
 */
$(MASKDIR) :INSTALLDIR: CTS_2.M TSC_5.M TRAC.M

/* The transformation rule of generating .frm
 * from .per files is applied.
 * Install the resulting .frm files.
 */
$(ETCDIR) :INSTALLDIR: lsec.frm wctr.frm wstn.frm

/* The transformation rule defined in \
 * this Makefile
 * for generating .rpt from .sh files is applied.
 * Install the resulting .rpt files.
 */
$(BINDIR) :INSTALLDIR: LIST.rpt PENDAUDIT.rpt

/* Install man page files */
$(MANDIR)1 :INSTALLDIR: TICKT.1 COVER.1
$(MANDIR)3 :INSTALLDIR: PENDAUDIT.3 LIST.3

/* Install data files */
$(ETCDIR) :INSTALLDIR: fmt.d grp.d dkt.d
```

Additional Techniques

This section describes some additional techniques that can be used when designing makefiles.

Reading Shell Output

The :COMMAND: assertion operator can be used to assign an nmake variable a value from the shell. In the following example, the output value from the cat list command is the value assigned to the FILES variable. In other words, FILES contains the contents of the file called list. Since list is the prerequisite of this rule, when the contents of list get changed, $(FILES) reflects the change.

```
FILES :COMMAND: list
        cat list

tst :
        : $(FILES)
```

Recursion on Different Makefile Names

When you use the :MAKE: assertion operator to do implicitly recursive builds, if the lower-level makefiles are named after the directory names with the suffix of .mk, you can specify the nmake variable MAKEFILES in the upper-level makefile as illustrated in the following example.

```
MAKEFILES := $(MAKEFILES):$$(PWD:B).mk
.EXPORT : MAKEFILES
:MAKE: dir1 dir2 dir3
```

Changing Target Permissions at Installation

The :INSTALLDIR: assertion operator can be used to change the permissions of a target at installation. In the following example, user= provides the option to change the owner ID, group= provides the option to change the group ID, and mode= provides the option to change the permission mode of the files.

```
$(BINDIR) :INSTALLDIR: user=root group=adm \
mode=4770 file1 file2
```

Generated Files

nmake detects only source files that nmake builds as targets or that exist prior to nmake execution because nmake scans directories at the beginning of execution.

Whenever a file is to be generated during nmake execution, it is recommended that the file be specified as a target in an assertion, instead of having the file generated as a side-effect (as shown in the following example).

Consider the following makefile.

```
all : label file3
label : file1
        cp file1 file2
```

```
file3 : file2
        action
```

In this example, the build will fail because file2 is generated as a side-effect. When nmake attempts to build file3, nmake cannot detect the existence of file2 because file2 is not a target that was built by nmake or existed prior to nmake execution.

In order to correct this situation, file2 should be defined as a target as shown in the following makefile.

```
file2 : file1
        cp $(*) $(<)

file3 : file2
        action
```

In this example, file2 is a target that is built during nmake execution. When nmake attempts to build file3, nmake detects the existence of file2.

Another way of correcting the situation shown in the first example would be to place the two assertions in two different makefiles and/or in separate subdirectories. In this way, you can ensure that the first assertion is always made first; therefore, the generated file, file2, would exist before nmake attempted to build the target, file3.

Complete Global Makefile Listing

This section contains the complete listing of the global makefile that was discussed in the section *Global Makefile Example*, above.

```
/*Define installation destination directories*/
INSTALLROOT = $(VROOT)
MASKDIR = $(INSTALLROOT)/masks

/* Define software package locations and their
 * flags. */

AWKDIR = /usr/add-on/awkcc      /* awk package */
AWKCC = $(AWKDIR)/bin/awkcc
AWKCCFLAGS =

C5DIR = /usr/add-on/aitools     /* C5 package */
CC5 = $(C5DIR)/bin/cc5
CC5FLAGS = -h -Z

G2DIR = /usr/add-on/g2          /* g2 package */
G2CC = $(G2DIR)/bin/g2comp
G2CCFLAGS=

INFORMIXDIR = /usr/add-on/informix   /* INFORMIX
                                                               package */
IESQL = $(INFORMIXDIR)/bin/esql
IESQLFLAGS =
ISFORMBLD = $(INFORMIXDIR)/bin/sformbld
ISFORMBLDFLAGS =

TUXDIR = /tux/topas_units          /* TUXEDO package */
TUX.FTBL = $(TUXDIR)/udataobj/Usysflds
BLDSRVR = $(TUXDIR)/bin/buildserver
BLDSRVRFLAGS = -v
MC = $(TUXDIR)/bin/mc
MCFLAGS = -r 40

LNFLAGS = -s /* flag for symbolic link */

/* stop expansion of these variables */
(BLDSRVR) (BLDSRVRFLAGS) (MC) : .PARAMETER
(MCFLAGS) (CC5) (CC5FLAGS) (G2CC) : .PARAMETER
(G2CCFLAGS) (AWKCC) (AWKCCFLAGS) : .PARAMETER
(IESQL) (IESQLFLAGS) (ISFORMBLD) : .PARAMETER
(ISFORMBLDFLAGS) (LNFLAGS) : .PARAMETER

/* Define source file search paths */
ancestor = 3
.SOURCE : ../src
.SOURCE.a : $(TUXDIR)/lib $(INFORMIXDIR)/lib \
        $(G2DIR)/lib $(C5DIR)/lib
.SOURCE.h : $(TUXDIR)/include $(INFORMIXDIR)/incl \
        $(G2DIR)/include $(C5DIR)/include

/* rules to compile awk scripts into executables */
% : %.awk (AWKCC) (AWKCCFLAGS) CLEAN.TEMPS
        : "compiling awk files: $(!)"
        $(AWKCC) $(AWKCCFLAGS) -c $(%).c \
        -o $(<) $(>)

/* rules to compile INFORMIX ESQL files */
.ATTRIBUTE.%.ec : .SCAN.sql

%.c : %.ec (IESQL) (IESQLFLAGS)
        : "ESQL prerequisites are: $(!)"
        $(IESQL) $(IESQLFLAGS) -e $(>)


/* rule to compile g2 files. */
%.c %.h : %.g (G2CC) (G2CCFLAGS)
        : "compiling g2 record: $(!)"
        $(G2CC) $(G2CCFLAGS) $(>)

/* rule to compile C5 files */
```

```
%.o : %.c5 (CC5) (CC5FLAGS) (LDFLAGS) CLEAN.TEMPS
        : "compiling c5 files: $(!)"
        if [ ! -f $(>:B:S) ]
        then
                $(LN) $(LNFLAGS) $(>) .
        fi
        $(CC5) $(CC5FLAGS) -c $(>:B:S)

/* rule to compile INFORMIX PERFORM files */
%.frm : %.per (ISFORMBLD) (ISFORMBLDFLAGS) \
        CLEAN.TEMPS
        if [ -z "$DBPATH"]; then
                : "ERROR - DBPATH must be set to \
                        compile PERFORM files"
                false
        fi
        if [ ! -f $(>:B:S=.per) ]; then
                $(LN) $(LNFLAGS) $(>) .
        fi
        : "FRMBLD prerequisites are: $(!)"
        $(ISFORMBLD) $(ISFORMBLDFLAGS) $(>:D:B)

/* The mc processor builds masks in the directory
 * where the source for the masks is found.  In
 * using viewpathing, this means that the masks
 * will be built into whatever node the mask source
 * is found. The following rule links all the
 * explicit and implicit prerequisites into  the
 * current directory. The masks will be built there
 * after which all of the .m's and .fld's will be
 * removed.  */

/* masks are run through the m4 preprocessor */
.ATTRIBUTE.%.m : .SCAN.m4

/* rules to compile TUXEDO masks */
%.M : %.m (MC) (MCFLAGS) $(TUX.FTBL) CLEAN.TEMPS
        : "MASK prerequisites are: $(!)"
        for i in $(!:N=*.fld|*.m)
        do
                case "$i" in
                        *.m )        if [ ! -f ‘basename $i‘ ]
                                     then
                                           $(LN) $(LNFLAGS) $i .
                                     fi ;;
                        *.fld )echo '* temporary nmake field \
                                           table'>‘basename $i‘
:                                    ;;
                        * )  ;;
                esac
        done
        FIELDTBLS=$(SET.FTBL)
        FLDTBLDIR=‘pwd‘
        export FLDTBLDIR FIELDTBLS
        $(MC) $(MCFLAGS) $(>:B:S)

CLEAN.TEMPS : .MAKE .AFTER .VIRTUAL .FORCE
        .DONE : CYAN.CLEAN

/* User defined attribute to be used to identify
 * field tables */
.IS.A.FTBL : .ATTRIBUTE

/* TOPAS field tables have .fld suffix */
.ATTRIBUTE.%.fld : .IS.A.FTBL

/* TUXEDO standard field table */
$(TUX.FTBL) : .IS.A.FTBL

/* SET.FTBL function returns the proper setting
 * for FIELDTBLS */
 SET.FTBL : .FUNCTION
        local FT FL
        for FT $(!!:A=.IS.A.FTBL)
                FL := $(FL:Y?$(FL),??)$(FT)
        end
        return $(FL)

/* rule used in :SERVER: operator rule to build
 * a TUXEDO server */
.DO.SERVER : .USE (BLDSRVR) (BLDSRVRFLAGS) (CC) \
        (CCFLAGS) (LDFLAGS) \
        ($$(<:B:S=_SERVICE)) -ltux -lfml -lPW -lgp
        : "BSRVR prerequisites are: $(!)"
        $(BLDSRVR) $(BLDSRVRFLAGS) -o $(<) -f "$(*)" \
                $($(<:B:S=_SERVICE):C/^/-s/)

/* operator to build a TUXEDO server */
":SERVER:" : .OPERATOR .MAKE
        eval
                $(<) : $(>) .DO.SERVER
                        $(@:V)
        end

/* rule to clean up the current directory */
CYAN.CLEAN : .USE .VIRTUAL .FORCE .REPEAT
        CURDIR=‘pwd‘
        if  [ ${CURDIR##*/} = obj ]
        /* equivalent to [ "‘basename $CURDIR‘" = obj ]*/
        then
                $(RM) $(RMFLAGS) *.fld *.m *.per *.c5 *.c
        fi

/* startup control */

/* not to save .old files in install directories */
clobber = 1

/* build everything and install if necessary */
.MAIN : .ALL .INSTALL

/* clean up the directory if last build \
 *wasn't a complete one */
.INIT : CYAN.CLEAN
```

Contents

Contents

Building with Alcatel-Lucent nmake

10

Viewpathing

A viewpath is a virtual directory structure that is the union of corresponding directories in parallel directory structures. The roots of these structures are called nodes.

Viewpathing allows you to modify portions of a large product without interfering with the work of other developers on the product and without duplicating the entire product in your own directory. Viewpathing allows you to develop or modify a few modules in local directories and also have access to the rest of the modules required to build a product. And, because viewpathing works for intermediate object files as well as for source files, it saves on repeated compilations.

Each person working on the product sets up a parallel directory structure, using the same directory names as those used in the baseline source directory structure. The node names are different for each developer, since each developer has a private work area. Whenever a developer wants to work on a subset of the files from the baseline source, he or she creates a *developer's view*, a parallel directory structure that is populated with copies of only those files that the developer wishes to modify (see Figure 10-1).

nmake looks for files in the virtual directory structure by searching the corresponding directory in each node for a named file. nmake then uses the first file of that name found, regardless of the time-stamp.

The order in which the files are found can be controlled by setting the viewpath (VPATH) environment variable. The VPATH variable definition uses the same syntax as the standard UNIX system PATH variable, except that null paths are not allowed. On typical platforms the VPATH can contain at least 32 nodes but platforms may vary. A warning is output when the limit is exceeded.

In the virtual directory, nmake sees the files as if the nodes named in VPATH were transparent and laid on top of one another with the rightmost node on the bottom and the leftmost on the top. Only the top file of the same name in any directory is seen and used.

nmake looks at nodes specified in VPATH as they are listed from left to right to select files needed to build the product. Files to be used must be in directories accessible down the VPATH.

All products are generated in the first node given in the VPATH variable. nmake must be invoked from within the first node of the VPATH when that variable has been set otherwise nmake will run in a non-viewpathing mode.

NOTE:
The first VPATH node must be writable by the user.

Suppose the viewpath for a product is:

VPATH = /F1/DEV:/F1/TST:/F1/OFC

nmake first searches for required files in directory /F1/DEV. If the required files are not found in /F1/DEV, nmake searches in the directory /F1/TST. If the required files are not found in F1/TST, nmake retrieves the files from the official node /F1/OFC (see Figure 10-1). The official node is the directory structure containing the baselined versions of the product software.

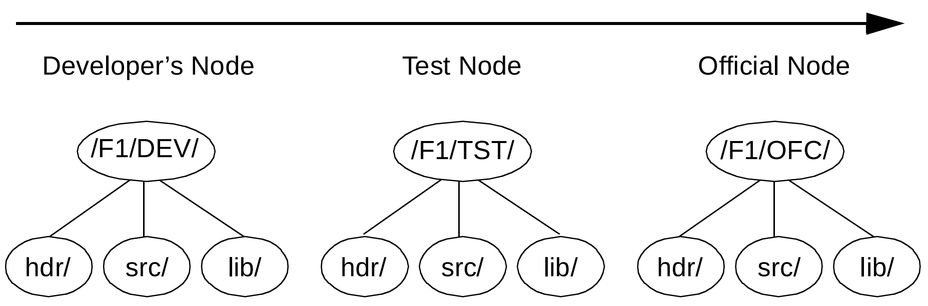


Figure 10-1. Order of Directory Search

As shown in Figure 10-1, nmake first checks the developer's node (top view), then the test node, and finally the official node (bottom view) for required files.

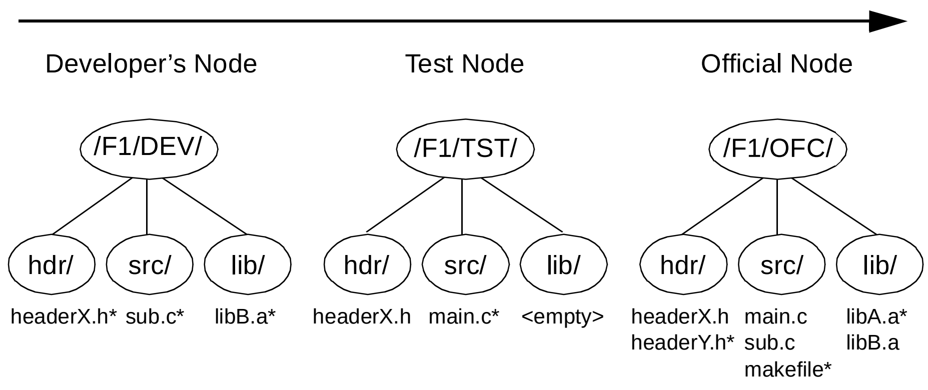


Figure 10-2. VPATH File Selection

The files that would be selected are those marked with an asterisk (*) in Figure 10-2.

The .SOURCE Special Atom can be used to specify directories that are to be included in the search algorithm. For example, if a makefile contains:

```
.SOURCE.h : dir1 dir2
```

the directories dir1 and dir2 will be searched for .h files in that order in each node specified in the VPATH variable. Note that filenames and all path specifications must be relative for viewpathing and .SOURCE features to work.

If the same file name is found in more than one directory specified in the .SOURCE Special Atom, regardless of the node, a warning message is displayed. For example, a warning message would be displayed if file1.h were found in dir1 in the official node and also in dir2 in the developer's node.

When the VPATH variable is set, nmake uses strictview as the default search algorithm. (See the *Alcatel-Lucent nmake Product Builder Reference Manual* for information about the strictview command-line option.) For example, if the VPATH variable is set to
/v2:/v1 and the .SOURCE.h Special Atom is set to./include ./include2, the order of directory search is as follows:

```
/v2   /v1   /v2/include   /v1/include   /v2/include2 \
      /v1/include2   /usr/include
```

The nostrictview command-line option could be specified to alter the search algorithm. For example, if nostrictview is specified, the order of directory search is as follows:

```
/v2  /v2/include  /v2/include2  /v1  /v1/include \
```

/v1/include2 /usr/include

Viewpathing: A Sample Session

The following session first shows how to build a product using source files that are located in several directories and then demonstrates how viewpathing can be used to build different versions of the product.

Building the Product

1. Create a directory structure in your $HOME directory similar to the one shown in Figure 10-3.

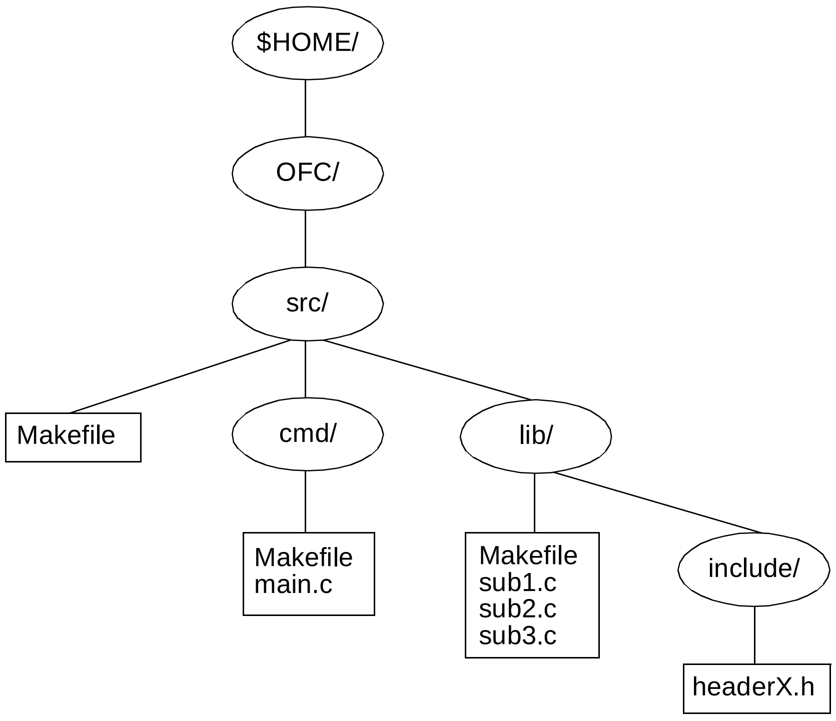


Figure 10-3. Contents of $HOME/OFC Node

In the examples shown in this chapter, $HOME was set to /home/u1/ral.

The contents of the files in the $HOME/OFC node are explained below.

Contents of $HOME/OFC/src/Makefile

```
:MAKE: lib - cmd
```

The :MAKE: operator instructs nmake to go first to the lib directory and process the makefile contained there and then to the cmd directory and process the makefile contained in that directory. The dash (-) is used to synchronize processing; in other words, nmake must complete any processing in the lib directory before processing the makefile in the cmd directory. (See the *Alcatel-Lucent nmake Product Builder Reference Manual* for more information on the - Special Atom.)

Contents of $HOME/OFC/src/lib/Makefile

```
.SOURCE.h : include
FILES = sub1.c sub2.c
libsub.a :: $(FILES)
```

The .SOURCE.h Special Atom informs nmake where header files are located, specifically, those files that contain .h suffixes. The FILES variable contains a prerequisite list of files that are to be used in the build.

The source dependency assertion builds the libsub.a library out of the list of files specified in the FILES variable. The include files do not have to be specified in the prerequisite list; nmake's scan rules automatically detect that they are prerequisites.

Contents of $HOME/OFC/src/lib/sub1.c

```
#include "headerX.h"
sub1()
{
        printf("In sub1.c, %s\n",LOC);
}
```

This subroutine prints the contents of LOC defined in headerX.h.

Contents of $HOME/OFC/src/lib/sub2.c

```
sub2()
{
        printf("In sub2.c, OFC Node\n");
}
```

This subroutine prints the string enclosed in quotes.

Contents of $HOME/OFC/src/lib/include/headerX.h

```
#define LOC "In OFC Node"
```

This header file defines the string stored in the LOC variable.

Contents of $HOME/OFC/src/cmd/Makefile

```
.SOURCE.a : ../lib
target :: main.c -lsub
```

The .SOURCE.a Special Atom tells nmake where library (.a) files are located. The source dependency assertion specifies that main.c should be compiled and then linked with libsub.a. The absolute path to the lib directory was not specified because viewpathing is to be used.

Contents of $HOME/OFC/src/cmd/main.c

```
main()
{
            sub1();
            sub2();
}
```

The contents of main.c call two subroutines, sub1() and sub2().

2. Change directories to $HOME/OFC/src by entering the following command:

```
cd $HOME/OFC/src
```

3. Run nmake:

```
$ nmake
```

nmake builds the root target (target) in the /home/u1/ral/OFC/src/cmd directory. The output from nmake is shown below.

```
lib:
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
    pp/B49DF4E0.bincc -Iinclude -I- -c sub1.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
    pp/B49DF4E0.bincc -I- -c sub2.c
+ ar r libsub.a sub1.o sub2.o
ar: creating libsub.a
+ ignore ranlib libsub.a
+ rm -f sub1.o sub2.o
cmd:
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/probe/C/\
    pp/B49DF4E0.bincc -I- -c main.c
+ cc -O -o target main.o ../lib/libsub.a
```

As shown in the output above, nmake first processes the makefile in the $HOME/OFC/src/lib directory. Two source files, sub1.c and sub2.c, are compiled, creating sub1.o and sub2.o. The library libsub.a is then created from the object modules sub1.o and sub2.o. After the makefile is processed, nmake goes to the $HOME/OFC/src/cmd directory and processes the makefile contained in that directory. The makefile instructs nmake to build the target using the source file main.c and then link the object module main.o with the library libsub.a.

The contents of the directory structure after the build are shown in Figure 10-4. Makefile.mo is the compiled makefile and Makefile.ms is the statefile; both are generated by nmake during the build.

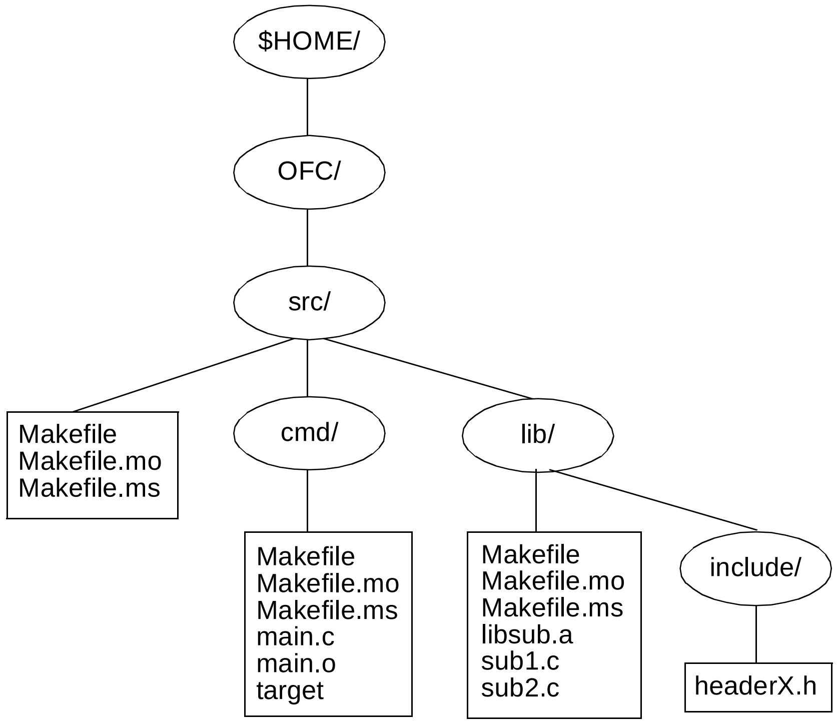


Figure 10-4. Contents of $HOME/OFC Node After Build

4. Change directories to $HOME/OFC/src/cmd by entering the following command line:

```
cd $HOME/OFC/src/cmd
```

5. Execute the target target byentering:

```
target
```

The output generated by target is shown below:

```
In sub1.c, In OFC Node
In sub2.c, OFC Node
```

As shown in the above output, the contents of the LOC variable defined in headerX.h are displayed (this is from the first statement executed in main, which called sub1()). The second statement executed in main, called sub2(), simply displays the string shown above.

Using Viewpathing

This section shows how to build a different version of the product by using viewpathing.

1. Create a directory structure in your $HOME directory similar to the one shown in Figure 9-5.

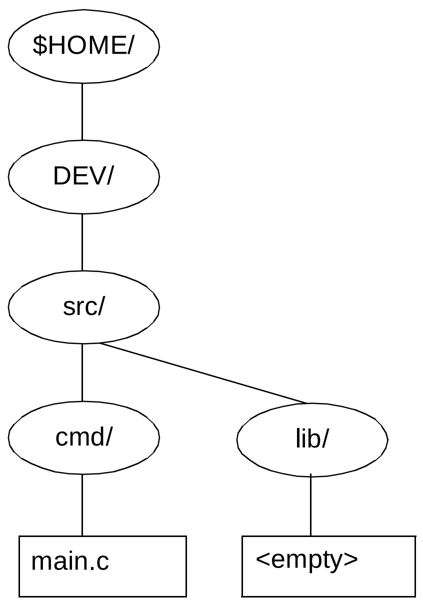


Figure 10-5. $HOME/DEV Node Directory Structure

The file contents of the $HOME/DEV node are explained below.

Contents of $HOME/DEV/src/cmd/main.c:

```
#include <stdio.h>
main()
{
printf("_IOEOF in stdio.h is %d\n",_IOEOF);
sub1();
sub2();
}
```

Note that the _IOEOF variable is defined in <stdio.h>.

2. Set the VPATH variable by entering the following command line:

   ```
   export VPATH=$HOME/DEV:$HOME/OFC
   ```

   Notice that the $HOME/DEV directory is the top view and the $HOME/OFC directory is the bottom view.

3. Change directories to $HOME/DEV/src by entering the following command line:

   ```
   cd $HOME/DEV/src
   ```

4. Run nmake:

   ```
   $ nmake
   ```

   nmake then builds the root target using the files located in the $HOME/DEV node and the $HOME/OFC node. The output from nmake is shown below.

   ```
   lib:
   cmd:
   + cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
       pp/B49DF4E0.bincc -I- -c main.c
   + cc -O -o target main.o /home/u1/ral/OFC/\
       src/lib/libsub.a
   ```

   As shown in the output above, nmake first processes the makefile in the $HOME/OFC/src/lib directory from the $HOME/DEV/src/lib directory; nothing must be updated. After the makefile is processed, nmake goes to the $HOME/DEV/src/cmd directory and processes the makefile contained in the $HOME/OFC/src/cmd directory. Since main.c is in $HOME/DEV/src/cmd, nmake builds the target again using this source file and then links the object module main.o with the library libsub.a in $HOME/OFC/src/lib.

5. Change directories to $HOME/DEV/src/cmd by entering the following command line:

   ```
   cd $HOME/DEV/src/cmd
   ```

6. Execute target by entering the following command:

   ```
   target
   ```

   The output generated by target is shown below:

   ```
   _IOEOF in stdio.h is 16
   In sub1.c, In OFC Node
   In sub2.c, OFC Node
   ```

   As shown above, the contents of the _IOEOF variable defined in stdio.h is displayed (this was the first statement executed in main). The second statement executed in main called sub1(). The subroutine sub1() is in sub1.c in the official node (bottom view). The third statement executed in main called sub2(). The subroutine sub2() is in sub2.c also in the official node (bottom view).

The directory contents after the build are shown in Figure 10-6. Since Makefile is not changed (it is up-to-date), there is no Makefile.mo generated in the $HOME/DEV node.

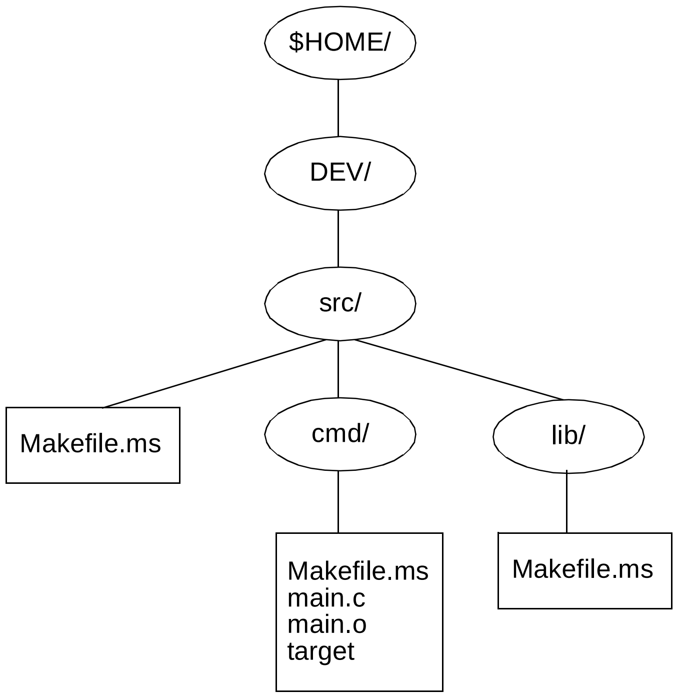


Figure 10-6. Contents of $HOME/DEV Node After Build

7. Create a subdirectory called include under the $HOME/DEV/src/lib directory as shown in Figure 10-7.

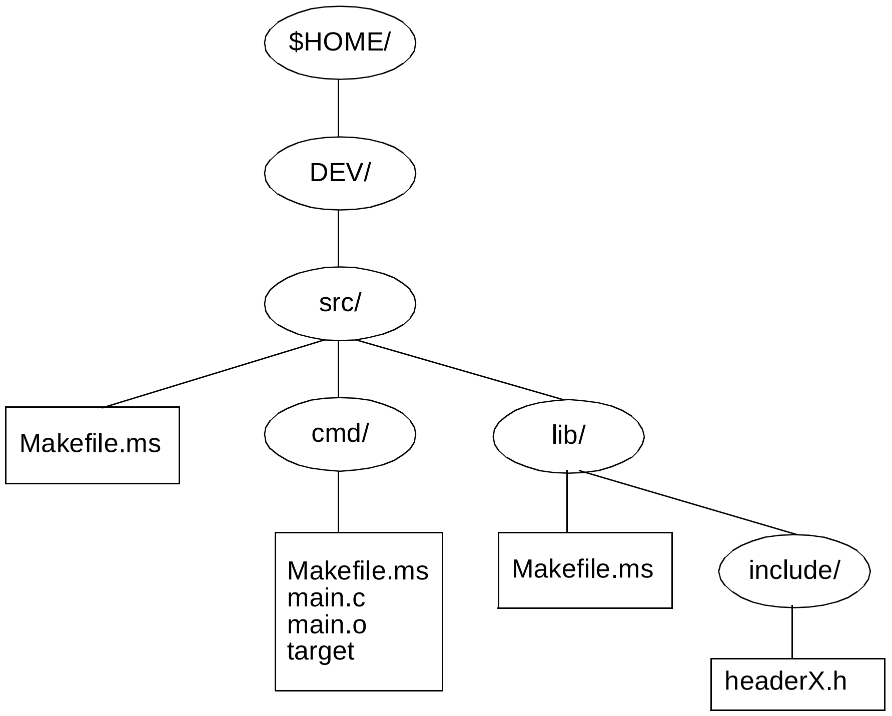


Figure 10-7. Contents of $HOME/DEV Node

8. Create a file called headerX.h that has the following contents:

   ```
   #define LOC "In DEV Node"
   ```

9. Change directories to $HOME/DEV/src by entering the following command line:

   ```
   cd $HOME/DEV/src
   ```

10. Run nmake:

    ```
    $ nmake
    ```

    nmake then builds the target target in the $HOME/DEV/src/cmd directory.

    The directory contents after the build are shown in Figure 10-8.

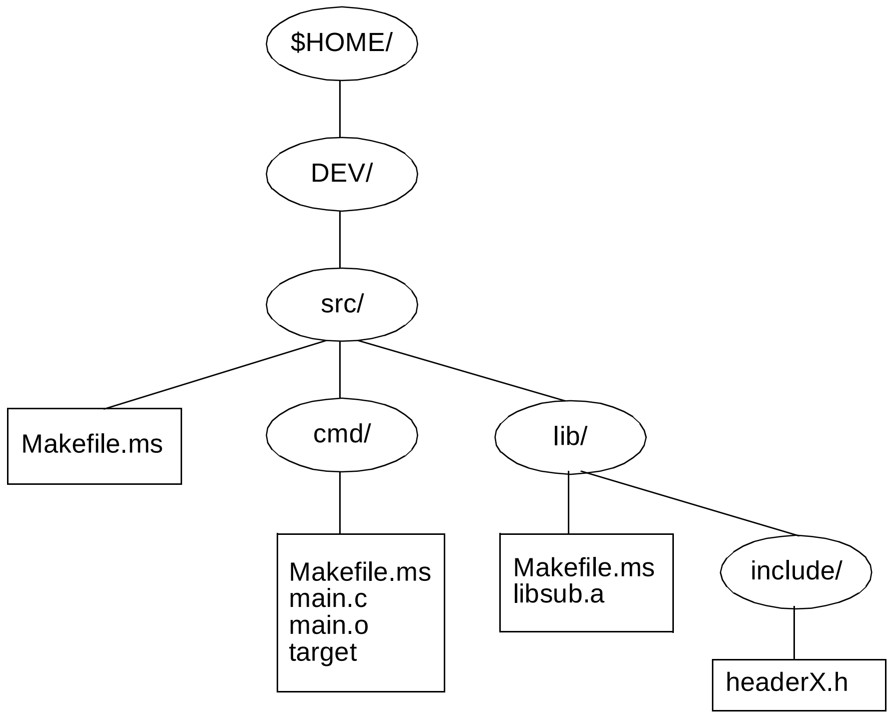


Figure 10-8. Contents of $HOME/DEV Node After Build

The output from nmake is shown below:

```
lib:
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -Iinclude -I- -c /home/u1/\
ral/OFC/src/lib/sub1.c
+ cp /home/u1/ral/OFC/src/lib/libsub.a libsub.a
+ ar r libsub.a sub1.o
+ ignore ranlib libsub.a
+ rm -f sub1.o
cmd:
+ cc -O -o target main.o ../lib/libsub.a
```

nmake reads the first valid compiled makefile (.mo) in $VPATH and also reads all the statefiles from every view in the viewpath, in this case, both the top view and the bottom view.

nmake stores all the information on a target in the statefile (.ms), in the same view in which the target is built; therefore, when nmake builds in a viewpath environment, it needs access to all statefiles in the different views of the viewpath. Of course, this setup allows you to have a different set of variable values than that of the official (bottom) view.

11. Change directories to $HOME/DEV/src/cmd by entering the following command:

```
cd $HOME/DEV/src/cmd
```

12. Execute target by entering the following command:

```
target
```

The output generated by target is shown below.

```
_IOEOF in stdio.h is 16
In sub1.c, In DEV Node
In sub2.c, OFC Node
```

As shown above, the contents of the _IOEOF variable defined in stdio.h are displayed (this was the first statement executed in main). The second statement executed in main called sub1(). The subroutine sub1() is in sub1.c in the official node (bottom view); however, its included header file, headerX.h, that defines LOC is in the developer's node (top view). The third statement executed in main called sub2(). The subroutine sub2() is in sub2.c, which is also in the official node (bottom view).

Using coshell with nmake

coshell is the network shell coprocess server. It can establish shell coprocesses to machines producing compatible executables on the local area network and send user jobs to these shell coprocesses. This section describes the use of coshell with nmake to build a product. Some introductory information is presented first. A sample session follows that further describes the introductory material. Before reading the rest of this section, please read the section *The Shell Interface* in Chapter 1, *Overview of Alcatel-Lucent nmake*. Additional information regarding coshell can also be found in the coshell manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual*.

coshell requires at least a working UNIX system and Transmission Control Protocol (TCP). A UNIX system domain socket or mounted pipes/streams (fattach()) are preferred. Ask your system administrator if your machine meets these requirements.

If coshell is not present in $nmake_install_root/bin, where $nmake_install_root is the directory under which the nmake software package is installed, coshell is not supported for your environment.

coshell Overview

There is one coshell server per user. This server runs as a daemon on the user's home host, and only processes running on the home host have access to the server. The server controls a background ksh(1) shell process, initiated by rsh(1) or remsh(1), on each of the connected hosts. The environment of the local host shell is inherited from the server, whereas the environment of remote shells is initialized by .profile and $ENV. The shells run with the ksh, bgnice, and monitor options on. The environment variables set or changed after the coshell server has been started can be passed to shells using the COEXPORT environment variable. For example, if PS4 and CHOME are changed to have the following values:

```
export PS4='+co'
export CHOME=$HOME/c++
```

the export COEXPORT=PS4:CHOME command causes these changes to take effect on the other shells.

Job requests are accepted from user processes on the local host and are executed on the connected hosts. stdout, stderr, FPATH, NPROC (the number of jobs run concurrently), PWD, PATH, VPATH, vpath, umask, and the environment variables listed in COEXPORT are set to match requesting user values. stdin is set to /dev/null; coshell does not directly support interactive jobs. Job scheduling is based on load and idle time information generated by the ss(1) system status daemon. This information is updated every 60 seconds on average.

Prerequisites for Using coshell with nmake

The following are prerequisites for using coshell with nmake.

- Good Network Timekeeper

  All clocks must be kept in sync across the network. The cron command can be used, but may not be good enough. The Network Time Protocol (NTP) is available on Sun Workstations.

- File System

  The file systems that are needed for nmake must be mounted on all machines in the network that coshell accesses.

NOTE:
The share directory of the distributed nmake software package must be the same for all coshell machines.

- Logins

  All users of coshell must be able to use rlogin to log in from the machine that starts coshell to machines accessed by coshell, without having to enter the login and password. Ideally, each user's home directory should be shared across the coshell network (i.e., the home file system should also be mounted across the network). If rlogin fails, the targeted machine is not used by coshell.

- ksh

  ksh version 88i (or a later version) must be included in PATH to set up communications between the host machine and other machines in the network; what $(whence ksh) should return this version as its output.

- Lock Screen

  If you want to lock the screen, use a lock-screen program that blocks on a read request and does not inflate the load average. Otherwise, workstations will always look busy to the coshell server.

- .profile and $ENV

  Interactive queries within .profile and $ENV must be disabled for noninteractive shells; otherwise, coshell will fail:

```
case $- in
        *i*)
                query
                ...
        ;;
esac
```

The System Status Daemon

$nmake_install_root/lib/ssd runs on all the coshell machines in the network. This daemon records CPU state/usage information for use by coshell and ss. The daemon is automatically initiated on the machine when information is first requested of that machine.

The CPU state/usage information for each coshell machine is compressed into 8 bytes and stored in the time-stamp field of a file whose name is the same as the machine name under the $nmake_install_root/share/lib/ss directory. This information is updated every minute by ssd.

The ssd must be owned by the same user as the owner of the share directory tree ($nmake_install_root/share). In the distributed nmake software package, ssd has the setuid and setgid bits on. If the permissions are changed, ssd will not operate properly. For more efficient operation, the group ID of ssd can be changed to *kmem*; however, this change may require root privileges.

Setup

To use the coshell tool, a local host attribute file must be created and certain environment variables must be set.

Local Host Attribute File

The local host attribute file contains the names of all the machines that are available in the local area network along with some other pertinent information. The file local.ast provided with the distributed nmake software package (located in $nmake_install_root/share/lib/cs) is an example.

Two shell scripts, genlocal and genshare, are provided with the distributed nmake software package; they are located in $nmake_install_root/bin and can be used to generate the local host attribute file. The local host attribute file is shared by all the coshell users. It should be updated as hosts are added to or deleted from the local network.

The genshare script is run first to generate information on servers for the network. By default, this information is stored in $nmake_install_root/lib/cs/share. After this information has been stored, the genlocal script is run to generate the local host attribute file. By default, this information is stored in $nmake_install_root/share/lib/cs/local.

If the share file generated by the genshare script is not stored in the default path, you must pass its path to the genlocal script using the -f option. For example,

genlocal -f *path_to_***share***_file*

You can modify the generated files to meet your needs.

The sample session in this chapter examines the contents of the local host attribute file.

Environment Variables

The COSHELL environment variable must be set to coshell in order to use the coshell server. If the COSHELL environment variable is not set, nmake will use a Bourne-based shell (either sh or ksh) and builds will be performed on a single machine.

The NPROC environment variable must be set to the desired number of concurrent jobs. This setting is equivalent to using the jobs command-line option. However, the jobs command-line option overrides the NPROC environment variable. The value of NPROC can be set to a high number. The actual number of concurrent jobs may be less, based on the values of the percpu, peruser, perserver, and maxload coshell global attributes. (See the sample session below and the coshell manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual* for details about attributes).

The SHELL environment variable must be set to the ksh 11/16/88i (or later) version.

The following environment variables must be set if coshell is installed in a non-standard directory (other than /bin, /usr/bin, or /usr/local/bin):

root=*coshell_installation_root_directory*
export PATH=$root/bin:$PATH

If coshell is dynamically linked, the environment variable for locating the shared library (e.g., LD_LIBRARY_PATH) must include $root/lib:

export LD_LIBRARY_PATH=$root/lib:$LD_LIBRARY_PATH

The shell function called cosh is provided with the distributed Alcatel-Lucent nmake software package and located in $nmake_install_root/fun for the purpose of starting up the coshell server. This function also adds the word *coshell* to the title bar of the window in which the coshell server is started.

If cosh is used, FPATH must be set, e.g.:

export FPATH=$root/fun:$FPATH

When coshell is killed by cosh -Q, the output of the date and the coshell -t -sl -Q command is appended to $HOME/.cosh.

NOTE:
If you have trouble starting up coshell due to bugs as a result of /tmp implementation, set and export CS_MOUNT_LOCAL to a temporary directory on a disk file system where all the users have read and write permissions.

The following is a list of environment variables that are used by coshell:

COATTRIBUTES	Host attribute expression, (*type@local*) by default. Non-numeric-valued attributes can appear as the first operand of the comparison operators <, <=, ==, !=, >=, and >, where the second operand must be a "..." or '...' string that is compared with the attribute value. For the == and != operators, the second operand is taken to be a ksh file-match pattern. For example, given the host definitions:

```
coot     type=sun4    mem=8m    rating=11.0     cad
dodo     type=sun3    mem=4m    rating=2.0
loon     type=mips    mem=16m   rating=20.0
```

(type=='sun*'&&mem>6m) selects coot
(rating>=11.0) selects coot and loon
(cad) selects coot.

attribute@host represents the attribute value for *host*. For example, *type@local* matches the type of the host running the coshell server

COEXPORT	A colon-separated list of environment variables to export to each job. This variable supports the rare cases where some environment variables change after the coshell server has been started. For example, some commands use environment variables rather than arguments or options to pass input data.
COSHELL	Set to coshell for the network shell service.
COTEMP	Set to a different value for each shell command. It is used for temporary file names (see *Engine Variables* on the nmake manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual*). This variable can be referenced in .profile.
HOMEHOST	Set within each action in the action block of nmake assertions to the name of the host executing coshell.
HOSTNAME	Set within each action in the action block to the name of the host executing the action. This variable can be referenced in .profile.
HOSTTYPE	Set within each action in the action block to the type (from the local coshell host attribute file) of the host executing the action. This variable can be referenced in .profile.
NPROC	Default command concurrency level.

coshell: A Sample Session

This section provides a sample session using coshell with nmake. The sample session is intended to familiarize the user with coshell and also demonstrate some of the capabilities provided by coshell. In the sample session, a product is built using both coshell and nmake. The sample session should be run in a windowing environment so that job distribution can be monitored.

NOTE:
Make sure that all the prerequisites for running coshell listed in the section *Prerequisites for using coshell with nmake*, above, are met before you start the sample session.

1. Set the following environment variables and export them:

```
root=coshell_installation_root_directory
export PATH=$root/bin:$PATH
export FPATH=$root/fun:$FPATH
export SHELL=path_to_88i_ksh_or_later_version
export LD_LIBRARY_PATH=$root/lib:$LD_LIBRARY_PATH
```

2. If $root/lib/cs/share and $root/share/lib/cs/local exist, you can skip this step.

   Execute the following two commands in the order given to generate the local host attribute file, which contains a list of all the machines that can be accessed in the local area network.

```
genshare > $root/lib/cs/share
genlocal > $root/share/lib/cs/local
```

   If you encounter write-permission problems, contact your system administrator.

3. Modify the local file by entering any new information for the local network. You may need help from your system administrator if you don't have write permission for the local file. The local host attribute file used in this sample session is shown below.

```
#
# Modified local host attributes
#
local       busy=3m         pool=8        bias=4
erato22     type=386        rating=17.5   idle=15m
calliope    type=hp.ux      rating=33.5   idle=15m
sparta      type=ibm.risc   rating=13.5   idle=15m
circe9      type=sgi.mips   rating=24.0   idle=15m
athena      type=sol.sun4   rating=10.0   idle=15m
joven       type=sol.sun4   rating=11.0   idle=15m
hawaii      type=sol.sun4   rating=12.0   idle=15m
banji       type=sol.sun4   rating=12.0   idle=15m
aramis4     type=sun3       rating=3.x    idle=15m
chorus      type=sun4       rating=10.0   idle=15m
newton      type=sun4       rating=10.0   idle=15m
tiger       type=sun4       rating=16.5   idle=15m
dime        type=sun4       rating=20.0   idle=15m
wilma       type=sun4       rating=22.0   idle=15m
coqui       type=sun4       rating=22.0   idle=15m
peipu       type=sun4       rating=22.0   idle=15m
bugs        type=sun4       rating=22.0   idle=15m
dwarfs      type=sun4       rating=22.5   idle=15m
haydn       type=sun4       rating=22.5   cpu=4
```

```
khush           type=sun4       rating=23.5     idle=15m
```

The following is an explanation of the attributes used in the local host attribute file local. Some of the attributes are global and are used to control coshell. Global attributes are not associated with any specific coshell machine. You can associate project-specific attributes with any machine by editing the local file or by using the coshell -a *machine*,*attributes*... command. Attribute names must match [a-zA-Z_][a-zA-Z_0-9]*. See the coshell manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual* for more details.

- local

  local specifies the name of the host that starts the coshell server. For example, if one were to log in to one of the machines listed in the local host attribute file, for example *chorus*, and start the coshell server by executing cosh or coshell +, *chorus* would be considered the local machine. The local machine has the attribute bias=4 and the attributes associated with the *chorus* entry (i.e., type=sun4, rating=10.0, and idle=15m). The attributes pool=8 and busy=3m are global attributes and are not associated with any specific machine.

- pool=8

  The pool=8 entry indicates that there are eight central processing units in the processor pool. The pool entry is a global attribute.

- bias=4

  Hosts with a high bias are (linearly) least likely to be scheduled for job execution (the default bias is 1.00). Therefore, the local host is least likely to be chosen for job execution because it has the highest bias of all the hosts. The other hosts get the default.

- busy=3m

  The busy entry is a global attribute. The busy=3m entry specifies that 3 minutes is the grace period for jobs running on busy hosts. A host is busy when its idle attribute is nonzero and its minimum logged-in user idle time is less than the value of busy. For a job running on an idle host, busy is the maximum amount of time the job can run after the host becomes busy. If the job does not finish within busy, the SIGSTOP signal is sent to the job, and the job stops. When the host idle time exceeds busy, the SIGCONT signal is sent to the job and the job resumes. The meaningful unit of time can be m(inute) or h(our).

- type=sun4

  This entry specifies that the host type is sun4. The entry for this field is usually related to the object and executable attributes. Any name can be specified as the machine type. coshell performs a string comparison when searching for all machine types that are the same as the local machine's type. The default type is the wild card character *. A machine with this type (*) would always be selected for job processing.

- rating=10.0

  This entry specifies that the host rating is 10 million machine instructions per second (MIPS). The entry for this field is usually in network relative MIPS.

- idle=15m

  This entry specifies that the minimum logged-in user idle time before jobs can be scheduled on the host is 15 minutes.

  Compute servers that are available to all users usually have no idle attribute, whereas personal workstations are given at least idle=15m out of courtesy to the workstation owner.

4. The ss command in $nmake_install_root/bin (see the ss manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual*) can be used to list the system status of all the hosts on the network. Run the following command line:

   ss

   Output similar to the following will be displayed.

```
tiger         up    3w02d,   4   users,  idle  3h36m,   load  0.32,  %usr  0,  %sys  0
wilma         up    3d04h,   3   users,  idle  53m36s,  load  0.08,  %usr  0,  %sys  0
coqui         up    3d04h,   5   users,  idle  2.00s,   load  0.00,  %usr  0,  %sys  0
circe9        down  3m41s,   0   users,  idle  0,       load  0.00,  %usr  0,  %sys  0
chorus        up    3d03h,   5   users,  idle  25.00s,  load  0.00,  %usr  0,  %sys  0
dwarfs        up    3d04h,   3   users,  idle  21m51s,  load  0.00,  %usr  0,  %sys  0
erato22       down  3m45s,   0   users,  idle  0,       load  0.00,  %usr  0,  %sys  0
khush         up    3d03h,   0   users,  idle  %,       load  0.24,  %usr  0,  %sys  0
joven         up    10h54m,  86  users,  idle  1m26s,   load  3.52,  %usr  0,  %sys  204
dime          up    6d17h,   2   users,  idle  1h23m,   load  2.16,  %usr  0,  %sys  0
aramis4       down  3m34s,   0   users,  idle  0,       load  0.00,  %usr  0,  %sys  0
peipu         up    3d03h,   5   users,  idle  25.00s,  load  0.00,  %usr  0,  %sys  0
bugs          up    3d02h,   3   users,  idle  5.00s,   load  0.00,  %usr  0,  %sys  0
calliope      down  3m44s,   0   users,  idle  0,       load  0.00,  %usr  0,  %sys  0
haydn         up    3d10h,   76  users,  idle  1.00s,   load  1.12,  %usr  0,  %sys  0
hawaii        up    10h54m,  89  users,        1m29s,   load  3.52,  %usr  0,  %sys  204
spartacirce9  down  3m42s,   0   users,  idle  0,       load  0.00,  %usr  0,  %sys  0
banji         up    10h54m,  89  users,  idle  1m28s,   load  3.52,  %usr  0,  %sys  204
newton        up    3M00w,   6   users,  idle  11.00s,  load  1.68,  %usr  0,  %sys  0
athena        up    10h54m,  85  users,  idle  1m25s,   load  3.52,  %usr  0   %sys  204
```

The fields that are displayed are *host name*, *up time*, *active users*, *idle time*, *load*, *% user time*, and *% sys time*. The following table lists the fields and explains them.

host name	The host name as known on the local network.
up time	The time since the host was last booted. Down time is noted by *down*.
active users	The number of logged in users that have been active within the last 24 hours.
idle time	The time since any user last typed or used the mouse.
load	The load average, averaged over all connected hosts.
%usr time	The percentage of CPU time used by user and low-priority processes.
%sys time	The percentage of CPU time used by the system itself. The CPU idle time is (100−%usr−%sys).

With the ss command, if the host arguments are specified, the listing is restricted to those hosts. If no options are specified, the listing is sorted by host name. Any other sorting is done in order from best to worst, i.e., the best load averages are small, the best idle times are large. The option order that is specified determines the sort order from highest to lowest precedence. The ss options are:

-c	Sort by %usr and %sys CPU utilization.
-i	Sort by idle time.
-l	Sort by load average.
-r	Reverse the sort ordering to worst to best.
-t	Sort by uptime.
-u	Sort by number of active users.

A system will be reported as being down until ssd is up and running (see the section *The System Status Daemon*, above). The initial report is also subject to network file system latencies.

5. The shell function cosh is used to start coshell. Execute cosh by entering the following command line.

```
cosh
```

The following message will be displayed:

```
coshell 9.1.2.1 (Alcatel-Lucent Bell Labs) 06/30/06
```

If you don’t get this output, verify that you are running ksh version 88i or later and that your SHELL environment variable is set to that version of ksh. If this information is correct, set and export the environment variable CS_MOUNT_LOCAL to a directory other than /tmp where all the coshell users have read and write permissions; then execute cosh again.

6. Open another window and then enter the following command line to monitor the job distribution (recall that there is one coshell server per user):

```
$ coshell -
```

In interactive mode, the prompt has changed to the following:

```
coshell>
```

7. At the coshell> prompt, enter the following command line to obtain a list of commands that can be used.

```
coshell> h
```

The following help message will be displayed:

```
a host ... change host shell attributes
c host ... close host shell
d [level] set debug trace stderr [and level]
f [fd] internal fd status [drop CON fd]
g global state
h help
j job status
k [ckst] job terminate [continue|kill|stop|term] job
l host list matching host names
o host ... open host shell
q quit
r host [cmd] run cmd [hostname] on host
s [aelopst] shell status [attr|every|long|open|pid|sched|temp]
t shell and user connection totals
u user connection status
v server version
Q kill server and quit
```

The g, j, q, ss, and Q coshell interactive commands will be used in the remainder of this sample session. All interactive commands can be used on the coshell command line.

8. The g command can be used to list global attributes and their values. Enter g at the coshell prompt to obtain the global status.

```
coshell> g
```

The output resembles the following:

```
coshell 9.1.2.1 (Alcatel-Lucent Bell Labs) 06/30/06

mesg      /dev/tcp/135.3.118.2/50745
profile   { ../.profile; eval test -f \$ENV \&\& . \$ENV; } \
            >/dev/null 2>&1 </dev/null
pump        /dev/tcp/135.3.118.2/50746
service     /dev/fdp/local/coshell/user
sh          /usr/bin/ksh

access55m15s    fdtotal  64       override 0       shells    1
busy    2m00s   grace    2m00s    percpu   2       shellwait 0
clock   4m45s   joblimit 0        perserver32      sys       0
cmds    2       jobs     0        peruser  10      user      0
connect 0       jobwait  0        pool     8       users     2
debug   0       maxload  0.00     real     0       wakeup    0
disable 0       open     1        running  0
```

9. Return to the window where cosh was invoked; change directories to where your source files reside.

10. Run nmake.

    ```
    $ nmake
    ```

    The nmake command must be executed in the window where the coshell server was started so that the server can distribute jobs to other machines in the LAN.

11. The j command can be used to list the status of all jobs at any instance of time. Enter j at the coshell> prompt in the job-monitoring window to obtain the current job status:

    ```
    coshell> j
    ```

    Output similar to the following will be displayed. Two jobs have been scheduled at this time.

    ```
    coshell> j
    JOB USR RID     PID TIME HOST         LABEL
       1   12     1 12542      0 tiger      make x1.o
       2   12     2 QUEUE      0*khush      make x2.o
    ```

    Each field is explained below.

JOB	The ID assigned to the job by the server. This number may be used as an argument to the k command (see the coshell man page for details).
USR	The ID assigned to the requesting user by the server.
RID	The ID assigned to the job by the requesting user.
PID	The job process ID; QUEUE if the shall is in the process of opening, START if the PID has not been determined yet, and WARPED if the job completed before its PID was determined.

TIME	The elapsed time since the job started. * follows the time if the job is about to terminate.
HOST	The host where the job is running. The most recent signal sent to the job follows the host name.
LABEL	The label assigned to the job by the requesting user.

12. At the coshell> prompt, enter the j command to list the status of all jobs.

```
coshell> j
```

Output similar to the following will be displayed. Five jobs have been scheduled at this time.

```
JOB USR RID     PID TIME   HOST   LABEL
 2    12    2 START 3.x0s khush make x2.o
 3    12    3 12549 1.00s tiger make x3.o
 4    12    4 START 1.00s khush make y1.o
 5    12    5 12551 1.00s tiger make y2.o
 6    12    6 12552 1.00s tiger make y3.o
```

No more than three jobs are distributed to *tiger* because percpu has 3 set as the maximum number of jobs running on each CPU (see the output of the g command in Step 8).

If jobs are distributed only to the machine that starts the coshell server, check the load and idle information on other machines with the ss command; also check the prerequisites listed in the section *Prerequisites for using coshell with nmake*, above.

13. At the coshell> prompt, enter the j command to list the status of all jobs.

```
coshell> j
```

Output similar to the following will be displayed; two jobs are still running.

```
JOB USR RID     PID   TIME HOST   LABEL
 7    12    4 12570 1.00s tiger make x
 8    12    6 7055   1.00s khush make y
```

14. You can change the global attribute status at the coshell prompt:

```
coshell> a local,percpu=6
```

The maximum number of jobs allowed to run on each CPU is changed to six. This change can be verified:

```
coshell> g

coshell 9.1.2.1 (Alcatel-Lucent Bell Labs) 06/30/06

mesg      /dev/tcp/135.3.118.2/50745
profile   { ../.profile; eval test -f \$ENV \&\& . \$ENV; } \
          >/dev/null 2>&1 </dev/null
pump      /dev/tcp/135.3.118.2/50746
service   /dev/fdp/local/coshell/user
sh        /usr/bin/ksh

access55m15s   fdtotal  64      override 0      shells    1
busy    2m00s  grace    2m00s   percpu   6      shellwait 0
clock   4m45s  joblimit 0       perserver32     sys       0
cmds    2      jobs     0       peruser  10     user      0
connect 0      jobwait  0       pool     8      users     2
debug   0      maxload  0.00    real     0      wakeup    0
disable 0      open     1       running  0
```

Similar changes can be made to other global attributes. For example, you can set NPROC to a high number, then use the interactive commands to tune the global variable values to meet your environmental requirements.

15. The ss coshell interactive command can be used to list the shell scheduling status. This command is used primarily for debugging. The output is different from the ss command. Please refer to the coshell manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual* for a description of each field. At the coshell> prompt, enter the following command line:

```
coshell> ss
```

Output similar to the following will be displayed.

```
CON  OPEN  USERS  UP      CONNECTUPDATEOVERRIDE IDLE    TEMP    RANK      HOST
7    1     17     3d07h   1h14m   37.00s  0/1    0       325.12  1572.49   tiger
10   1     5      22h51m  23.x0s  37.00s  0/0    15m00s  325.12  1862.96   khush
-    1     12             5d08h   37.00s  0/1    15m00s  325.12  1924.39   peipu
-    0     7              1w06d   15m37s  0/0    15m00s  325.12  2400.77   coqui
-    0     0              DOWN    2m07s   0/0    15m00s  25.12   30000.19  newton
coshell>
```

16. At the coshell> prompt, enter the j command to list the status of all jobs.

```
coshell> j
```

Since there are no more jobs that must be scheduled, only the coshell> prompt is displayed, as shown below:

```
coshell>
```

17. At the coshell> prompt, enter the q command to stop monitoring coshell actions.

```
coshell> q
```

If q is entered, your system prompt will be displayed. The coshell server remains running and scheduling jobs.

18. Use the cosh -Q command to kill the coshell server. At the prompt, enter the following command line:

```
cosh -Q
```

This command generates the .cosh file in your home directory. You can examine the contents of this file.

You may also enter the Q command at the coshell> prompt to kill the coshell server, but this does not unset the environment variables (e.g., COSHELL, NPROC) set by the shell function cosh or by you.

The coshell server is usually left running, since it is implemented as a daemon.

19. Now change directory to $nmake_install_root/share/lib/ss and enter the following command:

```
ls -l *
```

Output similar to the following will be displayed:

```
---------- 1 nmake              0 Dec 31 1969 coqui
---------- 1 nmake              0 Dec 31 1969 khush
---------- 1 nmake              0 Jan 24 1971 tiger
---------- 1 nmake              0 Dec 31 1969 newton
```

The time-stamp field of each file is unexpected. This field is actually the compressed CPU state/usage information for the host of the same filename and is updated every minute by the ssd running on each machine.

20. If you have the nmake :MAKE: assertion operator for implicit recursive make in your makefile, you must set the base rule variable recurse to an appropriate number *n* for coshell to distribute jobs from *n* directories at a time. recurse is set to 1 by default.

If you want a target to be built on the machine that starts the coshell server, you can used the .LOCAL Special Atom with your assertion:

```
a :: a1.c a2.c .LOCAL
```

coshell distributes a1.c and a2.c compilation work to other machines and links all the .o files to create the target on the local machine.

The COATTRIBUTES variable can be used to select machines for job distribution. For example:

```
export COATTRIBUTES="(type==’sun3|sun4’)"
```

can be used when you build the system for the target machines using a cross-compiler. You can also use this variable in the prerequisite list so that actions in the assertion can be executed on a specific machine. For example:

```
x.o : COATTRIBUTES="(name==’dodo’)"
```

x.o is compiled on the machine *dodo*.

CAUTION:
Never put coshell -r *in makefiles;* coshell *does not talk to daemons on other machines. (See the* coshell *manual page in the Alcatel-Lucent nmake Product Builder Reference Manual for details.)*

Examples of nmake Usage

The following pages contain a wide range of examples of nmake usage. The examples are intended both to suggest the types of builds that are possible with nmake and to demonstrate the techniques that can be used in the various build types. Each example shows the makefile source, the command that is run, and the output of the command. Explanatory comments call attention to the signficant features of each example.

Building an Executable from a Single C Source File

When the source dependency assertion operator (::) is used to create an executable, it invokes the proper default metarules and operators from the default base rules file to create an executable from the given source file(s). In this example, which builds foo from the file specified in the SRC variable, the metarule %.o : %.c is invoked to create the .o from the .c file and the operator .COMMAND.o is invoked to create the executable from the .o file.

Contents of Makefile

```
SRC = foo.c
foo :: $(SRC)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c foo.c
+ cc -O -o foo foo.o
```

Building and Installing an Executable from Multiple C Source Files

The installation of an executable can be handled in one of three ways:

- If neither :INSTALLDIR: nor :INSTALL: is specified, the target is installed into $(BINDIR) by default (assuming the source dependency assertion operator (::) is being used). Similarly, libraries default to $(LIBDIR).

- The :INSTALLDIR: operator can be used to install a prerequisite file into a directory while preserving the prerequisite's filename. It can also be used to disable installs for a file by specifying a null target on the left-hand side.
- The :INSTALL: operator can be used to install a file while specifying a new filename for the installed file.

Installing in the Default Directory

The common action install must be specified on the nmake command line for installation to occur (i.e., nmake install). The value of $(BINDIR) will default to $HOME/bin if $(BINDIR) or $(INSTALLROOT) is not specifically set. This default install operation is the same as using $(BINDIR) :INSTALLDIR: foo

The following makefile builds foo and installs it into directory $(BINDIR). Running nmake on the file generates foo, while running nmake install generates foo and installs it in $(BINDIR).

Contents of Makefile

```
BINDIR = $(VROOT)/bin
SRC = foo.c bar.c orb.c
foo :: $(SRC)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c foo.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c bar.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c orb.o
+ cc -O -o foo foo.o bar.o orb.o
```

Run nmake again

```
$ nmake install
```

Output

```
+ ignore cp foo ../bin/foo
```

Installing in a User-Defined Directory

The common action install must be specified on the nmake command line for installation to occur (i.e., nmake install). The :INSTALLDIR: operation replaces the default install operation caused by the source dependency assertion operator.

Example 1 - Building and Installing

The following makefile generates foo and installs it into directory $(UTILDIR).

Contents of Makefile

```
UTILDIR = $(VROOT)/util
SRC = foo.c bar.c orb.c
$(UTILDIR) :INSTALLDIR: foo
foo :: $(SRC)
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c foo.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c bar.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c orb.o
+ cc -O -o foo foo.o bar.o orb.o
+ ignore cp foo ../util/foo
```

Example 2 - Turning off the Install Operation

The following makefile generates foo and disables the install operation for foo. Lack of a target on the left-hand side of :INSTALLDIR: turns off the install operation for the prerequisite.

Contents of Makefile

```
:INSTALLDIR: foo
SRC = foo.c bar.c orb.c
foo :: $(SRC)
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c foo.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c bar.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c orb.o
+ cc -O -o foo foo.o bar.o orb.o
```

Installing with a Different Name

The common action install must be specified on the nmake command line for installation to occur (i.e., nmake install). Use :INSTALL: if you want to name the installed file differently than its original name.

:INSTALLDIR: vs. :INSTALL: :INSTALLDIR: takes a directory on the left-hand side, while :INSTALL: takes a directory/filename for which the file being installed will be named.

Example 1

The following makefile builds foo and installs it twice, once in $(BINDIR) with the filename foobar, and again as foo. The second installation is a result of the default installation action associated with the :: assertion as in Method 1 above.

Contents of Makefile

```
BINDIR = $(VROOT)/bin
SRC = foo.c bar.c orb.c
$(BINDIR)/foobar :INSTALL: foo
foo :: $(SRC)
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c foo.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c bar.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c orb.o
+ cc -O -o foo foo.o bar.o orb.o
+ ignore cp foo ../bin/foobar
+ ignore cp foo ../bin/foo
```

Example 2

The following makefile builds foo and installs it once in $(BINDIR) with the filename foobar. Notice that the placement of :INSTALL:, :INSTALLDIR:, and the source dependency assertion is important in the makefile. They must appear in the order shown, or foo will be installed twice, as above.

Contents of Makefile

```
BINDIR = $(VROOT)/bin
SRC = foo.c bar.c orb.c
$(BINDIR)/foobar :INSTALL: foo
:INSTALLDIR: foo
foo :: $(SRC)
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c foo.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c bar.o
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c orb.o
+ cc -O -o foo foo.o bar.o orb.o
+ ignore cp foo ../bin/foobar
```

Building an Executable from Multiple C Source Files Containing #define

This example illustrates the building of an executable from multiple C sourcefiles (during which the propagation of implicit state variables occurs automatically) in conjunction with the -D option required by the C preprocessor.

The makefile below contains an example for each case.

Contents of Makefile

```
FILE1_CONSTANT == 50
FILE2_IFDEF ==
FILE3_IFDEF == 1

(FILE3_IFDEF) : .PARAMETER

target1 :: file1.c file2.c file3.c
```

Contents of file1.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
        printf("hello from %s, constant is %d\n", __FILE__, FILE1_CONSTANT);
        file2();
        file3();
        exit(0);
}
```

Contents of file2.c

```
#include <stdio.h>
#ifdef FILE2_IFDEF
#define    HELLO_MSG "hello with ifdef"
#else
#define    HELLO_MSG “hello”
```

```
#endif

void file2(void)
{
        printf("%s from %s\n", HELLO_MSG, __FILE__);
}
```

Contents of file3.c

```
#include <stdio.h>
#ifdef FILE3_IFDEF
#define    HELLO_MSG "hello with ifdef"
#else
#define   HELLO_MSG "hello"
#endif

void file3(void)
{
        printf("%s from %s\n", HELLO_MSG, __FILE__);
}
```

In file1.c, FILE1_CONSTANT serves as a symbolic constant, while in file2.c FILE2_IFDEF is an #ifdef which determines the output message. file3.c is modeled on file2.c, but with differing results based on the makefile.

Run nmake

```
$ nmake
```

Output

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
    /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -I- \
    -DFILE1_CONSTANT=50 -c file1.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
    /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -I- \
    -c file2.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
    /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -I- \
    -c file3.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O \
    -o target1 file1.o file2.o file3.o
```

Run the executable

```
$ target1
```

Output

```
hello from file1.c, constant is 50
hello from file2.c
hello from file3.c
```

From this call, -DFILE1_CONSTANT=50 is set up as an option to the preprocessor. Because FILE2_IFDEF is null, it is not used in the compilation of file2.c. And even though FILE3_IFDEF is non-null in the makefile, the Special Atom .PARAMETER suppresses its expansion.

Run nmake again

```
$ nmake FILE2_IFDEF=1
```

Output

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
      /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -I- \
      -DFILE2_IFDEF -c file2.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O \
      -o target1 file1.o file2.o file3.o
```

Run the executable

```
$ target1
```

Output

```
hello from file1.c, constant is 50
hello with ifdef from file2.c
hello from file3.c
```

In the second call, FILE2_IFDEF=1 is an explicit command-line option, resulting in -DFILE2_IFDEF being passed to the preprocessor. Now a modified message is given by the function in file2.c. Since all other files are up-to-date, nothing else is compiled.

Run nmake a third time

```
$ nmake
```

Output

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
      /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -I- \
      -c file2.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O \
      -o target1 file1.o file2.o file3.o
```

Run the executable

```
$ target1
```

Output

```
hello from file1.c, constant is 50
hello from file2.c
hello from file3.c
```

Since FILE2_IFDEF is null in the Makefile, file2.c is recompiled, restoring the original case.

Building an Executable from Multiple C Source Files in Private and Official Source Directories

This example demonstrates the construction of an executable from multiple C sourcefiles, and shows how viewpathing can be used to define both official and private build node locations. Such a technique is useful when developers only need to alter a subset of a project's source tree, while referring to the remainder via the VPATH environment variable.

In this example, the official node contains directories src and hdr, which contain source code and include files, respectively. Of the files listed below, module1_main.c and module1_common.c are in src, while module1.h is in hdr.

Contents of Makefile

```
.SOURCE.c : src
.SOURCE.h : hdr

module1 :: module1_main.c module1_common.c
```

Contents of module1_main.c

```
#include <stdio.h>
#include "module1.h"

int main(int argc, char **argv)
{
        printf("%s from %s\n", HELLO_MSG, __FILE__);
        common1(__FILE__, __LINE__);
        common2(__FILE__, __LINE__);
        exit(0);
}
```

Contents of module1_common.c:

```
#include <stdio.h>
#include "module1.h"

void common1(char *msg, int line)
{
        printf("%s, common1 in %s called by %s, line %d\n",
                HELLO_MSG, __FILE__, msg, line);
}

void common2(char *msg, int line)
{
        printf("%s, common2 in %s called by %s, line %d\n",
                HELLO_MSG, __FILE__, msg, line);
}
```

Contents of module1.h

```
#define    HELLO_MSG "hello"
```

Run nmake from official build node root

```
/home/user/official>> nmake
```

Output

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
      /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -Ihdr -I- \
      -c src/module1_main.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
      /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -Ihdr -I- \
      -c src/module1_common.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O \
      -o module1 module1_main.o module1_common.o
```

Run the executable

```
/home/user/official>> module1
```

Output

```
hello from src/module1_main.c
hello, common1 in src/module1_common.c called by src/module1_main.c, line 7
hello, common2 in src/module1_common.c called by src/module1_main.c, line 8
```

The executable module1 is constructed in typical fashion, with the message “hello” (from module1.h) listed in the output.

Now, suppose a developer wanted to change the text of the message, without altering any other source files. This could be accomplished by defining a private build node, along with a subset of the official directory structure, as follows.

Display the environment

```
/home/user/private>> ls -R

.:
hdr

./hdr:
module1.h
/home/user/private>> echo $VPATH
/home/user/private:/home/user/official
/home/user/private>>
/home/user/private>> cat hdr/module1.h
#define    HELLO_MSG “hello again”
```

Run nmake again

```
/home/user/private>> nmake
```

Output

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
      /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -Ihdr -I- \
      -c /home/user/official/src/module1_main.c
```

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools \
      /nmake/sparc5/v3.1.2/lib/probe/C/pp/890893F4obincc -Ihdr -I- \
      -c /home/user/official/src/module1_common.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O \
      -o module1 module1_main.o module1_common.o
```

Run the executable

```
/home/user/private>> module1
```

Output

```
hello again from /home/user/official/src/module1_main.c
hello again, common1 in /home/user/official/src/module1_common.c called by /home/user/
      official/src/module1_main.c, line 7
hello again, common2 in /home/user/official/src/module1_common.c called by /home/user/
      official/src/module1_main.c, line 8
```

The modified message from the private build node is used in conjunction with the source code in the official area. Also, note that the preexisting make object and statefiles (.mo and .ms) in the official area are referenced. Upon completion, only a statefile exists in the private area.

Building an Executable from Mixed C and C++ Source Files

The $(CC) variable points to your C++ compiler. The $(cc) variable points to your C compiler. :cc: identifies the C code. Prerequisites to :cc: are compiled using $(cc). Other files are compiled using $(CC). The C and C++ files can have the same filename suffix, because the suffix is not used by nmake to distinguish between the two languages.

NOTE:
Do not write metarules to run $(cc) or $(CC) depending on your source filename suffix. This will probably result in the wrong probe files being referenced for your C code, which will cause your C files to be preprocessed as C++ code. When you use :cc:, the correct probe files are referenced as necessary for each compiler. This is the only supported way to mix C and C++ code in the same makefile. Also, do not try to change the value of $(CC) in .makeinit, as this will also result in improper probing.

This example compiles $(CSRC) with the C compiler and $(CPLUSSRC) with the C++ compiler to create the executable rainbow

Contents of Makefile

```
CC = CC
cc = cc
```

```
CPLUSSRC = red.c orange.c yellow.c green.c
CSRC = blue.c indigo.c violet.c
rainbow :: $(CPLUSSRC) $(CSRC)
:cc: $(CSRC)
```

Run nmake

```
$ nmake
```

Output

```
+ CC -O --prelink_copy_if_nonlocal -I- -c red.c
+ CC -O --prelink_copy_if_nonlocal -I- -c orange.c
+ CC -O --prelink_copy_if_nonlocal -I- -c yellow.c
+ CC -O --prelink_copy_if_nonlocal -I- -c green.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c blue.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
    -c indigo.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
    -c violet.c
+ CC -O --prelink_copy_if_nonlocal -o rainbow red.o \
    orange.o yellow.o green.o blue.o indigo.o violet.o
+ cc -O -L/home/ferrari/chome/sun4/4.1 -o rainbow red.o \
    orange.o yellow.o green.o blue.o indigo.o biolet.o -1C
```

Building a Static Library from C Source Files

There are two ways of building a static library from C source files. Each will be discussed along with its merits.

Method 1: Using the :: Assertion Operator

Usage: <*target*> :: <*prequisites*>

CCFLAGS has no bearing on the name of the library.

This operator is recommended for building a library that has a name that does NOT start with lib.

NOTE:
All the libraries built in these examples may be installed by setting the LIBDIR variable appropriately in the makefile and invoking the install common action ($ nmake install).

Example 1 - Library Name Starts with lib

This example builds the library libfoo.a from the files specified in the SRC variable.

Contents of Makefile

```
SRC = file1.c file2.c file3.c
libfoo.a :: $(SRC)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file1.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file2.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file3.c
+ ar r libfoo.a file1.o file2.o file3.o
ar: creating libfoo.a
+ ignore ranlib libfoo.a
+ rm -f file1.o file2.o file3.o
```

Example 2 - Library Name Does Not Start with Lib

This example builds the library, foolib.a from the files specified in the SRC variable.

Contents of Makefile

```
SRC = file1.c file2.c file3.c
foolib.a :: $(SRC)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file1.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file2.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file3.c
+ ar r foolib.a file1.o file2.o file3.o
ar: creating foolib.a
+ ignore ranlib foolib.a
+ rm -f file1.o file2.o file3.o
```

Method 2: Using the :LIBRARY: Assertion Operator

Usage: <*name*> :LIBRARY: <*source_files*> [*required_libs*]

where <*name*> is the basename of the library without the prefix lib.

The archive library name depends on CCFLAGS.

If CCFLAGS includes any of the following flags,

-p,-pg,-g,-G

nmake searches for -p or -pg before -g or -G. Hence for CCFLAGS = -pg -g the archive library name generated is, lib<*name*>-pg.a

This operator will NOT build a static library that has a name that does NOT start with lib, e.g., it will not build <*name*>lib.a

Example 1

This example builds the library libfoo.a from the files specified in the SRC variable. CCFLAGS is set to -O (the default value).

Contents of Makefile

```
SRC = file1.c file2.c file3.c
foo :LIBRARY: $(SRC)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
     /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
     -c file1.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
     /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
     -c file2.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
     /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
     -c file3.c
+ ar r libfoo.a file1.o file2.o file3.o
ar: creating libfoo.a
+ ignore ranlib libfoo.a
+ rm -f file1.o file2.o file3.o
```

Example 2

This example builds the library libfoo-pg from the files specified in the SRC variable. CCFLAGS is set to -g -pg (i.e., profiling and debugging).

Contents of Makefile

```
CCFLAGS = -g -pg
SRC = file1.c file2.c file3.c
foo :LIBRARY: $(SRC)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -g -pg -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -D_TRACE_ -c file1.c
+ cc -g -pg -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -D_TRACE_ -c file2.c
+ cc -g -pg -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -D_TRACE_ -c file3.c
+ ar r libfoo-pg.a file1.o file2.o file3.o
ar: creating libfoo-pg.a
+ ignore ranlib libfoo-pg.a
+ rm -f file1.o file2.o file3.o
```

Building a Static Library from C Source Files with Prerequisite Libraries

It is possible to build a static library from a list of C source files and prerequisite libraries by using the :LIBRARY: assertion operator:

Usage: <*name*> :LIBRARY: <*source_files*> [*required_libs*]

where <*name*> is the basename of the library without the prefix lib.

The archive library name depends on CCFLAGS. If CCFLAGS includes any of the following flags,

-p,-pg,-g,-G

nmake searches for -p or -pg before -g or -G. Hence, for CCFLAGS = -pg -g the archive library name generated is, lib<*name*>-pg.a

This operator will NOT build a static library that has a name that does NOT start with lib, i.e., it will not build *<name>*lib.a

Example 1

This example builds the library libfoo2.a from the files specified in the SRC variable and the library in the LIBS variable, libfoo.a.

Contents of file1.c

```
file1()
{
 return 1;
}
```

Contents of file2.c

```
file2()
{
 return 1;
}
```

List contents of prerequisite library

```
$ ar tv lib/libfoo.a
```

Output

```
rw-r--r--674/674     38 Feb  5 13:33 1998 __.SYMDEF
rw-r--r--674/674     64 Feb  5 13:33 1998 file1.o
rw-r--r--674/674     64 Feb  5 13:33 1998 file2.o
```

Contents of Makefile

```
SRC = file11.c file12.c file13.c
LIBDIR = $(VROOT)/lib
LIBS = -lfoo
.SOURCE.a : lib
foo2 :LIBRARY: $(SRC) $(LIBS)
```

Contents of file11.c

```
file11()
{
 return 1;
}
```

Contents of file12.c

```
file12()
{
 return 1;
}
```

Contents of file13.c

```
file13()
```

```
{
 return 1;
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file11.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file12.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file13.c
+ ar r libfoo2.a file11.o file12.o file13.o
ar: creating libfoo2.a
+ ignore ranlib libfoo2.a
+ rm -f file11.o file12.o file13.o
```

NOTE:
The prerequisite information is not generated.

Run nmake again

```
$ nmake install
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file11.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file12.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -c file13.c
+ ar r libfoo2.a file11.o file12.o file13.o
ar: creating libfoo2.a
+ ignore ranlib libfoo2.a
+ rm -f file11.o file12.o file13.o
+ ignore cp libfoo2.a lib/libfoo2.a
+ mkdir -p lib/lib
+ 2> /dev/null
+ echo main(){return(0);}
+ 1> 1.2126321302.c
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc \
       -c 1.2126321302.c
+ + cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
       /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc \
       -o 1.2126321302.x 1.2126321302.o -l*
+ sed -e s/[][()+@?]/#/g
+ 2>& 1
x=ld: -l*: No such file or directory
+ 1> foo2.req
+ echo  -lfoo2
+ test ! -f ./lib/libfoo.a
+ echo  -lfoo
+ rm -f 1.2126321302.c 1.2126321302.o
+ ignore cp foo2.req lib/lib/foo2
```

Example 2

This example shows how the library and its prerequisite library are linked into an application. The assumption is that the application and library makefiles are separate makefiles. This method has the following advantages:

- it is not necessary to remember what other libraries to specify when linking -lfoo2 into an application
- only the library makefile must know what the library dependencies are
- if the library dependencies changes, only the library makefile must change.

Contents of Makefile

```
.SOURCE.a : lib
foo :: foo.c -lfoo2
```

Contents of foo.c

```
extern file1();
extern file2();
extern file11();
extern file12();
extern file13();
main() {
 file11();
 file12();
 file13();
 file1();
 file2();
return 0;
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c foo.c
+ cc -O -o foo foo.o lib/libfoo2.a lib/libfoo.a
```

Run nmake again

```
$ nmake install
```

Output

```
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- -c foo.c
+ cc -O -o foo foo.o lib/libfoo2.a lib/libfoo.a
+ ignore cp foo bin/foo
```

The following two examples use profiling, such as setting CCFLAGS to -g, thereby changing the library.

Example 3

This example builds the library from the files specified in the SRC variable and the library in the LIBS variable with CCFLAGS set to the debugging option (-g). Note the name of the generated library in the output.

Contents of Makefile

```
SRC = file11.c file12.c file13.c
CCFLAGS = -g
LIBDIR = $(VROOT)/lib
LIBS = -lfoo
.SOURCE.a : lib
foo2 :LIBRARY: $(SRC) $(LIBS)Run nmake
$ nmake install
```

Output

```
+ cc -g -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -D_TRACE_ -c file11.c
+ cc -g -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -D_TRACE_ -c file12.c
+ cc -g -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
      -D_TRACE_ -c file13.c
+ ar r libfoo2-g.a file11.o file12.o file13.o
ar: creating libfoo2-g.a
+ ignore ranlib libfoo2-g.a
+ rm -f file11.o file12.o file13.o
+ ignore cp libfoo2-g.a lib/libfoo2-g.a
+ echo main(){return(0);}
+ 1> 1.1350813541.c
+ cc -g -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
      /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -D_TRACE_ \
     -c 1.1350813541.c
+ + cc -g -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
     /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -D_TRACE_ \
     -o 1.1350813541.x 1.1350813541.o -l*
+ sed -e s/[][()+@?]/#/g
+ 2>& 1
x=ld: -l*: No such file or directory
+ 1> foo2.req
+ echo  -lfoo2
+ test ! -f ./lib/libfoo.a
+ echo  -lfoo
+ rm -f 1.1350813541.c 1.1350813541.o
+ ignore cp foo2.req lib/lib/foo2
```

Example 4

This example shows how the (profiling) library and its prerequisite library are linked into an application. The profiling library is requested by setting CCFLAGS to -g in the application makefile. As before, the assumption is that the application and library makefiles are separate makefiles.

Contents of Makefile

```
BINDIR = $(VROOT)/bin
CCFLAGS = -g
.SOURCE. : lib
foo :: foo.c -lfoo2
```

Run nmake

```
$ nmake
```

Output

```
+ cc -g -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
     /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
     -D_TRACE_ -c foo.c
+ cc -g -o foo foo.o lib/libfoo2-g.a lib/libfoo.a
```

Building a Shared Library from C Source Files

It is possible to build a shared library from its source files and optional -l library prerequisites by using the :LIBRARY: assertion operator:

Usage: <*name*> [*version#*] [*req_libs*] :LIBRARY: <*source_files*> [*dashl_req_libs*]

where <*name*> is the basename of the library without the prefix lib.

The position-independent option in $(CC.PIC) must be specified in the CCFLAGS variable.

This operator will NOT build static libraries that do not NOT start with lib, i.e., it will not build <*name*>lib.a. Dynamic libraries will be built without the lib prefix on win32 platforms and with the lib prefix on other platforms.

For SVR4 systems like Solaris platforms, the library is created with the specified version number. If no version is specified, the default is 1.0. On systems such as HP-UX the version number is not significant. In all cases the version number can be supressed by setting variable sharedlibvers=0.

Example 1

This example builds the library, libfoo.so.1.0 (the default version number on a SUNOS 4.x O/S) from the files specified in the SRC variable.

Contents of Makefile

```
LIBDIR = $(VROOT)/lib
CCFLAGS += $$(CC.PIC)
SRC = file1.c file2.c file3.c
foo :LIBRARY: $(SRC)
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -pic -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
    -c file1.c
+ cc -O -pic -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
    -c file2.c
+ cc -O -pic -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools \
    /nmake/sparc4/v3.1.2/lib/probe/C/pp/361FBDCFrucbcc -I- \
    -c file3.c
+ ar r libfoo.a file1.o file2.o file3.o
ar: creating libfoo.a
+ ignore ranlib libfoo.a
+ rm -f file1.o file2.o file3.o
+ sed -e /[ ][TDBC][ ][        ]*[_A-Za-z]/!d -e s/.*[ ][TDBC] \
    [     ][           ]*// -e /_STUB_/d -e s/^/-u /
+ nm libfoo.a
+ ld -o libfoo.so.1.0 -u _file1 -u _file2 -u _file3 libfoo.a
+ ignore cp libfoo.a lib/libfoo.a
+ /usr/bin/rm -f lib/libfoo.so
+ /usr/bin/ln -s libfoo.so.1.0 lib/libfoo.so
+ /usr/bin/cp libfoo.so.1.0 lib/libfoo.to.1.0
+ /usr/bin/rm -f lib/libfoo.so.1.0
+ /usr/bin/ln -s libfoo.to.1.0 lib/libfoo.so.1.0
+ /usr/bin/ln lib/libfoo.to.1.0 lib/libfoo.no.1.0
+ /usr/bin/rm -f lib/libfoo.so.1.0
+ /usr/bin/ln -s libfoo.no.1.0 lib/libfoo.so.1.0
+ /usr/bin/rm lib/libfoo.to.1.0
```

List libraries created in current directory

```
$ ls -l libfoo*
```

Output

```
-rw-r--r--  1 pete         496 Feb  9 14:10 libfoo.a
-rwxr-xr-x  1 pete       24576 Feb  9 14:10 libfoo.so.1.0
```

List libraries created in lib directory

```
$ ls -l lib/libfoo*
```

Output

```
-rw-r--r--  1 nmake         496 Feb  9 14:10 lib/libfoo.a
-rwxr-xr-x  1 nmake       24576 Feb  9 14:10 lib/libfoo.no.1.0
lrwxrwxrwx  1 nmake          13 Feb  9 14:10 lib/libfoo.so -> libfoo.so.1.0
lrwxrwxrwx  1 nmake          13 Feb  9 14:10 lib/libfoo.so.1.0 -> libfoo.no.1.0
```

Example 2

This example builds the library libfoo.sl on an HP-UX 10.10 O/S, from the files specified in the SRC variable.

Contents of Makefile

```
LIBDIR = $(VROOT)/lib
CCFLAGS += $$(CC.PIC)
SRC = file1.c file2.c file3.c
foo :LIBRARY: $(SRC)
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O +z -t p,/home/bugati/nmake/build/3.1.2/year2000/jason \
      /lib/cpp -I-D/home/bugati/nmake/build/3.1.2/year2000/jason \
      /lib/probe/C/pp/B3A01E21.bincc -I- -c file1.c
make: warning: file1.o file system time leads local time by at least 2.00s
+ cc -O +z -t p,/home/bugati/nmake/build/3.1.2/year2000/jason \
      /lib/cpp -I-D/home/bugati/nmake/build/3.1.2/year2000/jason \
      /lib/probe/C/pp/B3A01E21.bincc -I- -c file2.c
+ cc -O +z -t p,/home/bugati/nmake/build/3.1.2/year2000/jason \
      /lib/cpp -I-D/home/bugati/nmake/build/3.1.2/year2000/jason \
      /lib/probe/C/pp/B3A01E21.bincc -I- -c file3.c
+ ar rf libfoo.a file1.o file2.o file3.o
ar: creating libfoo.a
+ rm -f file1.o file2.o file3.o
+ /bin/nm -p libfoo.a
+ sed -e /[ ][TDBC][ ][          ]*[_A-Za-z]/!d -e s/.*[ ][TDBC] \
      [      ][            ]*// -e /_STUB_/d -e s/^/-u /
+ /bin/ld -b -o libfoo.sl.1.0 -u file1 -u file2 -u file3 libfoo.a
+ ignore cp libfoo.a lib/libfoo.a
+ /usr/bin/rm -f lib/libfoo.sl
+ /usr/bin/ln -s libfoo.sl.1.0 lib/libfoo.sl
+ /usr/bin/cp libfoo.sl.1.0 lib/libfoo.to.1.0
+ /usr/bin/rm -f lib/libfoo.sl.1.0
+ /usr/bin/ln -s libfoo.to.1.0 lib/libfoo.sl.1.0
+ /usr/bin/ln lib/libfoo.to.1.0 lib/libfoo.no.1.0
+ /usr/bin/rm -f lib/libfoo.sl.1.0
+ /usr/bin/ln -s libfoo.no.1.0 lib/libfoo.sl.1.0
+ /usr/bin/rm lib/libfoo.to.1.0
```

List libraries created in current directory

```
$ ls -l libfoo*
```

Output

```
-rw-r--r--   1 pete     pete       2560 Feb  9 15:25 libfoo.a
-rwxr-xr-x   1 pete     pete      12310 Feb  9 15:25 libfoo.sl.1.0
```

List libraries created in lib directory

```
$ ls -l lib/libfoo*
```

Output

```
-rw-r--r--   1 pete     pete       2560 Feb  9 15:25 lib/libfoo.a
-rwxr-xr-x   1 pete     pete      12310 Feb  9 15:25 lib/libfoo.no.1.0
lrwxrwxrwx   1 pete     pete         13 Feb  9 15:25 lib/libfoo.sl -> libfoo.sl.1.0
lrwxrwxrwx   1 pete     pete         13 Feb  9 15:25 lib/libfoo.sl.1.0 -> libfoo.no.1.0
```

Building a C++ Class Template Library

This example shows how to build a C++ class template library in the simplest possible way. We use the Alcatel-Lucent C++ Compiler version 4.1. The procedures for other C++ compilers may be different, but the underlying concepts remain the same.

The example comprises two template classes, A and B, each of which is to be placed into its own class library. Template class A is to be placed into library liba.a, and template class B is to be placed into libb.a. Following standard C++ source code organization practice, we put the client interface declarations into a header (.h) file, and the class implementation details into a corresponding implementation (.C) file. As required by C++ 4.1, we also put template entity definitions in the header file. Following is the entire C++ source for libraries liba.a and libb.a. Note that the implementation of these template classes requires the use of the List template class, which is a foundation class provided by the C++ 4.1 compiler.

Contents of A.h:

```
void inita();
template <class T>
   class A {
   public:
    A();
};

template <class T>
A<T>::A()
{
    inita();
}
```

Contents of A.C:

```
 #include <List.h>
#include <A.h>

void
inita()
{
    List<double> ld;
}
```

Contents of B.h

```
void initb();

template <class T>
class B {
public:
    B();
};

template <class T>
B<T>::B()
{
    initb();
}
```

Contents of B.C

```
#include <List.h>
#include <B.h>

void
initb()
{
    List<double> ld;
}
```

We illustrate a use of the library with a simple client application, defined in app.C.

Contents of app.C

```
     #include <List.h>
#include <A.h>
#include <B.h>
```

```
template class List<double>;

main()
{
   A<int> a;
   B<int> b;
}
```

This Makefile shows the simplest way to build both template class libraries and then link them into our sample client application. We include the -l++ dependency because it contains the List template class, required by the liba.a and libb.a implementations.

Contents of Makefile

```
CC = CC
app :: app.C -la -lb -l++
liba.a :: A.C
libb.a :: B.C
```

Run nmake

```
$ nmake
```

Output

```
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c app.C
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c A.C
+ ar r liba.a A.o
ar: creating liba.a
+ rm -f A.o
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c B.C
+ ar r libb.a B.o
ar: creating libb.a
+ rm -f B.o
+ CC -O --prelink_copy_if_nonlocal -o app app.o liba.a \
   libb.a -l++
C++ prelinker: voidP_List_sort_internal(List<double> &, \
   int (*)(const double &, const double &)) assigned to file app.o
C++ prelinker: lnnk_ATTLC<double>::pool assigned to file app.o
C++ prelinker: Const_listiter<double>::next() assigned to \
   file app.o
C++ prelinker: lnnk_ATTLC<double>::getnewlnnk_ATTLC(const \
   double &) assigned to file app.o
C++ prelinker: lnnk_ATTLC<double>::deletelnnk_ATTLC \
   (double &, lnnk_ATTLC<double> *) assigned to file app.o
C++ prelinker:\ Const_listiter<double>::Const_listiter<double>(const\ List<double> &)
   assigned to file app.o
C++ prelinker: B<int>::B<int>() assigned to file app.o
C++ prelinker: A<int>::A<int>() assigned to file app.o
C++ prelinker: lnnk_ATTLC<double>::copy() assigned to file app.o
C++ prelinker: lnnk_ATTLC<double>::operator ==(lnk_ATTLC &) assigned to file app.o
```

```
C++ prelinker: lnnk_ATTLC<double>::~lnnk_ATTLC<double>() assigned to file app.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex1/. app.C
C++ prelinker: lnnk_ATTLC<double>::operator new(unsigned int) assigned to file app.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex1/. app.C
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -O -L/apps/tools/sparc5/C++/4.1 -o app app.o
   liba.a libb.a /apps/tools/sparc5/C++/4.1/libg2++.a /apps/tools/sparc5/C++/4.1/libGA.a /
   apps/tools/sparc5/C++/4.1/libGraph.a /apps/tools/sparc5/C++/4.1/lib++.a -lC
```

This approach places template instances A<int>, B<int>, and the implementation of List<double> into the application module app.o.

This assignment occurs because the C++ 4.1 language system performs template instantiation at link time into some available object file, and app.o is the only object file accessible to the compiler at link time.

Note that, in order for this build procedure to work, app.C had to include List.h, and had to explicitly instantiate List<double>, even though the List template is not used directly in app.C. We can eliminate this requirement by using the technique of template class library closure, the subject of the next example.

Building and Closing a C++ Class Template Library

This example builds and closes a C++ class template library using the Alcatel-Lucent C++ 4.1 compiler. The procedures for other C++ compilers may be different, but the underlying concepts remain the same. A closed template class library resolves internal template entity references so that they do not have to be instantiated in the client application.

The source for the library components, A.h, A.C, B.h, B.C, is identical to that used in the previous example:

Contents of A.h

```
void inita();

template <class T>
class A {
public:
   A();
};

template <class T>
A<T>::A()
{
   inita();
}
```

Contents of A.C

```
#include <List.h>
#include <A.h>

void
inita()
{
   List<double> ld;
}
```

Contents of B.h

```
void initb();

template <class T>
class B {
public:
   B();
};

template <class T>
B<T>::B()
{
   initb();
}
```

Contents of B.C

```
#include <List.h>
#include <B.h>

void
initb()
{
   List<double> ld;
}
```

The source for the client application, app.C, is similar to that used in the previous example. The only difference is the removal of the lines required to instantiate List<double> in the application object module.

Contents of app.C

```
#include <A.h>
#include <B.h>

main()
{
   A<int> a;
   B<int> b;
}
```

Class template library closure requires an extra step at library build time, necessitating modifications and additions to the base rules which specify how to build archive libraries. Since these modifications and additions are useful throughout a project, they are generally placed in the project global rules file. In this example, we place these modified and enhanced rules in a separate file, global.mk:

Contents of global.mk

```
.ARCHIVE.LIST. : .FUNCTION
        local I
        .UNBIND : . $(***:T=F:T=G:A!=.ARCHIVE)
        .UNION : .CLEAR
        for I $(***:T=F:T=G:A!=.ARCHIVE)
                .REBIND : $(I)
                .UNION : $(I)
        end
        return $(~.UNION)

.DO.ARCHIVE : .AFTER .VIRTUAL .FORCE .REPEAT .ARCLEAN
        $(.ARCHIVE.LIST.:K=$(AR) $(ARFLAGS) $(<<))

.ARCHIVE.o : .CLEAR .USE .ARCHIVE (AR) (ARFLAGS) (LDFLAGS)\ .ARPREVIOUS
        .ARUPDATE .DO.ARCHIVE
        $(^:?$$(CP) $$(^) $$(<)$$("\n")??)$(.ARPREVIOUS.$(<:B:S)\
                :@?$(IGNORE) $$(AR) d $$(<) $$(.ARPREVIOUS.$$(<:B:S))\
                $$("\n")??)
        if [ "$(>:A!=.ARCHIVE)" ]
        then    if [ -f $(<) ]
                then    silent ignore $(AR) d $(<) $(>:A!=.ARCHIVE)
                        $(CC) --prelink_objects --prelink_copy_if_nonlocal\ $(LDFLAGS) -L/lib -L/usr/lib


                else
                        $(CC) --prelink_objects --prelink_copy_if_nonlocal\ $(LDFLAGS) -L/lib -L/usr/lib
                fi
        fi
```

The library directory makefile then references global.mk through an include statement.

Contents of Makefile

```
include global.mk
CC = CC

app :: app.C -la -lb -l++

libb.a :: B.C

liba.a :: A.C -lb
```

Run nmake

```
$ nmake
```

Output

```
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c app.C
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c A.C
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c B.C
+ [ B.o ]
+ [ -f libb.a ]
+ CC --prelink_objects --prelink_copy_if_nonlocal -L/lib -L/usr/lib B.o --
C++ prelinker: List<double>::List<double>() assigned to file B.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex2/. B.C
C++ prelinker: lnnk_ATTLC<double>::pool assigned to file B.o
C++ prelinker: lnnk_ATTLC<double>::copy() assigned to file B.o
C++ prelinker: lnnk_ATTLC<double>::operator ==(lnk_ATTLC &) assigned to file B.o
C++ prelinker: lnnk_ATTLC<double>::~lnnk_ATTLC<double>() assigned to file B.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex2/. B.C
C++ prelinker: lnnk_ATTLC<double>::operator new(unsigned int) assigned to file B.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex2/. B.C
+ ar r libb.a B.o
ar: creating libb.a
+ [ A.o ]
+ [ -f liba.a ]
+ CC --prelink_objects --prelink_copy_if_nonlocal -L/lib -L/usr/lib A.o -- libb.a
+ ar r liba.a A.o
ar: creating liba.a
+ CC -O --prelink_copy_if_nonlocal -o app app.o liba.a libb.a -l++
C++ prelinker: B<int>::B<int>() assigned to file app.o
C++ prelinker: A<int>::A<int>() assigned to file app.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex2/. app.C
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -O -L/apps/tools/sparc5/C++/4.1 -o app app.o
   liba.a libb.a /apps/tools/sparc5/C++/4.1/libg2++.a /apps/tools/sparc5/C++/4.1/libGA.a /
   apps/tools/sparc5/C++/4.1/libGraph.a /apps/tools/sparc5/C++/4.1/lib++.a -lC
```

The List<double> template entities are now instantiated into object file B.o, which is archived into libb.a. Because of this, they do not have to be instantiated directly into the application file, app.o. This has the advantage that the author of app.C does not need to know about the template entities used in the implementation of the classes it uses. We also improve build performance for both speed and space required.

The List<double> template entity was instantiated into libb.a only, and not also into liba.a, even though it is referenced in both libraries. If the List<double> template entities had been instantiated into both A.o and B.o, a multiple definition error would have resulted at link time. This introduces a build dependency between the libraries, indicated by the addition of -lb on the liba.a command line. This indicates that liba.a is dependent upon libb.a, and should always be built after libb.a. Furthermore, these libraries must always be used together, in the specified order.

Building, Closing, and Preinstantiating a C++ Class Template Library

This example builds, closes, and preinstantiates a C++ class template library, using the Alcatel-Lucent C++ 4.1 compiler. The procedures for other C++ compilers may be different, but the underlying concepts remain the same. Class library preinstantiation is a class library optimization technique in which commonly used template entity instantiations are built into the library, even though they are not referenced anywhere within the library.

Assume that we believe that instantiations of template classes A<int> and B<int> will be commonly required by client applications. The necessary modifications are shown below. The contents of the library source files are similar to those used in previous examples. There is only one difference: we inserted an explicit instantiation declaration on the fourth line of A.C and B.C.

Contents of A.h

```
void inita();

template <class T>
class A {
public:
   A() { inita(); }
};
```

Contents of A.C

```
#include <List.h>
#include <A.h>

template class A<int>;

void
inita()
{
   List<double> ld;
}
```

Contents of B.h

```
void initb();

template <class T>
class B {
public:
   B() { initb(); }
};
```

Contents of B.C

```
#include <List.h>
#include <B.h>

template class B<int>;

void
initb()
{
   List<double> ld;
}
```

Contents of app.C

```
#include <A.h>
#include <B.h>

main()
{
   A<int> a;
   B<int> b;
}
```

Contents of global.mk

```
.ARCHIVE.LIST. : .FUNCTION
        local I
        .UNBIND : . $(***:T=F:T=G:A!=.ARCHIVE)
        .UNION : .CLEAR
        for I $(***:T=F:T=G:A!=.ARCHIVE)
                .REBIND : $(I)
                .UNION : $(I)
        end
        return $(~.UNION)

.DO.ARCHIVE : .AFTER .VIRTUAL .FORCE .REPEAT .ARCLEAN
        $(.ARCHIVE.LIST.:K=$(AR) $(ARFLAGS) $(<<))

.ARCHIVE.o : .CLEAR .USE .ARCHIVE (AR) (ARFLAGS) (LDFLAGS)\ .ARPREVIOUS
        .ARUPDATE .DO.ARCHIVE
        $(^:?$$(CP) $$(^) $$(<)$$("\n")??)$(.ARPREVIOUS.$(<:B:S)\
                :@?$(IGNORE) $$(AR) d $$(<) $$(.ARPREVIOUS.$$(<:B:S)\
                $$("\n")??)
        if [ “$(>:A!=.ARCHIVE)” ]
        then    if [ -f $(<) ]
                then    silent ignore $(AR) d $(<) $(>:A!=.ARCHIVE)
                        $(CC) --prelink_objects --prelink_copy_if_nonlocal\ $(LDFLAGS) -L/lib -L/usr/lib
                else
                        $(CC) --prelink_objects --prelink_copy_if_nonlocal\ $(LDFLAGS) -L/lib -L/usr/lib
                fi
        fi
```

Contents of Makefile

```
include global.mk
CC = CC
app :: app.C -la -lb -l++
libb.a :: B.C
liba.a :: A.C -lb
```

Run nmake

```
$ nmake
```

Output

```
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c app.C
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c A.C
+ CC -O --prelink_copy_if_nonlocal -I- -I. -c B.C
+ [ B.o ]
+ [ -f libb.a ]
+ CC --prelink_objects --prelink_copy_if_nonlocal -L/lib -L/usr/lib B.o --
C++ prelinker: List<double>::List<double>() assigned to file B.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex3/. B.C
C++ prelinker: lnnk_ATTLC<double>::pool assigned to file B.o
C++ prelinker: lnnk_ATTLC<double>::copy() assigned to file B.o
C++ prelinker: lnnk_ATTLC<double>::operator ==(lnk_ATTLC &) assigned to file B.o
C++ prelinker: lnnk_ATTLC<double>::~lnnk_ATTLC<double>() assigned to file B.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex3/. B.C
C++ prelinker: lnnk_ATTLC<double>::operator new(unsigned int) assigned to file B.o
C++ prelinker: executing: /apps/tools/sparc5/C++/4.1/CC -c -O --prelink_copy_if_nonlocal -
   I- -I/home/bugati/gms/wrk/nmake/manual/ex3/. B.C
+ ar r libb.a B.o
ar: creating libb.a
+ [ A.o ]
+ [ -f liba.a ]
+ CC --prelink_objects --prelink_copy_if_nonlocal -L/lib -L/usr/lib A.o -- libb.a
+ ar r liba.a A.o
ar: creating liba.a
+ CC -O --prelink_copy_if_nonlocal -o app app.o liba.a libb.a -l++
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -O -L/apps/tools/sparc5/C++/4.1 -o app app.o
   liba.a libb.a /apps/tools/sparc5/C++/4.1/libg2++.a /apps/tools/sparc5/C++/4.1/libGA.a /
   apps/tools/sparc5/C++/4.1/libGraph.a /apps/tools/sparc5/C++/4.1/lib++.a -lC
```

Notice that A<int> and B<int> are no longer instantiated into the application file app.o.

Building an Executable from Multiple C Source Files Using Parallel Build

This example shows how nmake can be used to build multiple targets concurrently, which allows nmake to speed up processing by working in an interleaved fashion, thus decreasing its wait time.

Contents of Makefile

```
all : target1 target2
target1 :: file1.c
target2 :: file2.c
```

Run nmake

```
$ nmake
```

Output

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
      pp/890893F4obincc -I- -c file1.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o target1 file1.o
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
      pp/890893F4obincc -I- -c file2.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o target2 file2.o
```

Note how the targets are made sequentially; the actions for target2 are not initiated until target1 is complete. Improvements in build time could be obtained if both target actions were allowed to build concurrently.

Run nmake again

```
$ nmake clobber
```

Output

```
+ ignore rm -f target1 target2 file2.o file1.o Makefile.mo Makefile.ms
```

Run nmake a third time

```
$ nmake -j2
```

Output

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
      pp/890893F4obincc -I- -c file1.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
      pp/890893F4obincc -I- -c file2.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o target1 file1.o
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o target2 file2.o
```

Using the -j2 option, nmake is set to execute two target actions concurrently. In this case, the actions for target1 and target2 are interleaved, decreasing overall build time.

Building a Project Spanning Multiple Source and Header Directories with a Project Library Prerequisite

This example demonstrates a sample program that is dependent on both a common library and multiple header files. Using a supplied build node hierarchy, it illustrates how recursive nmake calls can simplify build tasks while maintaining the association between a target and its required files, when spread across multiple nodes.

When initiated, the build recursively calls nmake in both the lib and app subdirectories. In all, it uses three makefiles in three directories.

Contents of Makefile

```
:MAKE: lib - app
```

This makefile starts the build process, issuing builds in both the lib and app subdirectories. Note that the use of the - Special Atom ensures that the build in lib will be completed before any (possibly) dependent targets in app are built.

Contents of Makefile (lib sudirectory)

```
mycom :LIBRARY: gettime.c gettz.c
```

The library libmycom.a depends upon the source files gettime.c and gettz.c, which contain the routines gettime() and gettz(), respectively. gettime() and gettz() are utility functions that return time/date and timezone information, respectively. In a Solaris environment, these utility functions call time(2), ctime(3C), and getenv(3C).

Contents of Makefile (app sudirectory)

```
.SOURCE.a : ../lib
.SOURCE.h : ../hdr ../hdr2

target :: maintarget.c -lmycom
```

The target target depends on both the source file maintarget.c and the library libmycom.a. This makefile tells nmake to look for libraries in ../lib and for headers in ../hdr and ../hdr2.

Contents of app/maintarget.c

```
#include <stdio.h>
#include “first.h” /* located in ../hdr */
#include “second.h” /* located in ../hdr2 */

int
main(int argc, char **argv)
{
        char *gettime(void),
         *gettz(void);
```

```
                printf("The current time is %s", gettime());
                printf("Your time zone is %s\n", gettz());
                exit(0);
}
```

Contents of lib/gettime.c

```
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

char *gettime(void)
{
                time_t     mytime = time(0);
                return(ctime(&mytime));
}
```

Contents of lib/gettz.c

```
#include <stdlib.h>

char *gettz(void)
{
                return (getenv("TZ"));
}
```

Issue nmake

```
$ nmake
```

Output

```
lib:
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
        pp/890893F4obincc -I- -c gettime.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
        pp/890893F4obincc -I- -c gettz.c
+ ar r libmycom.a gettime.o gettz.o
+ rm -f gettime.o gettz.o
app:
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
        pp/890893F4obincc -I../hdr -I../hdr2 -I- -c maintarget.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o target maintarget.o ../lib/libmycom.a
$ app/target
The current time is Wed Mar 18 11:14:33 1998
Your time zone is US/Eastern
$
```

As expected, the library libmycom.a is created and is linked with the resultant target. Note that ../hdr and ../hdr2 are supplied as include file directories, because nmake located the header prerequisites for maintarget.c in those directories.

Now suppose that a change induces an update to the modification time of one of the library files.

Change modification time on library file

```
$ touch lib/gettz.c
```

Issue nmake agai

```
$ nmake
```

Output

```
lib:
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
      pp/890893F4obincc -I- -c gettz.c
+ ar r libmycom.a gettz.o
+ rm -f gettz.o
app:
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o target maintarget.o ../lib/libmycom.a
$
```

Note that only the updated library source file, gettz.c, is recompiled for libmycom.a. Also, the target is relinked with the altered archive in accordance with the dependency specified in the app subdirectory makefile.

Building Project Executables for Multiple Platforms

This example illustrates some basic techniques that have been used for building on multiple platforms. Typically, projects are required to perform different actions when building on a collection of operating system platforms. nmake can be tailored to detect the build environment automatically and to set any architecture-specific information associated with the environment for different scenarios. The approach below uses a global makefile, which is then included in the sample makefile for general use.

Contents of global Makefile

```
/* Use :COMMAND: to set nmake variables based on calls to uname(1) */

uname_s :COMMAND:
        set +x
        uname -s|sed ‘s/-//g’

uname_c1 :COMMAND:
        set +x
        uname -r|cut -c1
```

```
uname_f2 :COMMAND:
        set +x
        uname -r|cut -f2 -d’.’

/* Check _architecture_ from probe information, using it to set architecture-specific variables
    needed by product build. This technique sets the variable MTYPE from a chosen
    combination of uname(1) output. The cases below work with specified versions of Solaris,
    SunOS (pre-Solaris), HP-UX, IRIX, and ULTRIX. */

if "$(_architecture_:N=sparc)"
        MTYPE = $(uname_s)$(uname_c1)
        if "$(MTYPE:N!=SunOS[45])"
                error 2 ERROR: cannot determine specific sparc
                error 3 operating system version
        end
        if “$(MTYPE)” == “SunOS4”
                SYS_BSD = 1
                SYS_53 =
        elif “$(MTYPE)” == “SunOS5”
                JIO_WIN = 1
        end
elif "$(_architecture_:N=hp9000s700)"
        MTYPE=$(uname_s)$(uname_c1)$(uname_f2)
        if "$(MTYPE:N!=HPUXA09|HPUXB10)"
                error 2 ERROR: cannot determine specific hppa
                error 3 operating system version
        end
        NET_DK = 1
elif "$(_architecture_:N=mips)"
        MTYPE=$(uname_s)$(uname_c1)
        if "$(MTYPE:N!=IRIX[45]|ULTRIX4|*3|*4)"
                error 2 ERROR: cannot determine specific mips
                error 3 operating system version
        end
        if "$(MTYPE:N=*3)"
                MTYPE = FTR3
        end
        if "$(MTYPE:N=*4)"
                MTYPE = FTR4
        end
else
        error 3 ERROR: cannot determine machine architecture
end
```

Contents of Makefile

```
include "globalmake"

.INIT : .MAKE
        print **** nmake running on $(_architecture_) architecture, $(MTYPE) ****

hello :: hello.c
```

The combination of probe information with uname(1) output allows a great deal of flexibility; new platforms can easily be introduced. If a set of variables must be customized, the changes can be isolated to the global makefile, minimizing file changes.

Run nmake on Solaris

```
$ nmake
```

Output

```
**** nmake running on sparc architecture, SunOS5 ****
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
    pp/890893F4obincc -I- -c hello.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o hello hello.o
```

Run nmake on SunOS

```
$ nmake
```

Output

```
**** nmake running on sparc architecture, SunOS4 ****
+ cc -O -Qpath /tools/nmake/sparc4/v3.1.2/lib -I-D/tools/nmake/sparc4/v3.1.2/lib/probe/C/pp/
    835E4F4F5bincc -I- -c hello.c
+ cc -O -o hello hello.o
```

Run nmake on HP-UX

```
$ nmake
```

Output

```
**** nmake running on hp9000s700 architecture, HPUXB10 ****
+ cc -O -t p,/tools/nmake/hppa10/v3.1.2/lib/cpp -I-D/tools/nmake/hppa10/v3.1.2/lib/probe/C/
    pp/0F689CA5rbincc -I- -c hello.c
+ cc -O -o hello hello.o
```

Run nmake on IRIX

```
$ nmake
```

Output

```
**** nmake running on mips architecture, IRIX5 ****
+ ppcc -i /tools/nmake/sgi/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sgi/v3.1.2/lib/probe/C/pp/
    0F689CA5rbincc -I- -c hello.c
+ ppcc -i /tools/nmake/sgi/v3.1.2/lib/cpp cc -O -o hello hello.o
```

Building using a 3rd-Party Software Package

Example 1

This example shows the libraries and headers of packages mypack1 and mypack2 (simulated for this example) being picked up from the standard default directories include and lib and used to build an executable.

List files in current directory

```
$ ls src
```

Output

```
main.c   pack1.c  pack2.c
```

Contents of main.c

```
#include <stdio.h>
extern pack1();
extern pack2();
main() {
 pack1();
 pack2();
 printf("\n");
 exit(1);
}
```

Contents of pack1.c

```
#include <pack1.h>
pack1() {
 printf("\nIn pack1() ...");
 func1();
 func2();
 func3();
 return 1;
}
```

Contents of pack2.c

```
#include <pack2.h>
pack2() {
 printf(“\nIn pack2() ...”);
 func5();
 func4();
 return 1;
}
```

List files in include directory

```
$ ls include
```

Output

```
pack1.h  pack2.h
```

Contents of pack1.h

```
#include <func1.h>
#include <func2.h>
#include <func3.h>
```

Contents of pack2.h

```
#include <func4.h>
#include <func5.h>
```

List files in mypack1

```
$ ls -FCR /home/bugati/pete/jaguar/pack1
```

Output

```
/home/bugati/pete/jaguar/pack1:
include/  lib/
/home/bugati/pete/jaguar/pack1/include:
func1.h  func2.h  func3.h
/home/bugati/pete/jaguar/pack1/lib:
libmypack1.a
```

List files in mypack2

```
$ ls -FCR /home/bugati/pete/jaguar/workspace/pack2
```

Output

```
/home/bugati/pete/jaguar/workspace/pack2:
include/  lib/
/home/bugati/pete/jaguar/workspace/pack2/include:
func4.h   func5.h   func6.h
/home/bugati/pete/jaguar/workspace/pack2/lib:
libmypack2.a
```

Contents of Makefile

```
PACKAGE_mypack1 = /home/bugati/pete/jaguar/pack1
PACKAGE_mypack2 = /home/bugati/pete/jaguar/workspace/pack2

:PACKAGE: - mypack1
:PACKAGE: mypack2

.SOURCE.c : src
.SOURCE.h : include
myexec :: main.c pack1.c pack2.c -lmypack2 -lmypack1
```

Here PACKAGE_mypack1 is the root of mypack1 and PACKAGE_mypack2 is the root of mypack2. The - before mypack1 prevents $(INCLUDEDIR) from being redefined to $(INSTALLROOT)/include/mypack1. The header and library directories of mypack2 are searched before those of mypack1. $(INCLUDEDIR) is redefined to $(INSTALLROOT)/include/mypack2.

Run nmake

```
$ nmake
```

Output

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
      pp/BFE0A4FEobincc -I- -c src/main.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
      pp/BFE0A4FEobincc -I- -Iinclude -I/home/bugati/pete/jaguar/pack1/include -c src/
      pack1.c
```

```
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -I-D/tools/nmake/sparc5/v3.1.2/lib/probe/C/
    pp/BFE0A4FEobincc -I- -Iinclude -I/home/bugati/pete/jaguar/workspace/pack2/include -
    c src/pack2.c
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o myexec main.o pack1.o pack2.o /home/
    bugati/pete/jaguar/workspace/pack2/lib/libmypack2.a /home/bugati/pete/jaguar/pack1/
    lib/libmypack1.a
```

Run the executable

```
$ ./myexec
```

Output

```
In pack1() ...
In func1() of package "mypack1"
In func2() of package "mypack1"
In func3() of package "mypack1"
In pack2() ...
In func5() of package "mypack2"
In func4() of package "mypack2"
```

Example 2

This example is a bit more practical and illustrates the benefit of using the :PACKAGE: operator. This is an excerpt from a typical development makefile from a project that uses a number of third-party software packages, such as C5, G2, TUXEDO and INFORMIX.

Section of Makefile

```
C5DIR = /opt/tools/aitools
        .
        .

G2DIR = /opt/tools/g2
        .
        .

INFORMIXDIR = /opt/tools/informix
        .
        .

TUXDIR = /tux/topas_units
        .
        .

.SOURCE.a : $(TUXDIR)/lib $(INFORMIXDIR)/lib $(G2DIR)/lib $(C5DIR)/lib
.SOURCE.h : $(TUXDIR)/include $(INFORMIXDIR)/include $(G2DIR)/include $(C5DIR)/
    include
```

Here C5DIR is the root of the C5 package, G2DIR is the root of the G2 package, INFORMIXDIR is the root of the INFORMIX package, and TUXDIR is the root of the TUXEDO package.

Using the :PACKAGE: operator would be:

Section of Makefile using :PACKAGE:

```
PACKAGE_C5 = /opt/tools/aitools
        .
        .

PACKAGE_G2 = /opt/tools/g2
        .
        .

PACKAGE_INFORMIX = /opt/tools/informix
        .
        .

PACKAGE_TUXDIR = /tux/topas_units
        .
        .

:PACKAGE: - C5 G2 INFORMIX TUXDIR
```

Here PACKAGE_C5 is the root of the C5 package, PACKAGE_G2 is the root of the G2 package, PACKAGE_INFORMIX is the root of the INFORMIX package, and PACKAGE_TUXDIR is the root of the TUXEDO package. Library and header directories are searched as they were in the previous makefile, headers in include and libraries in lib, the default locations. The - in the last line prevents the redefinition of INCLUDEDIR.

The benefits of using :PACKAGE: in this way are:

- Makes the makefile a bit more readable. Though one could name the package root more appropriately, the :PACKAGE: operator facilitates the more descriptive naming scheme. One need only glance at the makefile with the operator to realize that C5, G2, INFORMIX, and TUXDIR are third-party software packages.
- Reduces the need for .SOURCE.a and .SOURCE.h assertions for the packages, making the makefile a bit more concise.

Building using the build log output serialization feature

Normally, when multiple build commands are executed concurrently (e.g. due to NPROC set > 1, or use of coshell(1)), the output of simultaneously executing commands appears intermixed in the build log. This occurs because output appears on the output immediately, as it is written. Use the nmakelog command to create a serialized build log. When nmakelog is used, the usual unserialized output is still produced to allow real-time monitoring of the build progress. In addition, a serialized build log is appended to the file makelog.

Example 1

The following example shows a concurrent build of an executable using Alcatel-Lucent C++ 3.0.3. Note that each compile line produces 3 lines of output in addition to the command trace of the compile line itself.

Contents of Makefile

```
CC=CC
m :: m.c a.c b.c c.c -l++
```

Configure nmake for a maximum of 12 concurrent actions. Use nmakelog rather that nmake to obtain a serialized log file. nmakelog accepts the same command line arguments as nmake.

```
$ nmakelog -j12
```

Output shows 4 concurrently executing compilations followed by a prelink and link step. In this output, each line appears in the order it was printed during the build.

Output

```
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -c m.c
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -c a.c
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -c b.c
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -c c.c
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
CC  m.c:
CC  a.c:
CC  b.c:
CC  c.c:
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -c  -O m.c
CC[ptcomp]: waiting to lock repository ./ptrepository, sleeping ...
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -c  -O a.c
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -c  -O b.c
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -c  -O c.c
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -I. -o m m.o a.o b.o c.o -l++
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -Wl,-L/apps/tools/sparc5/C++/3.0.3  -o
      m  -O m.o a.o b.o c.o -lg2++ -lGA -lGraph -l++ -lC
Instantiating List<char>...
Instantiating List<double>...
Instantiating lnnk_ATTLC<char>...
Instantiating lnnk_ATTLC<double>...
```

The serialized build log is appended to file makelog. The output of each shell action block (build command) appears contiguously in the output file. The entire action block output appears in the log file in order of command completion. This view of the build is easier to read and troubleshoot than the normal, unserialized version.

Contents of makelog

```
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -c m.c
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
CC  m.c:
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -c  -O m.c
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -c a.c
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
CC  a.c:
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -c  -O a.c
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -c b.c
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
CC  b.c:
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -c  -O b.c
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -c c.c
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
CC  c.c:
CC[ptcomp]: waiting to lock repository ./ptrepository, sleeping ...
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -c  -O c.c
+ CC -O -I-D/home/porsche/gms/wrk/nmake/312/solaris/lib/probe/C/pp/983D2B92.0.3CC -I. -
      I- -I. -o m m.o a.o b.o c.o -l++
+ cppC=/home/porsche/gms/wrk/nmake/312/solaris/lib/cpp
/tools/sun_workshop/SUNWspro/bin/cc -Xs -xs -xildoff  -Wl,-L/apps/tools/sparc5/C++/3.0.3  -o
      m  -O m.o a.o b.o c.o -lg2++ -lGA -lGraph -l++ -lC
Instantiating List<char>...
Instantiating List<double>...
Instantiating lnnk_ATTLC<char>...
Instantiating lnnk_ATTLC<double>...
```

Example 2

This example demonstrates a couple of important features of the build log serialization feature: support for recursive makes, and support for coshell(1). This example sets up a build hierarchy that looks like this:

```
Makefile
demoa/Makefile
demob/Makefile
democ/Makefile
democ/democ1/Makefile
```

Contents of Makefile

```
.INIT : .MAKE
        print top-level: init output from nmake
:MAKE: demoa demob democ
.DONE : .MAKE
        print top-level: done output from nmake
```

Contents of demoa/Makefile

```
TARGETS = a1 a2 a3 a4 a5 a6
all : $(TARGETS)
$(TARGETS) :
        : $(<) begin; uname -a; sleep 3; : $(<) end
.INIT : .MAKE
        print demoa: init from nmake
.DONE : .MAKE
        print demoa: done from nmake
```

Contents of demob/Makefile

```
TARGETS = b1 b2 b3 b4 b5 b6
all : $(TARGETS)
$(TARGETS) :
        : $(<) begin; uname -a; sleep 3; : $(<) end
.INIT : .MAKE
        print demob: init from nmake
.DONE : .MAKE
        print demob: done from nmake
```

Contents of democ/Makefile

```
:MAKE:
.INIT : .MAKE
        print democ: init from nmake
.DONE : .MAKE
        print democ: done from nmake
```

Contents of democ/democ1/Makefile

```
TARGETS = c11 c12 c13 c14 c15 c16
all : $(TARGETS)
$(TARGETS) :
        : $(<) begin; uname -a; sleep 3; : $(<) end
.INIT : .MAKE
        print democ1: init from nmake
.DONE : .MAKE
        print democ1: done from nmake
```

Configure for concurrent builds, concurrent recursive makes, enable distributed builds, and run nmake

```
$ export NPROC=12
$ export recurse=3
$ export COEXPORT=NPROC:recurse
$ export COSHELL=coshell
$ rm -f makelog
$ nmakelog
```

In the build output, note that output from multiple actions in the same recursive make, as well as output from multiple concurrent recursive makes, is intermixed. Also, note that the output of uname -a indicates that shell action blocks are being sent to other machines in the network for remote execution, as requested.

Output

```
top-level: init output from nmake
demoa:
demob:
demob: init from nmake
democ:
democ: init from nmake
demoa: init from nmake
+ : b2 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : b4 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : b1 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : b3 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
democ/democ1:
democ1: init from nmake
+ : b2 end
+ : b4 end
+ : b1 end
+ : b3 end
+ : b5 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : a1 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : c11 begin
+ : b6 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : b5 end
+ : a1 end
+ : a2 begin
+ : c12 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : c11 end
+ : b6 end
demob: done from nmake
+ : a3 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : c13 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : a2 end
+ : c12 end
+ : a4 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : c14 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : a3 end
+ : c13 end
+ : a5 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : c15 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : a4 end
+ : c14 end
+ : a5 end
+ : a6 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : c15 end
+ : c16 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : a6 end
+ : c16 end
demoa: done from nmake
democ1: done from nmake
democ: done from nmake
top-level: done output from nmake
```

makelog contains the serialized build log. As in example 1, output from each shell block appears together in the log. In addition, in this example output from each recursive make, itself containing serialized individual action blocks, also appears together. For example, output of action blocks executed in recursive make demob/Makefile, whose targets begin with the letter b, appears together. This approach extends to the case of multiple levels of recursion: the serialized output of democ/democ1/Makefile appears together with, and properly sequenced within the output of democ/Makefile. Note that output from nmake itself (as opposed to output of shell action blocks) is serialized as expected. Lines such as ``demob: init from nmake'' and ``demob: done from nmake'' illustrate this feature. The example also demonstrates serialization of output of a distributed build using coshell(1).

Contents of makelog

```
top-level: init output from nmake
demob:
demob: init from nmake
+ : b2 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : b2 end
+ : b4 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : b4 end
+ : b1 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : b1 end
+ : b3 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : b3 end
+ : b5 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : b5 end
+ : b6 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : b6 end
demob: done from nmake
demoa:
demoa: init from nmake
+ : a1 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : a1 end
+ : a2 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : a2 end
+ : a3 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : a3 end
+ : a4 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : a4 end
+ : a5 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : a5 end
+ : a6 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : a6 end
demoa: done from nmake
democ:
democ: init from nmake
democ/democ1:
```

```
democ1: init from nmake
+ : c11 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : c11 end
+ : c12 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : c12 end
+ : c13 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : c13 end
+ : c14 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : c14 end
+ : c15 begin
+ uname -a
SunOS yesac 5.5.1 Generic_103640-17 sun4m sparc SUNW,SPARCstation-20
+ sleep 3
+ : c15 end
+ : c16 begin
+ uname -a
SunOS jaguar 5.5.1 Generic_103640-09 sun4u sparc SUNW,Ultra-Enterprise
+ sleep 3
+ : c16 end
democ1: done from nmake
democ: done from nmake
top-level: done output from nmake
```

The Probe Tool

A

This appendix provides an introduction to the probe tool as it relates to nmake and nmake cpp.

Overview

The **probe** tool is used by nmake and nmake cpp to determine the configuration for the current processor and C compiler. probe determines the configuration information only once and reuses the saved information each time nmake or cpp is invoked.

For nmake, the probe information file contains nmake statements describing the C language environment (such as full compiler path name, alternate preprocessor flags, dialect (cross-compiler, C++, etc.), include directory names, and library directory names). For cpp, the probe information file contains cpp directives describing the hardware environment and the C compiler attributes (such as operating system type, operating system version, hardware type, C language keywords, and preprocessing attributes).

When nmake or cpp is invoked the first time for a C compiler, a new information file is generated. The information file is created when the probe program executes a shell script named probe in the directory where the information file is to reside. The probe shell script consists of many tests that search for the presence of features and gather information about the development environment.

Normally, probe does not require human intervention. However, there are some situations, usually involving either non-standard cross-compilers or local probe generation options, where the probe information has to be manually generated or modified. In these situations, the probe information can be modified by the nmake administrator. The easiest way to do this is through a probe_hints file.

The probe tool is general purpose in nature. This appendix discusses the probe tool only as it relates to nmake's handling of C compilers.

The probe configuration information file is generated according to the value of the nmake variable CC. The default value of CC is cc, which is the first cc in $PATH. If CC is assigned to a specific compiler (e.g., CC=clcc), a probe information file will be generated for the specific compiler. If you want to use a particular mode of a compiler to compile files, you can also specify the flags for the compiler to the CC variable (e.g., CC= clcc -ansi); an appropriate probe information file will be generated for the specified mode of the compiler.

probe is not sensitive to compiler versions. If a new compiler version is installed with the same path as the older version, probe configuration information files must be deleted manually before generating the information for the new version.

When probe generates the configuration file for a language processor, it takes the Universe and OS version as attributes to hash for the configuration file name. An nmake installation shared between machines running different versions of an operating system will generate different probe files for different OS versions. The OS attribute can be overridden by setting VERSION_OS in the environment. Universe reflects the environment used to access the compiler (att or ucb). The universe can be overridden by setting _AST_FEATURES='UNIVERSE - xxx' in the environment where xxx is the desired value.

Examining the *probe* Information Files

There are two probe information files maintained for each C compiler used with nmake: one for the cpp information and one for the nmake information.

In the following discussions, $nmake_install_root refers to the directory where all of the nmake files are installed; a $ at the beginning of a line denotes a line typed by a user.

A compiler is known to probe by its full path plus information that is keyed off the name. This information (referred to as attributes) includes the following software facilities: Preroot, Universe, Virtual Root, OS version, and the 3D File System, plus the environment variable VERSION_ENVIRONMENT. The probe information for a particular compiler is stored in a file whose name is a hashed representation of the full path to the compiler and any of the attributes listed above.

To determine the name of the probe information file for a particular C compiler, the following format should be followed:

probe -k *lang tool processor*

where *lang* is the name of the language to be probed. This argument will always be C for any type of C compiler. *tool* is either make or pp. *processor* is the name of the compiler to be probed. If the compiler name is not in $PATH, the full path must be given.

The following example shows what the full paths for the probe information files might be for the standard C compiler.

```
$ probe -k C make cc
/usr/local/lib/probe/C/make/C2C01576rucbcc
$ probe -k C pp cc
/usr/local/lib/probe/C/pp/C2C01576rucbcc
```

The basename of the probe configuration information file has the format of *XXXXXXXXPPPPPP*, where *XXXXXXXX* is a hex hash value for the fully qualified path for the language processor and its attributes and *PPPPPP* is the last six characters of the fully qualified path for the language processor with / deleted and left-filled with . for short paths.

The following is a brief description of other probe options. See the probe manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual* for additional information.

The -a option lists probe attribute definitions, such as preroot, universe, etc.

The -d option deletes the information file if the user is either the owner of the configuration information file or the language processor.

The -g option generates the information file if necessary.

The -l option is used to display the content stored in the probe information files.

The -p option is used to specify an alternate probe hierarchy root path.

The -t option is for testing. The information is generated on stdout.

The -D option is for debugging trace.

The following example shows the contents of the nmake probe information file for the GNU C compiler.

```
CC.CC = /usr/bin/gcc
CC.ALTPP.FLAGS =
CC.ALTPP.ENV =
CC.DIALECT = ANSI DOTI DYNAMIC -I-
CC.DYNAMIC = -Wl,-ashared
CC.LD = ld
CC.MEMBERS = -whole-archive
CC.MEMBERS.UNDEF = -no-whole-archive
CC.NM = nm
CC.NMEDIT = -e '/[	][TDBC][	][	]*[_A-Za-z]/!d' -e 's?.*[	][TDBC][	][	]*??'
CC.NMFLAGS =
CC.PIC = -fpic
CC.READONLY =
CC.SHARED = -G
CC.STATIC = -Wl,-aarchive
CC.STDINCLUDE = /usr/lib/gcc-lib/i386-redhat-linux/3.2.3/include /usr/include
CC.STDLIB = $(LPATH:/:/ /G) /usr/lib/gcc-lib/i386-redhat-linux/3.2.3 /usr/lib /usr/lib
CC.SUFFIX.ARCHIVE = .a
CC.SUFFIX.COMMAND =
CC.SUFFIX.LINKHEADER =
CC.LINK.OPTION =
CC.SUFFIX.OBJECT = .o
CC.SUFFIX.SHARED = .so
CC.SUFFIX.SOURCE = .c
CC.SUFFIX.STATIC =
CC.SYMPREFIX =
CC.INSTRUMENT =
CC.IMPLICIT.INCLUDE.ENABLE =
CC.IMPLICIT.INCLUDE.DISABLE =
CC.UNIVERSE = ucb
CC.VERSION_OS = Linux2.4 Red Hat Enterprise Linux ES release 3 (Taroon Update 9)
```

Probe Information—nmake

During initialization, nmake loads the contents of the probe information file for the compiler specified by $(CC). If you want to use another compiler's probe information file instead, you can set the cctype base rule variable to the name of the other compiler.

Each line in an nmake probe file contains an nmake variable. The following sections detail the meaning and possible values for these variables.

CC.CC

This variable is used to specify the full path to the compiler as determined by the setting of the CC variable. For example if CC was set to /bin/cc or just cc with /bin being the first in the shell PATH with the cc compiler, then CC.CC would evaluate to:

```
CC.CC = /bin/cc
```

NOTE:
The following nmake statements contain information about the compilation environment of the compiler set in CC.CC.

CC.ALTPP.FLAGS

This variable is set to the command-line options a compiler uses to specify an alternate preprocessor. For example, the native compiler (/usr/5bin/cc) in a SunOS environment uses the -Qpath option to specify the directory where an alternate preprocessor is located. Therefore, probe automatically generates the following line:

CC.ALTPP.FLAGS = -Qpath $(CPP:D)

CC.ALTPP.ENV

This variable is set to the environment variable a compiler uses to specify an alternate preprocessor. For example, the Alcatel-Lucent C++ 3.0.3 compiler, uses the realcppC environment variable to specify the full path to an alternate preprocessor. Therefore, probe automatically generates the following line:

CC.ALTPP.ENV = realcppC=$(CPP)

NOTE:
If both CC.ALTPP.FLAGS and CC.ALTPP.ENV are undefined, the Alcatel-Lucent nmake-supplied shell wrapper ppcc takes part in the compilation process. ppcc copies the file to /tmp and uses the nmake cpp to preprocess the file; after preprocessing, the compiler takes over the rest of the compilation.

CC.ARFLAGS

This flag is used for certain compilers, such as Sun’s C++ compilers, that support an option to generate libraries. For these compilers, this may be:

CC.ARFLAGS = -xar

CC.DIALECT

This variable, is a space-separated list of tokens that descibes various features of the compilation process.

ANSI	The compiler is an ANSI standard compiler.
C++	The compiler is a C++ compiler.
CROSS	The compiler is a cross-compiler.
DOTI	The compiler can accept a **.i** file as preprocessed input
DYNAMIC	Dynamic linking (i.e., linking at run time) is supported.
-I-	-I- is supported by the standard preprocessor.

IMPLICITC	Unless explicitly specified, the template definitions file is assumed to be the name of the associated template declaration file but with a source file suffix (say .c or .C).
PTRIMPLICIT	For *cfront*-based compilers, given the option -ptrPTR, the compiler expects the template DB to be PTR/<repository> and will create it if it doesn't exist. For example the Solaris C++ 4.2 compiler expects the template DB to be "PTR/Templates.DB" for the -ptrPTR flag.
LIBPP	The compiler already uses libpp (nmake cpp) by default.
TOUCHO	The compiler may touch **.o** files in implementing template instantiation.
VERSION	The linker can link shared library with version numbers

CC.DYNAMIC

This variable contains the flag the compiler/linker uses to requests that dynamic binding be the priority at link time, for the libraries specified via the **-l***x* option. So, for a compiler/linker that supports dynamic binding (indicated by **DYNAMIC** in the CC.DIALECT), this flag when specified in CCFLAGS will force nmake to search first for the shared version and then the archive one, as the linker does, for libraries specified with the **-l**x option. (Note, that this is nmake's default behavior, for linkers that supports dynamic binding, when static binding is not requested). This variable may be null if dynamic linking is enabled by default.

For example, the following line is generated for the SUN Solaris C compiler:

```
CC.DYNAMIC = -Bdynamic
```

CC.IMPLICIT.INCLUDE.DISABLE

This variable contains the flag(s) used by the C++ compiler to disable the support for implicit inclusion of template definitions. See CC.IMPLICIT.INCLUDE.ENABLE.

CC.IMPLICIT.INCLUDE.ENABLE

This variable contains the flag(s) used by the C++ compiler to enable the support for implicit inclusion of template definitions. For example, the following line may be generated for EDG-based C++ compilers:

```
CC.IMPLICIT.INCLUDE.ENABLE = --implicit_include -B
```

CC.INSTRUMENT

This variable contains the flag used by compilers such as Alcatel-Lucent's C++ v4.1 compiler that supports the instrument feature of tools such as purify and quantify that does link-time program analysis. For Alcatel-Lucent's v4.1, the following line would be generated:

CC.INSTRUMENT = -purify --purify -quantify --quantify

CC.LD

This variable contains the name of the loader processor or link editor. For example, the following line is generated on a Sun Workstation:

CC.LD = ld

CC.LIBOPTS

This variable contains flags supported by the compiler for incorporating optional compiler supplied libraries. For example, the following line is generated for Sun C++ compilers:

CC.LIBOPTS = -library -compat

CC.LINK.OPTION

This variable contains the link option used to specify nonstandard library names, such as libnss_dns.1 on HP-UX. The variable will not be set if no such option exists for the compiler. The following example shows the setting generated for HP native compilers on HP-UX. Specifying a prerequisite of -lfullname will be translated to the command line option -l:fullname.

CC.LINK.OPTION = -l:

CC.MEMBERS

This variable contains the linker flag to enable loading of all archive library members or a shell command that may be used in the action block of an assertion to extract the symbols, preceded by the -u flag, of its prerequisite archive library. For example, the following line is generated on a Sun Workstation:

CC.MEMBERS = -zallextract

CC.MEMBERS.UNDEF

This variable contains the linker flag to disable loading of all archive library members. For example, the following line is generated on a Sun Workstation:

CC.MEMBERS.UNDEF = -zdefaultextract

CC.NM

This variable contains the processor's name for printing symbol name list. For example, the following line is generated on a Sun Workstation:

CC.NM = nm

CC.NMEDIT

This variable contains the edit pattern that is used in the stream editor processor (sed) to edit NM output. For example, the following line is generated on a Sun Workstation:

```
CC.NMEDIT = -e '/[     ][TDBC][       ]/!d' -e 's?.*\
                        [       ][TDBC][][     ]*??'
```

CC.NMFLAGS

This variable contains the flags used by CC.NM.

CC.PIC

This variable contains the flag that the language processor uses to generate position-independent code for use in shared libraries. This flag in CCFLAGS is the trigger to build a shared library via the :LIBRARY: operator. Say, CCFLAGS += $$(CC.PIC). The following line is generated for the C compiler on a Sun Workstation:

CC.PIC = -Kpic

CC.PRELINK

This contains the flag that enables EDG-based C++ compilers like Alcatel-Lucent’s v4.1 to implement template instantiation with nmake’s viewpathing. The following line is generated for Alcatel-Lucent’s C++ 4.1 compiler:

CC.PRELINK = --prelink_copy_if_nonlocal

CC.PREROOT

Preroot is a software facility commonly found on UTS machines (i.e. Amdahl). This variable contains the physical name of the preroot. For example, the value for this variable on an Amdahl machine is:

CC.PREROOT = /native

Virtual root is a user-level counterpart to the UTS preroot used by some Sun compilers. CC.PREROOT will be set to the virtual root directory for these compilers.

CC.READONLY

This variable contains the command-line option to the cc wrapper for specifying that the data segment should be read only. For example, the following line will be generated for the C compiler on a Sun Workstation:

CC.READONLY = -R

CC.REPOSITORY

For *cfront*-based compilers, this flag refers to the name of the repository that the compiler creates in the template database area. For example, the Solaris C++ 4.2 compiler uses the name Templates.DB. So, the following line would be generated for that compiler:

CC.REPOSITORY = Templates.DB

CC.REQUIRE.*name*

This variable contains libraries required by lib*name* which will be bound and included as prerequisite libraries when -l*name* is a prerequisite. See CC.REQUIRE.*name* in the *Binding Library Prerequisites* section of chapter 5.

CC.REQUIRE.++

Some C++ scripts may implicitly expand -l++ to a number of libraries. As such, this variable contains the libraries (including -l++) that are required to be linked in with -l++. For some C++ language processors, CC.REQUIRE.++ may evaluate to -lg2++ -lGA -lGraph -l++.

CC.SHARED

This variable contains the flag that directs the link editor to generate a shared object. For most compilers this variable is:

CC.SHARED = -G

CC.SHARED.ALL

This variable contains a flag for the link editor. All members of the libraries following this argument are linked in the generated target. This flag is used to make a dynamic library from a static library and include all the symbols from the static library.

CC.SHARED.ALL = -Bwhole-archive

CC.SHARED.UNDEF

This variable contains a flag for the link editor. Only members that resolve undefined symbols in the libraries following this argument are linked in the generated target.

CC.SHARED.UNDEF = -Bno-whole-archive

CC.STATIC

This variable contains the flag the compiler/linker uses to requests that static binding be done at link time, for the libraries specified via the **-l***x* option. Even for a compiler/linker that supports dynamic binding (indicated by **DYNAMIC** in the CC.DIALECT), this flag when specified in CCFLAGS will force nmake to search ONLY for the archive one, as the linker does, for libraries specified with the **-l**x option. For example, the following line is generated for the SUN Solaris C compiler:

CC.STATIC = -Bstatic

CC.STDINCLUDE

This variable contains a list of all the standard include directories searched by the compiler. For most compilers, the value for this variable is:

CC.STDINCLUDE = /usr/include

CC.STDINCLUDE.OMIT

This variable contains a list of standard include directories that should not be passed to the compiler with -I flags. For most systems it will not be set. Some Sun systems may get a value such as the following:

CC.STDINCLUDE.OMIT = /opt/SUNWspro/prod/include/CC/std

CC.STDLIB

This variable contains a list of all the standard library directories searched by the compiler. A typical example would be:

CC.STDLIB = $(LD_LIBRARY_PATH:/:/ /G) /lib /usr/lib /usr/local/lib

NOTE:
The following group of CC.SUFFIX variables are set in the "make" part of the probe file. They define the suffixes of different file types that the compiler being probed understands or can handle.

CC.SUFFIX.ARCHIVE

Defines the suffix of an archive library file. For most compilers:

CC.SUFFIX.ARCHIVE=.a

CC.SUFFIX.COMMAND

Defines the suffix for an executable. For most compilers this is null, i.e.:

CC.SUFFIX.COMMAND=

CC.SUFFIX.DLL

Defines the suffixes for supplementary files created when generating .dll dynamic libraries.

CC.SUFFIX.DLL=.exp .def .map

CC.SUFFIX.DYNAMIC

Defines the suffix for dynamic runtime libraries when different from compile time shared libraries.

CC.SUFFIX.DYNAMIC=.dll

CC.SUFFIX.LINKHEADER

Contains a list of system headers with versions ending with SUNWCCh.

CC.SUFFIX.LINKHEADER=algorithm bitset cassert cctype errno ...

CC.SUFFIX.OBJECT

Defines the suffix for an object created by the compiler. For most C compilers:

CC.SUFFIX.OBJECT=.o

CC.SUFFIX.SHARED

Defines the suffix for a shared object file name. On an HP workstation:

CC.SUFFIX.SHARED=.sl

CC.SUFFIX.SOURCE

Defines the suffix of the source file the compiler can deal with. For most C compilers

CC.SUFFIX.SOURCE=.c

CC.SUFFIX.STATIC

Defines the suffix of a static link library the compiler can make. On most Sun workstations:

CC.SUFFIX.STATIC=.sa

CC.SYMPREFIX

This variable contains the symbol prefix in the library symbol table. For example, the following line is generated on a Sun Workstation:

CC.SYMPREFIX = _

CC.UNIVERSE

This variable contains the name of the universe facility (i.e., on Pyramid) or the environment used to access the compiler (att or ucb). It is used to determine the probe file name hash and should not be modified. To override set _AST_FEATURES='UNIVERSE - xxx' in the environment where xxx is the desired value.

CC.UNIVERSE=ucb

CC.VERSION_OS

This variable contains the OS version. It is used to determine the probe file name hash and should not be modified. Set VERSION_OS in the environment to override. For example, the following line is generated for a Solaris 10 system:

CC.VERSION_OS=SunOS5.10

Probe Information - *cpp*

During initialization, nmake cpp loads the contents of the probe information file for the compiler specified using the -I-D command-line option. Each line in a cpp probe configuration information file contains a cpp directive. The following sections detail the meaning and possible values for these directives. The configuration information file contains four sections: comments, predefined definitions, assertions, and pragmas.

Comments

The first line in the cpp probe configuration information file contains the name of the language processor (e.g., cc program) enclosed in a C-style comment.

Predefined Definitions

The next section contains all of the predefined definitions for the specified language processor. These definitions are determined by testing all the standard definitions in the pp.def file and those extracted from the language processor file. The ppsym program is used to extract strings from the language processor that might possibly be definition names (similar to the way the UCB strings command works).

The following example shows what the definition section might look like for a C++ compiler in a Sun computer environment.

```
#pragma pp:predefined
#define BSD 1
#define c_plusplus 1
#define sparc 1
#define sun 1
#define unix 1
#define __BSD__ 1
#define __BUILTIN_VA_ARG_INCR 1
#define __cplusplus 1
#define __c_plusplus__ 1
#define __sparc__ 1
#define __sun__ 1
#define __unix__ 1
#pragma pp:nopredefined
```

The first line, #pragma pp:predefined, causes the definitions that follow it to be marked as predefined. The preprocessor gives a warning if a predefined #define is referenced outside of a cpp directive. The last line, #pragma pp:nopredefined, resets the mode.

The first set of definitions is non-ANSI (no prefixed underscores). The second set of definitions is ANSI compatible. If the ANSI dialect is used, the predefined definitions without a leading underscore are ignored.

Assertions

This section contains definitions that describe the hardware, operating system, and language processor. The following example shows what the assertion section might look like for a C++ compiler in a Sun computer environment.

```
#define #system(unix)
#define #release(bsd)
#define #release(BSD)
#define #version()
#define #architecture(sparc)
#define #model()
#define #machine(sun)
#define #addressing()
#define #preroot()
#define #universe(att)
#define #dialect(C++)
#define #dialect(dynamic)
```

#define #system()

The name of the operating system (e.g. unix).

#define #release()

The release name of the operating system (e.g. att, uts, bsd, svr4, V9, xinu).

#define #version()

The version of the operating system.

#define #architecture()

The architecture of the machine (e.g. sparc, u3b, ibm, vax).

#define #model()

The model of the machine (e.g. 20, 370).

#define #machine()

Bundled machine name (e.g. sun, cray, datageneral, unixpc).

#define #cpu()

Set to the number of CPUs on the machine.

#define #addressing()

The addressing mode used for PC compiler implementations.

#define #preroot()

Set to the same value as CC.PREROOT in the nmake probe information file.

#define #universe()

Set to the same value as CC.UNIVERSE in the nmake probe information file.

#define #dialect()

Generate one of: ansi, C++, cross, dynamic.

Pragmas

This section contains pragmas for passing configuration information to the C preprocessor.

The following example shows what the pragma section might look like for a C++ compiler in a Sun computer environment.

```
#pragma pp:plusplus
#pragma pp:reserved asm=GROUP const enum inline \
signed void volatile
#pragma pp:splicecat
#pragma pp:compatibility
#pragma pp:plussplice
#pragma pp:hostdir
#pragma pp:include "/usr/local/include/CC3.0.2"
#pragma pp:linetype
#pragma pp:map "/#(pragma )?ident>/" "/#(pragma )?/###/"
#pragma pp:map "/#pragma lint:/" ",#pragma lint:(.*),\
##/*\1*/,u"
#pragma pp:map "/#(pragma )?sccs>/"
```

The following is a brief description of the pragmas that may be generated by probe. See the cpp manual page in the *Alcatel-Lucent nmake Product Builder Reference Manual* for additional information on what the pragmas do.

#pragma pp:plusplus

Preprocess for C++ mode.

#pragma pp:reguard

Output all #define statements to re-guard included header files.

#pragma pp:reserved

List of reserved words for the compiler. The reserved words with =GROUP appended are those that have a group syntax (id [(...)] + [{ ... }]).

#pragma pp:pluscomment

Enable C++ comments.

#pragma pp:splicecat

<newline> line splicing may be used to concatenate tokens in macro definitions.

#pragma pp:stringspan

Allow <newline> in strings.

#pragma pp:compatibility

Preprocess in compatibility dialect for non-ANSI compilers.

#pragma pp:transition

Preprocess for transition mode.

#pragma pp:strict

Preprocess for strict mode.

#pragma pp:spaceout

In the ansi dialect for the standalone cpp this pragma causes input spacing to be copied to the output. The default for the ansi dialect is to place a single space between each output token.

#pragma pp:truncate *n*

Truncate identifiers to *n* characters if specified; otherwise, to 8 characters for compatibility with non-flexname compilation systems.

#pragma pp:id "*string*"

Adds the characters in *string* to the set of characters that are allowed in an identifier name. For example, #pragma pp:id "$"allows to be used in identifier names.

#pragma pp:cdir

Allow C++ **extern "C" { ... }** include wrapping. Include files after this pragma are marked as C source. Equivalent to -I-C command-line option.

#pragma pp:plussplice

Allows for non-standard backslash. For the C++ code:

```
// comments \
static int x = 3;
```

the second line is part of the comment when this pragma is specified.

#pragma pp:hostdir

Mark the following files as hosted.

#pragma pp:include *directory*

One for each standard directory for the compiler (not needed for /usr/include).

#pragma pp:standard *directory*

If /usr/include is not used by the compiler (i.e., cross-compilation systems), list the standard directory here.

#pragma pp:lineid *line-directive*

Change the directive name for line number control output directives to *line-directive*. The line number control output directives are of the form #*line-directive line "file"*. The defaults are **line** if *line-directive* is omitted, and null if the **pp:lineid** pragma and **–D–L** option are both omitted.

#pragma pp:linetype

Causes the include type to appear as the third argument in line sync output. The include type syntax is defined and required by Sun compilers.

#pragma pp:noallmultiple

If the noallmultiple pragma is set then subsequent #include references to the header named by filename are ignored unless the header contains a #pragma multiple directive that was processed during the first inclusion of the header.

#pragma pp:map

The following maps may be generated. Certain directives that may be different by using the native preprocessors can be mapped to the ones that the new preprocessor handles.

```
#pragma pp:map "/#(pragma )?ident>/" "/#(pragma )?/###/"
#pragma pp:map "/#(pragma )?import>/" "/#(pragma )\
                    ?import(.*)/__IMPORT__(\1)/"
#macdef __IMPORT__(x)
#pragma pp:noallmultiple
#include x
#pragma pp:allmultiple
#endmac
#pragma pp:map "/#include_next>/" "/.*/#include <...>/"
#pragma pp:map "/#pragma lint:/" ",#pragma lint:\
(.*),##/*\1*/,u"
 #pragma pp:map "/#(pragma )?sccs>/"
```

ANSI asm processing

ANSI asm processing allows multiline assembly statements that need not be enclosed in quotation marks (""). In order to pass the assembly code through the preprocessor correctly, the following lines are inserted into the probe file (ANSI compilers only).

```
#macdef asm
#pragma pp:spaceout
#undef asm
asm
#endmac
```

Using Distinct Probe Configuration Files

The VERSION_ENVIRONMENT environment variable contains a colon separated list of environment variables for probe to use for generating the configuration filename hash. The values of the specified variables will influence the filename hash and can be used to generate and reference distinct probe configuration files for different conditions. For example, distinct probe files can be used for different OS versions by exporting the following variables in the environment:

```
export OS_VERSION=$(uname -sr)
export VERSION_ENVIRONMENT=OS_VERSION
```

Multiple variables can be specified to further distinguish the probe files.

```
export VERSION_ENVIRONMENT=OS_VERSION:PROJECT:RELEASE
```

The Probe Hints File

The probe_hints file provides a convenient and flexible way to specify local installation overrides for selected probe variables. Using this approach, Alcatel-Lucent nmake administrators provide a probe_hints file that is separate from the automatically generated probe file. Probe consults the probe_hints file whenever it generates a probe file, and uses the information in this file to guide the probe.

Use of a probe_hints file has a number of advantages over direct editing of the generated probe information file:

1. Since probe combines the information from the hints file with the automatically derived probe information, probe can continue to be responsible for most of the information in the probe file even if the user needs to override a couple of individual probe variables.
2. Alcatel-Lucent nmake installations and upgrades are easier and less error prone since the probe_hints file may simply be transferred to the new installation. There is no need to generate a probe file, obtain the hashed path value, change the permissions, and manually modify the generated file, for each version of each compiler.
3. Since the probe file is not manually edited, we eliminate a source of run-time errors.

4. Since the hints file is automatically consulted by all future probe invocations, we avoid the problem of user-triggered probe invocations inadvertently generating incorrect probe data.

Currently, the probe hints file is supported for nmake configuration information only, not for cpp information.

Using a Probe Hints file

In order to use a probe_hints file, a user must:

1. Write a valid probe_hints file.
2. Put the probe_hints file in the right place.

Content of Probe Hints file

The probe_hints file is a user-provided shell script. It overrides selected probe variables by setting shell variables to the preferred values. The probe_hints script may, as necessary, use any facilities available to obtain information from the host environment, file system, and 3rd party compilation tools. Additionally, probe_hints may refer to the automatically generated probe values—at the time the probe_hints file is run, the automatic probe has already completed and the probe variables are made available to probe_hints through mapped shell variables. There is a simple one-to-one correspondence between probe variables and mapped shell variables. This mapping is described in more detail in a following section.

The following tips may be helpful in writing a useful probe_hints file:

- The probe hints file works like a filter program. It imports mapped probe variables from the probe shell script and outputs revised shell variables to the calling probe shell script.
- In theory, the probe hints file can do anything. But the only effective result is the changed values of mapped probe variables which are recognized by the calling probe shell script.
- In practice, the probe hints file should be designed as a filter program. It should generate correct probe settings from the provided automatic probe settings and the user's specific environment.
- Probe hints file should only deal with those “incorrect” automatic probe settings. Already correct settings should not be touched.
- The probe hints file is run in a subprocess of the probe process. It should be a typically good shell script. For example, it should not occupy too many system resources.
- The probe hints file should not output any message to stdout or stderr. It should only use mapped shell variables as the interface to the outside world.

- The probe hints file should not call exit. Doing so may result in null values for all the probe variables.
- To include an unexpanded nmake variable in the value of a probe variable, escape the variable name with a backslash. For example, \$(VAR).
- All compilers use the same probe hints file. It's often the most important to identify what the current compiler is at the beginning of probe hints file.

Probe Hints Examples

Setting up the mapped shell variables and reading the modified values is performed automatically by the calling probe shell. The probe_hints script need only set the selected variable to the desired value to effect the change. Consequently, the simplest possible probe_hints script is a single variable assignment:

```
CC_STDINCLUDE="/tools/include/vendorA $CC_STDINCLUDE /usr/include"
```

This example uses the CC_STDINCLUDE mapped shell variable, which is mapped from the CC.STDINCLUDE probe variable. This example illustrates that the script may refer to the values of existing mapped variables.

A more realistic example sets values conditioned on the value of the specific compiler being probed:

```
case "$CC_CC" in
   */gcc | */g++ )
              CC_STDINCLUDE="/tools/include/gcc $CC_STDINCLUDE"
              ;;
   /tools/vendorA/v1.[23].*/bin/CC )
              CC_STDINCLUDE="/tools/include/vendorA_CC $CC_STDINCLUDE /usr/
include"
              ;;
   esac
```

Location of Probe Hints File

An Alcatel-Lucent nmake administrator should store the probe_hints file in the nmake installation area ($nmake_install_root/lib/probe/C/make).

An nmake user can store the probe_hints file in his/her localprobe area (<effective_localprobe_root>/lib/probe/C/make). effective_localprobe_root is specified by the nmake localprobe variable.

The basename of the probe hints file must be probe_hints. Its permission should be set to 755.

The specific search rules are:

1. When localprobe is not set the probe hints file is searched in the directory where the invoking probe shell exists. That is usually in an nmake installation area: $nmake_install_root/lib/probe/C/make.
2. When localprobe is set the probe hints file is first searched in the effective localprobe directory: <effective_localprobe_root>/lib/probe/C/make. The effective localprobe directory is the directory where the local probe information file is stored or will be generated. If no probe hints file can be found in the effective localprobe directory, nmake will search in the directory where the invoking probe shell exists. This is usually in an nmake installation area: $nmake_install_root/lib/probe/C/make.

Evaluation of the Probe Hints file

If the probe_hints file is located according to the above rules and is executable, it is evaluated whenever a probe is performed. In addition, probe information is automatically regenerated whenever the corresponding probe hints files is updated; touching the probe_hints file has the same effect as touching the probe shell script. The evaluation of the probe_hints file is safe; a malformed probe_hints file cannot interfere with normal probe processing or corrupt the probe information file.

As a diagnostic aid, probe will output an informational message if a probe hints file is found. In addition, in the probe information file, a comment is attached to each probe setting which is modified by the probe hints file, indicating which probe hints file has changed this setting.

Mapping between Probe Variables and Probe Hints Shell Variables

There is a uniform mapping between probe variables and shell variables. Taking the probe variable name and substituting ‘_’ for ‘.’ seems to be the simplest scheme. Following is a table of all exported probe variables and their corresponding shell variables.

Table A-1. Mapping between Probe Variables and Shell Variables

Probe Variable	Probe Hints Shell Variable
CC.ALTPP.ENV	CC_ALTPP_ENV
CC.ALTPP.FLAGS	CC_ALTPP_FLAGS
CC.ARFLAGS	CC_ARFLAGS
CC.CC	CC_CC
CC.DIALECT	CC_DIALECT
CC.DYNAMIC	CC_DYNAMIC

Table A-1. Mapping between Probe Variables and Shell Variables—*Continued*

Probe Variable	**Probe Hints Shell Variable**
CC.IMPLICIT.INCLUDE.DISABLE	CC_IMPLICIT_INCLUDE_DISABLE
CC.IMPLICIT.INCLUDE.ENABLE	CC_IMPLICIT_INCLUDE_ENABLE
CC.INSTRUMENT	CC_INSTRUMENT
CC.LD	CC_LD
CC.LIBOPTS	CC_LIBOPTS
CC.LINK.OPTION	CC_LINK_OPTION
CC.MEMBERS	CC_MEMBERS
CC.NM	CC_NM
CC.NMEDIT	CC_NMEDIT
CC.NMFLAGS	CC_NMFLAGS
CC.PIC	CC_PIC
CC.READONLY	CC_READONLY
CC.REPOSITORY	CC_REPOSITORY
CC.SHARED	CC_SHARED
CC.SHARED.ALL	CC_SHARED_ALL
CC.SHARED.UNDEF	CC_SHARED_UNDEF
CC.STATIC	CC_STATIC
CC.STDINCLUDE	CC_STDINCLUDE
CC.STDLIB	CC_STDLIB
CC.SUFFIX.ARCHIVE	CC_SUFFIX_ARCHIVE
CC.SUFFIX.COMMAND	CC_SUFFIX_COMMAND
CC.SUFFIX.DYNAMIC	CC_SUFFIX_DYNAMIC
CC.SUFFIX.DLL	CC_SUFFIX_DLL
CC.SUFFIX.LINKHEADER	CC_SUFFIX_LINKEADER
CC.SUFFIX.OBJECT	CC_SUFFIX_OBJECT
CC.SUFFIX.SHARED	CC_SUFFIX_SHARED
CC.SUFFIX.SOURCE	CC_SUFFIX_SOURCE
CC.SUFFIX.STATIC	CC_SUFFIX_STATIC
CC.SYMPREFIX	CC_SYMPREFIX

Manually Generating *probe* Information

The probe information files should be manually generated or updated only as a last resort. When probe files are manually changed, probe is no longer responsible for the maintenance of that file. It is better to use a probe hints file, or to try to place a shell wrapper around the language processor to correct the need for manual modification.

To generate the probe information file manually, the first step is to determine the name for the probe file using the -k option. Then, if possible, copy an existing probe information file to a file with that name and start from here.

The new file must be owned by the owner of the probe executable ($nmake_install_root/lib/probe/probe). Also, probe files can be generated in an alternate probe directory by specifying the -p option to probe; however, it is assumed that the full probe directory hierarchy already exists upon calling probe. Therefore, it is suggested that alternate probe directories only be used in conjunction with the localprobe base rules variable. The file must have permissions of 644 in order to edit the file. probe will not overwrite any probe files that have write permissions for the owner of the file.

Troubleshooting

Permissions

probe is sensitive to the permissions of the probe directories and files. All the directories and files under $nmake_install_root/lib/probe should be owned by the same user (and should not be owned by the root). The following is a sample listing of the directory structure under $nmake_install_root/lib/probe:

```
total 121
drwxr-xr-x 4 gsf 512 Sep 25 07:27 C
-rwsr-xr-x 1 gsf 114688 Dec 21 14:07 probe
C:
total 2
drwxr-xr-x 2 gsf 512 Dec 7 15:06 make
drwxr-xr-x 2 gsf 1024 Dec 21 14:03 pp
C/make:
total 13
-r--r--r-- 1 gsf 227 Dec 19 14:46 0DAAFFA8.bincc
-r--r--r-- 1 gsf 281 Dec 30 13:26 FED1C231bingcc
-rwxr-xr-x 1 gsf 8606 Dec 7 15:06 probe
C/pp:
total 49
-r--r--r-- 1 gsf 684 Dec 19 14:48 0DAAFFA8.bincc
-r--r--r-- 1 gsf 759 Nov 1 10:46 FED1C231bingcc
-rw-r--r-- 1 gsf 3087 Dec 7 14:22 pp.def
```

```
-rw-r--r-- 1 gsf 1115 Dec 4 11:19 pp.key
-rwxr-xr-x 1 gsf 24576 Dec 21 14:03 ppsym
-rwxr-xr-x 1 gsf 12584 Dec 21 10:39 probe
```

The $nmake_install_root/lib/probe/probe executable requires the setuid bit on (third line). Therefore, probe should not be installed using the root ID.

not a C compiler: ...

This message occurs when the C compiler does not return exit code 0 for success and non-zero for failure, or does not correctly handle the -c option. This problem can be corrected by writing a shell script on top of the compiler to correct the compiler's deficiencies.

C probe failed -- default assumed

This message is usually caused by permission problems in the probe directories or files. There will usually be another error message output in conjunction with this one. See the *Permissions* section, above.

cannot generate probe key

This message is usually caused by permission problems in the probe directories or files; or cannot find $nmake_install_root/lib/probe/*lang*/*tool* directory. See the *Permissions* section, above.

Probing language processor ...

This is the normal message that appears when a probe file is being generated. This message occurs when no probe configuration information file exists or the probe facility was updated.

probe hints file found

This is the normal message that appears when a probe hints file has been found. The probe hints file may modify the results of the automatic probe.

Debug mode of probe

The probe scripts (e.g. $nmake_install_root/lib/probe/C/[make|pp]/probe) can be run standalone in debug mode by using the -d flag to examine the probing steps.

alternate probe directory gid mismatch

This message occurs when the gid of the executing user is different than the gid of an alternate probe directory.

alternate probe directory not writable by gid

This message occurs when the alternate probe directory does not have group write permission.

probe shell for ... obsolete, attention required

This message occurs when the probe executable is newer than a previously modified probe shell. It indicates that the probe shell being used may not be compatible with the current version of Alcatel-Lucent nmake, and may need to be modified.

Makefile Construct List B

The following table provides descriptions of the symbols that are used when writing Alcatel-Lucent nmake makefiles. In the descriptions, *lhs* and *rhs* are the lists of atoms appearing on the left-hand and the right-hand side of the operator, respectively, and *action* is the action that is specified in the action block of an assertion.

Item	Description
Assertion Operators	
:	Dependency Assertion Operator
::	Source Dependency Assertion Operator
":*identifier*:"	User-defined assertion operator where *identifier* can be any character string (A–Z upper or lower case).
:ALL:	Causes the rhs targets, if they exist, and all :: targets to be made when no command-line targets are specified.
:COMMAND:	The lhs is a variable whose value is the result of shell command substitution on the action with respect to prerequisites on the rhs.
:COPY:	The lhs is a copy of the rhs file.
:FUNCTION:	Define lhs as a functional variable.
:INSTALL:	Install files. File on rhs is installed as file on lhs.
:INSTALLDIR:	The files in the rhs are installed into the directory on the lhs by the install common action.
:INSTALLMAP:	Install selected files in directory.
:INSTALLPROTO:	Install proto output files.

Item	Description
:JOINT:	The *action* jointly builds all targets on the lhs with respect to prerequisites on the rhs.
:LIBRARY:	This operator creates an archive library on the lhs from the specified source files on the rhs. A shared library may also be built by specifying CCFLAGS += $$((CC.PIC).
:LINK:	The lhs files are hard links to the rhs file.
:MAKE:	If the rhs is a directory, a recursive nmake is run in that directory. If rhs is not specified, a recursive nmake is run in all subdirectories containing a makefile from the list $(MAKEFILES:/:/ /G). If rhs is a makefile, nmake is run on the file.
:PACKAGE:	Find files and libraries in software package.
:READONLY:	The rhs compilation is to place tables and/or data in a read-only text object section.
:SAVE:	The rhs files are generated but are still to be saved by the save-oriented common actions (cpio, pax, etc.).
:cc:	The rhs files are compiled using $(cc) instead of $(CC).
Assignment Operators	
=	Define a variable.
==	Define a variable as an implicit state variable.
+=	Add to the definition of a variable.
:=	Define a variable and all contained variables.
&=	Add an auxiliary (hidden) value to a variable.
Arithmetic Operators	
!	Logical not
!=	Not equal
%	Modulo
&	Bitwise AND
&&	Logical AND
*	Multiplication
+	Addition
–	Subtraction
/	Division
<	Less than
<<	Bit shift left
<=	Less than or equal to

Item	**Description**
==	Logical equal
>	Greater than
>=	Greater than or equal to
>>	Bit shift right
?:	C-style operator that can be used in integer expressions and has the syntax: *expression ? true_expression : false_expression* such that if *expression* is true, *true_expression* is evaluated. If *expression* is false, *false_expression* is evaluated.
^	Bitwise exclusive OR
|	Bitwise inclusive OR
||	Logical OR
~	Bitwise one's compliment
Variables	
$( )	Expand the parenthesized variable
$$( )	Delay the expansion of the parenthesized variable once.
$(*var:edit_op...*)	Expand the variable specified in *var* using edit operators specified in *edit_op*.
( )	Include the current value of the variable in the statefile when this appears on the assertion line.
.*NAME*.	Variable *NAME* used in default rules.
Automatic Variables	
$(!)	The list of all implicit and explicit file prerequisites of the current target. Implicit file prerequisites are generated by the language-dependent scan.
$(#)	When used in a .FUNCTION, $(#)expands to the number of actual arguments.
$(%)	The *stem* of the current metarule match. Target for non-metarule targets.
$(&)	The list of all implicit and explicit state variable prerequisites of the current target. Implicit state variable prerequisites are generated by the language-dependent scan.
$(-)	The current option settings suitable for use on nmake command lines. The options are listed using the -o option-by-name style; only those option settings different from the default are listed.
$(*)	The list of all explicit file prerequisites of the current target.

Item	Description
$(+)	The current option settings suitable for use by the nmake set command. Only those option settings different from the default are listed.
$(+*option*)	The current setting for the named *option*.
$(...)	Represents all the atoms, rules, and state variables used when making .MAKEINIT.
$(;)	When the current target is a state variable, $(;) expands to the value of the state variable.
$(<)	The current target name.
$(=)	The list of command-line script arguments and assignments for variables that are prerequisites of the atom .EXPORT.
$(>)	The list of all explicit file prerequisites of the current target that are out of date with the target or new, or in metarules the primary metarule prerequisite. $(>) is always non-null for triggered metarule actions but otherwise may be null (even if the current target action has triggered).
$(?)	All prerequisites of the current or specified target. This value includes all implicit and explicit state and file prerequisites. It is equivalent to $(!) and $(&) together.
$(@)	The action for the current target.
$(~)	The list of all explicit prerequisites of the current target.
$(-*option*)	1 if the named *option* is set and nonzero, otherwise null.
Programming Statements	
/* */	Comment delimiters.
#	Restricted comment indicator (appears after a statement on a line).
local	Allows local variable definition.
let	Programming integer variable assignment.
if/elif else/end	if-then-else programming construct.
for/end	Looping construct.
while/end	Conditional looping construct.
break	Immediate exit from for loops or while loops.
error	Output an error message and severity level.
eval/end	Causes expressions to be expanded again.
set	Command-line options set within the makefile.
ignore	Shell script that causes nmake to ignore the exit code of the command that follows.

Item	**Description**
silent	Shell script that causes the command that follows not to be echoed.
include [–] *path*	Include the file specified by *path*. If – is present, the file is optionally included (no warning is generated if the file is not found).
print *msg*	Print *msg* to stdout; same as error 0.
return [*expression*]	Return from the action with results specified by *expression*: omitted:If *expression* is omitted, return as if the action completed normally. –1 The action failed. 0 The target exists but has not been updated. >0 Set the target time to the value specified by *expression*.
rules ["*base-rules*"]	Determines the base rules context.

C Differences Between make and Alcatel-Lucent nmake

The following table provides a brief summary of differences between make and nmake. It is not exhaustive. The table distinguishes the two systems in those areas where you would be likely to assume similarities. The intent of the table is to allow you to avoid common pitfalls.

make		***Alcatel-Lucent nmake***	
make	Interprets the makefile at each execution.	nmake	Compiles the makefile and recompiles only if the original is altered.
	Has no knowledge of any previous run and must infer out-of-date targets by checking time stamps of targets and prerequisites.		Creates a statefile and uses the data in it to decide whether a target is out of date.
	No preprocessor.		Uses a new C preprocessor to facilitate viewpathing.
	Single-character variables can be evaluated without parentheses ($<, $*, $?).		All variables require parentheses ($(<), $(*), $(?)).
make	Metarules defined with prerequisite on left and target on right. Form is: suf_1.suf_2 *rule*	nmake	Defines metarules using the colon assertion operator (:). Form is: pre_1%suf_1: pre_1%suf_1 *rule* (target on left, prerequisite on right)

make		***Alcatel-Lucent nmake***	
@	Prevents printing of command on stdout	@	Not available. Uses silent to suppress command printing in shell actions.
#	Specifies beginning of comment.	#	Remaining content of line interpreted as a comment (when not in first column of line)..
/* */	No significance.	/* */	Marks the start and end of comment.
$(@)	Current target in the case of dependency rules	$(<)	Current target in all cases.
		$(@)	Action for the current target.
$(?)	Current out-of-date prerequisites in the case of dependency rules.	$(>)	Current out-of-date or new prerequisites, or primary metarule prerequisite in metarules.
		$(?)	List of all prerequisites of the current target.
$(*)	Current target for metarules.	$(*)	All current prerequisites.
$(~)	Not available.	$(~)	All current explicit prerequisites.
$(%)	Not available.	$(%)	The file name portion matched by file-generation character (stem) in metarule definition.
$(!)	Not available.	$(!)	All current explicit and implicit prerequisites.
::	Not available.	::	Used in dependency assignments that rely on predefined base rules.
=	Variable assignment by reference.	=	Variable assignment by reference. If the list contains a variable, it is assigned unexpanded. Subsequent changes to that variable affect the expansion of this one.
+=	Not available.	+=	Append to assignment.
==	Not available.	==	Declare variable as candidate implicit state variable.
:=	Not available.	:=	Expand variable in definition only.
&=	Not available	&=	Append auxiliary value to variable.
%	Not available.	%	Metarule file-generation character.

Attribute Differences

In make, .IGNORE is a makefile attribute causing all exit codes to be ignored during make. In nmake, .DONTCARE is an assertion attribute causing related exit codes to be ignored during make, while .IGNORE is an attribute that forces target generation.

Alcatel-Lucent nmake and SCCS

D

SCCS is a maintenance and enhancement tracking tool that runs under the UNIX system. It takes custody of a file and, when changes are made, identifies and stores them in the file with the original source code and/or documentation.

Retrieval of the original or of any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification.

Alcatel-Lucent nmake is a software manufacturing tool that also allows you to generate and control new versions of the software. There is clearly a relationship between SCCS and nmake and some overlap between the two systems. This appendix is intended for knowledgeable users of SCCS who want to obtain the benefits of both nmake and SCCS.

Similarities

SCCS is highly compatible with nmake as long as SCCS g-files are extracted on a top-delta-only basis. Like nmake, SCCS is capable of implementing version control. The extraction of versions can be derived from SCCS's ability to store all states of development of a given software system.

Incompatibilities

Users frequently exploit this capability in the form of extractions done with:

- i-files of applied deltas
- cutoff time/dates for applied deltas

- collection files containing SID release information. In such cases, SCCS-based version control shades into the nmake semantics: rule-based views and viewpathing.

Defining SCCS Metarules

In order to inhibit nmake from scanning s. files for implicit header information, the following metarule can be added to your makefile:

```
.ATTRIBUTE.s.% : .SCAN.IGNORE
```

The following example shows a makefile that defines metarules for SCCS.

Contents of Makefile sccs1.mk

```
%.m : s.%.p
        get -p $(>)  >$(>:/s\.//)
        p2m $(>:/s\.//)  >$(<)

pgm : ymacs.m
```

Run nmake

```
nmake -f sccs1.mk
```

Output

```
+ get -p s.ymacs.p > ymacs.p
1.19
1173 lines
+ p2m ymacs.p > ymacs.m
```

Another example of the use of SCCS is shown below. In this example, the .TERMINAL metarule %.c : s.%.c is defined and is applied to the :: assertion, since .TERMINAL is automatically on for all the prerequisites. Without .TERMINAL on the metarule, this metarule cannot be applied to the :: assertion; thus nmake fails to build pgm because cmd.c does not exist.

Contents of Makefile

```
GET=get
GETFLAGS=
%.c : s.%.c .TERMINAL
        $(GET) $(GETFLAGS) $(>)

pgm :: cmd.c
```

Run nmake

```
$ nmake
```

Output

```
+ get s.cmd.c
1.1
```

```
3 lines
+ cc -O -Qpath /nmake3.0/lib -I-D\
/nmake3.0/lib/probe/C/\
pp/B49DF4E0.bincc -I- -c cmd.c
+ cc -O -o pgm cmd.o
```

Glossary

References to terms that are included in this glossary are *italicized*.

A

Action
See *Update Action Block*

Assertion
An *assertion* consists of a *target* on the left, an assertion operator, and optional *prerequisites* on the right. It is a fundamental *nmake* statement that specifies a *dependency relationship* between different *nmake* entities. Commonly, an assertion specifies that one file depends upon one or more other files. The *action* may be specified in the assertion.

Assertion Operator
The assertion operator introduces optional prerequisites and precedes the *update action block* in an *assertion*.

The : dependency assertion operator is the only built-in assertion operator that is defined in the *nmake* engine. Other assertion operators have been defined in the *default base rules*. You can also define your own assertion operators. The assertion operators defined in the *default base rules* are “macro like” and are expanded to the : dependency assertion operator at *compile* time.

Atom
An atom is any name that appears on either the left-hand side (lhs) or the right-hand side (rhs) of an *assertion operator* in an *assertion*. Specifically, an atom can be in the target position (lhs) or the prerequisite position (rhs). In addition, *implicit* and *explicit prerequisites* are also considered as atoms. The exception to this definition occurs in the *state variable* assignment statement. A state variable declared using the == assignment operator is also considered as an atom by *nmake*.

Attribute
A property of an *atom* that can affect when and how an *assertion* is processed.

Automatic Variable
A *variable* whose value may change circumstantially, but whose value is always known to *nmake* and is accessible by the user. For example, automatic variables can refer to the *target*, the list of *prerequisites*, or even the *nmake* command-line options.

B

Base Rules
nmake supplies a default base rules file. These rules are intended for all *nmake* users and contain rules for a variety of transformations (e.g., compiling, cleaning up, and creating backup files). The compiled base rules file is read as the first set of rules in *nmake* execution. If none of the default base rules are needed, users can define their own base rules files to supersede the default base

rules file that is provided with *nmake*, although complete redefinition of the default base rules is rare. If some rules need to be redefined, they should be redefined in the *global rules* instead of modifying the base rules.

Binding

Atoms are initially names with no relation to objects, such as regular files (not directories or special files). The process of binding relates an *atom* with a specific entity. *Atoms* can be bound to regular files, *state variables*, *virtual atoms* or *labels*. The binding process also encompasses the assignment of *attributes* to *atoms*.

Build

The process of generating a target from a list of *prerequisites*.

Built-in Rules

Often used synonymously for *base rules*. It refers to a set of transformational *rules* that the user can count on as being stable.

C

Canonicalized Path

A path having all extra slashes (/) and dots (.) removed and all double dots (..) moved to the front of the name.

Compile

Convert an ASCII source file into a form that is significantly more compact and can be processed much faster (usually referred to as “machine readable”). *nmake* compiles a user's *makefile* and produces a *make object file*. The process of compilation transforms source files into *object files*.

cpp

The *nmake* C preprocessor is an altered version of the standard C preprocessor that supports the directory search features of *nmake*.

Common Action Target

A *target* whose *actions* are commonly used and predefined in the *base rules*.

Common Actions

An *action* that is specified on the command line. Common actions are defined in the *default base rules*.

Current

When an *atom* has a *time stamp* that is no older than any of its *prerequisites* (also called being up to date).

D

Default Base Rules

The *base rules* provided with *nmake* that define a collection of *options*, *operators*, *common target atoms*, *metarules*, and *variables* suitable for use in the UNIX system programming environment.

Dependency Relationship
A statement that specifies that the value of one or more *target atoms* is a function of one or more other *atoms*, called *prerequisites*. For example, an *object file* has a dependency relationship with its source file and any header files that are included. Dependency relationships in *nmake* are expressed as *assertions*.

Dependency Rule
The collection of all *assertions* for a given target.

E

Edit Operator
An operator used to control the *expansion* of *variables*. In *nmake*, this takes the form of :*operator* following the *variable* name.

Epoch
The time in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970.

Expansion
The process of replacing a *variable* by its value.

Explicit Prerequisites
Prerequisites that are specifically mentioned (either directly or within an *expanded variable*) in an *assertion*.

F

Frozen Variable
A *variable* whose value is fully determined when the *makefile* is *compiled*. If the value of a frozen variable changes, the *makefile* will automatically be recompiled. A *variable* will not be frozen if it is a *state variable*.

G

Global Rules
These rules are similar to *base rules*, but their use is optional. They are intended to be applied on a product-wide basis, rather than an organization-wide basis. If you use a global rules file, you include it in your *makefile* or on the command line, and the file is read immediately after the *base rules* file has been read.

I

Implicit Prerequisites

These prerequisites are determined automatically by *nmake* when processing assertions. For example, *nmake* knows that #include statements name files that are *prerequisites* of the C source files containing the statements.

L

Label

An *atom* of the type label are special objects that do not represent regular files, *state variables*, or *virtual atoms*. If an *assertion* has no *prerequisites* but has an *update action block*, or if an *assertion* has *prerequisites* but no actions and the *target* is not a file, the *target* represents a label in the *makefile*. The *time stamps* of the labels are stored in the *statefiles* but are ignored by *nmake*; therefore, they are made on subsequent invocations of *nmake*.

M

Make

The process of generating a *target*.

Makefile

A text file containing specifications used to produce one or more desired *targets* or *products*, or to perform specific processes using *nmake*. It contains programming constructs, *variable* definitions, *assertions*, and corresponding *actions* that govern the performance of the *nmake* engine.

Makefile Object File

When a user's *makefile* is compiled, the object file is produced, called *makefile_name*.mo.

Metarule

A generic *rule* that specifies the transformation of *prerequisites* that conform to a pattern into *targets* that conform to a different *pattern*. Metarules are formed using the % as a wild card, indicating any number of any characters. The form of a metarule is typically *aaa%bbb* : *ccc%ddd*, followed by an *update action block*.

N

nmake **Engine**

The executable version of *nmake*.

Node

A set of directories arranged in a UNIX system directory structure. The root of the node is the top-most directory in the structure. Any file that can be reached as a descendant from the root is contained in the node. A file in a node is identified by a *relative path name* describing the path from the root of the node to the file.

O

Object File
The result of *compiling* a source file. See *Makefile Object File*.

Operator
A symbol that determines the transformation *rules* to use (if it is an *assertion operator*) or the way the *variable* will be *expanded* (if it is an *edit operator*).

Option
A parameter that can be specified in the *makefile* or on the *nmake* command line. The option specified on the command line overrides any similar options that are specified in the *makefile*. The option manages specific details of the command execution. Options can be used to manage debugging, verbosity, and so on.

P

Parent
A *target* for which the current *target* is a *prerequisite.*

Pattern
Specifies a generic name, where symbols are used in place of one or more characters in the name. In *nmake*, patterns are used in *metarules*, in the shell *scripts* that constitute *update action blocks*, and in the *expansion* of *variables*, when using ed(1) regular expressions.

Prerequisite
A file or value that must be *current* for the *target* to be *current*. Prerequisites appear as *atoms* on the rhs of *assertions*.

Product
Often, the single result of invoking *nmake*; that is, the *root target*. A single invocation of *nmake* can build more than one product.

R

Recursive *nmake* Calls
When a function or program or entity refers to itself, either directly or indirectly (through a chain of other entities), it is *recursive* and creates a condition where *nmake* is initiated from within *makefile* processing using the MAKE engine *variable*.

Relative Path Name
The directory specification leading from the base of the user's *node* to a specified file.

Rule
The collection of all *assertions* for a specific *target*.

Root Target
The *target* that is built by default or specified on the *nmake* command line.

S

Script
A source file or portion of a source file that is interpreted, rather than compiled.

Sibling
If a *prerequisite* of the current *target* is itself a *target*, it is considered a sibling of the current *target*.

Source Dependency Operator (::)
The :: *assertion operator* that allows source files to be *prerequisites*. In addition, use of the source dependency operator eliminates the necessity of explicitly providing the *action* in an *assertion*.

Special Atoms
Special Atoms are used to provide fine control over *nmake* processing. Any *atom* that has been marked with a Special Atom is treated differently during compilation or execution.

Statefile
A file (named *makefile_name*.ms), created and updated each time *nmake* is executed, that contains the *time stamps* of all *prerequisite* files and the values of all *state variables*. The statefile allows *nmake* to determine what *prerequisites* are *current*.

State Variable
A *variable* whose value is stored in the *statefile* and preserved from one invocation of *nmake* to the next. There are three ways of specifying state variables:

- Assign them using the == *operator*.
- Assign them the .STATE *attribute*.
- Enclose them in parentheses when they are given as a *prerequisite*.

T

Target
The object to be produced from a given *rule*.

Time Stamp
The last time the file was referenced. The time stamp is visible in any long directory listing (for example, by using the UNIX system command ls -l *filename*). When you execute the UNIX system command touch *filename*, you change the time stamp to the current time.

U

Update Action Block

The shell *script* defined in the *assertion* that specifies how the *target* is to be made. If the *target* has the .MAKE *Special Atom*, the *script* is to be interpreted by *nmake* rather than the shell.

V

Variable

A symbolic representation of a value. In *nmake*, variables are arbitrary strings (often lists of file names). Variables are defined by name and referenced (that is, *expanded*), similar to shell variables. They are referenced either by $(name) or by $$(name). The *name* may be followed by an *edit operator* that determines how the *expansion* will occur.

Viewpath

An ordered structure of a *product*, seen in multiple views or versions.

Virtual Atom

Declared using the .VIRTUAL *attribute*, this specifies an *atom* that is stored in the *statefile*, but does not *bind* to a file or to a *state variable*.

Index