Alcatel·Lucent

Alcatel-Lucent *nmake* Product Builder
nmake 10 Reference Manual

Issue 1.9
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.

LIMITED USE SOFTWARE LICENSE AGREEMENT

NOTICE:

READ THE TERMS AND CONDITIONS OF THIS AGREEMENT BEFORE YOU DOWNLOAD THE SOFTWARE.

THE TERMS AND CONDITIONS OF THIS AGREEMENT AND ANY APPLICABLE ADDENDUM APPLY TO THE DOWNLOADED SOFTWARE AND DERIVATIVES OBTAINED THEREFROM, INCLUDING ANY COPY, ALL OF WHICH IS REFERRED TO HEREIN AS "SOFTWARE". THE TERM SOFTWARE ALSO INCLUDES PROGRAMS AND RELATED DOCUMENTATION PROVIDED AS PART OF THE DOWNLOAD. BY DOWNLOADING THE SOFTWARE YOU SHOW YOUR ACCEPTANCE OF THE TERMS OF THIS LICENSE AGREEMENT AND SUCH ADDENDUM.

THIS SOFTWARE REQUIRES THE ISSUANCE OF PURCHASED LICENSE KEYS BEFORE THE SOFTWARE MAY BE USED. PLEASE CONTACT THE BELL LABS SOFTWARE LICENSING TEAM AT +1-908-582-5880 OR software@alcatel-lucent.com FOR PRICING INFORMATION. AFTER ALCATEL-LUCENT HAS RECEIVED A PURCHASE ORDER OR AUTHORIZATION WHICH SPECIFIES THE PROPER FEES AND OTHER INFORMATION (INCLUDING, FOR EXAMPLE, PLATFORM AND HOST IDENTIFICATION) NECESSARY TO ISSUE THE LICENSE KEY(S), ALCATEL-LUCENT WILL ISSUE TO YOU THE LICENSE KEY(S) GOOD FOR THE PERIOD OF TIME SPECIFIED IN SUCH ORDER OR AUTHORIZATION, SUBJECT TO THE PROMPT RECEIPT OF PAYMENT BY ALCATEL-LUCENT. ALL RIGHTS GRANTED TO YOU UNDERTHIS AGREEMENT AND ANY APPLICABLE ADDENDUM ARE IN EFFECT UNTIL THE END OF THE TIME PERIOD SPECIFIED IN YOUR ORDER, OR SOONER IF YOU NOTIFY ALCATEL-LUCENT THAT YOU HAVE DESTROYED THE SOFTWARE AND ANY AND ALL ASSOCIATED LICENSE KEYS.

1. **TITLE AND LICENSE GRANT**

   The software is copyrighted and/or contains proprietary information protected by law. All SOFTWARE, and all copies thereof, are and will remain the sole property of ALCATEL-LUCENT or its suppliers. Subject to the payment of fees specified in an accepted purchase order or authorization for one or more license keys, ALCATEL-LUCENT. hereby grants to you, for the period of time specified in such order or authorization, a limited, personal, nontransferable and non-exclusive right to use the SOFTWARE, in accordance with all applicable United States laws and regulations, on only that computer, server or authorized clients and only for up to the maximum number of authorized users specified in such order or authorization. Any other use of this SOFTWARE inconsistent with the terms and conditions of this Agreement, including without limitation, transfer of the SOFTWARE to another computer or to another party, shall automatically terminate this license.

   You agree to use your best efforts to see that any use of the SOFTWARE licensed hereunder complies with the terms and conditions of this License Agreement as well as all laws and regulations of the United States and any foreign country in which the SOFTWARE is used. You further agree to refrain from taking any steps, such as reverse engineering, reverse assembly or reverse compilation, to derive a source code equivalent of the SOFTWARE.

2. **SOFTWARE USE**

   A. You are permitted to make a single archive copy of the SOFTWARE, provided the SOFTWARE shall not be otherwise reproduced except as is necessarily incident to the execution of the SOFTWARE in a computer or server. Any such copy shall contain the same copyright notice and proprietary marking appearing on the original SOFTWARE. The SOFTWARE shall not be disclosed to others in whole or in part.

   B. The SOFTWARE and any and all associated license keys,

      1. together with any archive copy thereof, shall be either returned to ALCATEL-LUCENT or certified as having been destroyed when the SOFTWARE is no longer used in accordance with this License Agreement, or when the right to use the software is terminated; and

      2. shall not be removed from a country in which use is licensed or from any computer or server on which it is licensed without ALCATEL-LUCENT's express written permission.

   C. You acknowledge and agree that the SOFTWARE subject to this agreement is subject to the export control laws and regulations of the United States, including but not limited to the Export Administration Regulations (EAR), and sanction regimes of the U.S. Department of Treasury, Office of Foreign Assets Controls. You agree to comply with these laws and regulations. You shall not, without prior U.S. Government authorization, export, reexport, or transfer any SOFTWARE subject to this agreement, either directly or indirectly, to any country subject to a U.S. trade embargo or sanction (Cuba, N. Korea, Iraq, Iran, Syria, Libya, Sudan) or to any resident or national of said countries, or to any person, organization, or entity on any of the restricted parties lists maintained 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.

5. **SUPPORT, UPDATES, AND UPGRADES**

If you have obtained a license to use SOFTWARE during a defined annual period, you will be entitled to obtain support, upgrades and updates that Alcatel-Lucent makes available during that annual period, without additional charge. If you have obtained a perpetual license to use SOFTWARE (with optional maintenance), you will only be entitled to support, updates and upgrades if you have also obtained and paid for a valid maintenance agreement. All support will be provided in accordance with Alcatel-Lucent's support policy, which may be viewed at the following website:

http://www.bell-labs.com/projects/sablime/TechSupportPolicy.htm

Notwithstanding any other provision in this License Agreement, while Alcatel-Lucent will use reasonable efforts to provide support, updates or upgrades, Alcatel-Lucent shall not be responsible for (i) any loss or damage to your property (software or hardware) that occurs in connection with the provision of support or while providing other services under this agreement, (ii) any interruption in the use of the SOFTWARE or any other third-party software or hardware operating in conjunction with the SOFTWARE, (iii) issues that arise as a result of third party software or hardware used in conjunction with the SOFTWARE, (iv) any damage to such third party software or hardware that occurs during the provision of support.

6. **COMPLETE AGREEMENT**

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT AND UNDERSTAND IT, AND THAT BY DOWNLOADING THE SOFTWARE YOU AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT THIS AGREEMENT, TOGETHER WITH ANY APPLICABLE ADDENDUM APPLICABLE TO THE SOFTWARE, IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE RIGHTS AND LIABILITIES OF THE PARTIES. IN THE EVENT OF CONFLICT, THE PROVISIONS IN AN APPLICABLE ADDENDUM SHALL TAKE PRECEDENCE. THIS AGREEMENT AND ANY APPLICABLE ADDENDUM SUPERSEDE ALL PRIOR ORAL AGREEMENTS, PROPOSALS OR UNDERSTANDINGS, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

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

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Commands

1

This section contains manual pages for the Alcatel-Lucent nmake Product Builder commands.

coshell Command

coshell - network shell coprocess server

Synopsis

coshell + [*info*... **]**
coshell –
coshell –*op* **[** *arg*... **]**

Description

coshell is a local network shell coprocess server for programs using *coshell*(3). 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), 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.

Job requests are accepted from user processes on the local host and are executed on the connected hosts. stdout, stderr, FPATH, NPROC (see Environment), PWD, PATH, VPATH, vpath, umask and the environment variables listed in COEXPORT (see the *Environment* section) 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 system status daemon. This information is updated every 60 seconds on average.

The server is started by running coshell +. The command exits after a child server process is forked in the background. The optional info arguments name files containing local network host information which may be generated from two shell scripts genlocal and genshare under the subdirectory bin of the installation root directory. If no files are specified then the default local is used. The local network is comprised of hosts sharing the same file name space.

Attributes used by coshell can be categorized into two types, global and host-specific. The global attributes control coshell and are not associated with any particular host. Attribute value pairs, not including readonly ones, may be specified in the local network host information files, in COATTRIBUTES (see Environment) or may be set/added using coshell -a, and may be referenced in an expression in COATTRIBUTES. Attribute names must match [a-zA-Z_][a-zA-Z_0-9]*. In the following description on these attributes, host may be an actual host name or a comma-separated list of attribute-value pairs specified in COATTRIBUTES.

The attributes used by coshell are:

auto=*n*	auto=1 adds the host to the automatic selection pool. Hosts named in the file global-attribute have **auto=1** by default.
bias=*n.m*	The host scheduling bias. Hosts with a high bias are (linearly) least likely to be scheduled for job execution. The default bias is **1.00**.
busy=*time*	(global attribute) The grace period for jobs running on busy hosts. A host is busy when its idle attribute is non-zero 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 may 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* may be m(inute) and h(our). The default is **busy=2m**.
cpu=*n*	The number of CPUs on the host. The default is **1**.
file=*path*	(global attribute) Names a file containing default attributes for machines on the local network. If no directory components are specified then the subdirectory share/lib/cs of the installation root directory is searched. The default attribute file for the local network is **share/lib/cs/local**.
idle=*time*	The minimum logged in user idle time before jobs will be scheduled on the host. The meaningful unit of *time* may be m(inute) and h(our). The default is **0**, meaning no idle time restrictions. idle is usually **15m** for workstations and is not specified (i.e., always available) for compute servers.
label=*string*	*string* labels either the current coshell connection (via coopen in coshell(3)) or the current job (via coexec in coshell(3)). Labels are displayed but are otherwise ignored.
load=*n.m*	A readonly attribute that evaluates to the host load average.
maxload=*n.m*	(global attribute) The maximum host load average. No job will be scheduled on a host with load average >= maxload**.** The default maxload=0 means no load average limit.
name=*host*	The host name in the local domain (i.e., no .'s in the name). The name= may be omitted. In a host selection context: *host* may be a sh(1) file match pattern; **–** matches any host; local matches the local host.
open=*fd*	A readonly attribute that evaluates to **0** if the host shell is closed, **<0** if the host shell is being opened, and **>0** if the host shell is open.

percpu=*n*	(global attribute) The maximum number of concurrent jobs on each CPU. The default is **3**.
perserver=*n*	(global attribute) The maximum number of concurrent jobs run by *coshell.* perserver has an upper limit that is silently enforced. The limit is the half of the number of file descriptors allowed. perserver=0 queues jobs until perserver>0**.**
peruser=*n*	(global attribute) The maximum number of concurrent jobs per user connection to *coshell.* The default is **12**.
pool=*n*	(global attribute) The number of CPUs in the processor pool.
profile=*string*	(global attribute) Command sequence used to initialize remote shells. Default is **{ eval test -f ./.profile \&\& . ./.profile; eval test -f \$ENV \&\& . \$ENV; } >/dev/null 2>&1 </dev/null**
rating=*n.m*	The host CPU rating, usually in mips relative to the other hosts on the local network. This is usually the observed rating rather than the one in the vendor's advertisements.
type=*string*	The host type that differentiates different processor types, usually related to the object and executable attributes. The default type is *.
up=*n*	A readonly attribute that evaluates to the number of seconds the host has been up. If *n* is less than 0 then it is the number of seconds the host has been down.

Other user-defined attributes may be specified. They may be referenced in COATTRIBUTES expressions, but are otherwise ignored by coshell.

The coshell – form of the synopsis opens an interactive connection to the running server. The commands are:

a *host[,attributes...]*	Set or add attributes for the named hosts.
c *host...*	Close the shell connections to the named hosts. The hosts are also removed from the automatic selection pool.
d [*level*]	Set the server stderr to the stderr of the calling process. If *level* is specified then the server debug trace level is set to –*level*. Higher debug levels produce more output on stderr.
f [*fd*]	This is a debugging option and may not be present in all implementations. If *fd* is specified then close the internal server file descriptor *fd*, otherwise list the status of all open file descriptors in the server.
g	List global state.

h	List command help.	
j	List the status of all jobs. The status fields are:	
	JOB	The id assigned to the job by the server. This number may be used as an argument to the **k** command.
	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 shell 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.
k [c | k | s | t] *job*	Kill the job with the server JOB id *job*. If no argument is specified then the SIGTERM signal is sent to the job. **c** sends SIGCONT, **k** sends SIGKILL, **s** sends SIGSTOP, and **t** sends SIGSTERM.	
l *expr*	List all host names matching the attribute expression *expr*. The names are sorted in scheduling rank order from best to worst. If pool=*n* is specified in *expr* then only the first *n* names (after sorting) are listed.	
o *host...*	Open a shell connection to the named hosts.	
q	Quit the interactive connection.	
Q	Kill the server and quit the interactive connection.	
r *host* [*command*]	Run *command* on *host*. *host* may be an attribute expression. If *command* is omitted, then hostname(1) is used.	

s [a | e | l | o | p | s | t]	List the shell connection status. There is at most one shell connection per host. If no argument is specified then only open connections are listed. a lists the attributes for all shells, e lists all shells, l lists all shells in the processor pool, o lists all open shells, p lists the process id of all open shells, s lists the shell scheduling status (primarily for debugging), and t lists all open shells sorted by the recent job activities running on each host. The status fields for se and sl are:	
	CON	The id assigned to the open shell by the server, @ if the shell is not open and is not in the processor pool, **–** if the shell is not open, and **+** if an open is in progress.
	JOBS	The number of jobs currently running on the host. ***** follows the number if any of the jobs are queued pending the completion of an open in progress.
	TOTAL	The total number of jobs run on the host.
	USER	The accumulated user time (times(2)) of all jobs on the host.
	SYS	The accumulated sys time (times(2)) of all jobs on the host.
	IDLE	The elapsed time since the most recent logged in user activity. ***** follows the time if the host does not meet the processor pool idle time requirements.
	CPU	The number of CPUs on the host.
	LOAD	The host load average.
	RATING	The host rating, usually in network relative mips.
	BIAS	The scheduling bias. Hosts with lower bias are more likely to be scheduled.
	TYPE	The host type, usually related to object and executable attributes.
	HOST	The host name.

s [a | e | l | o | p | s | t] (cont.)	The status fields for so,ss, and st are:	
	CON	The id assigned to the open shell by the server, @ if the shell is not open and is not in the processor pool, **–** if the shell is not open, and **+** if an open is in progress.
	OPEN	The accumulated number of times the server has connected to the host.
	USERS	The current number of active users.
	UP	The amount of time the host has been up.
	CONNECT	The amount of time the server has connected to the host.
	UPDATE	The amount of time before the host status information is out-of-date.
	OVERRIDE	The amount of time of keeping the host connection followed by the host identification code, 1 for the local host, 0 for other hosts in the network.
	IDLE	The specified idle time.
	TEMP	A measure of the recent job activities running on the host.
	RANK	A measure of the desirability of the host. This takes idle time restriction, load average, and the number of CPU into account. Two digits after the decimal point are random numbers which are used to break ties between different coshell servers. Hosts with lower RANK are more likely to be scheduled.
	HOST	The host name.

t	List the accumulated totals. The fields are:	
	SHELLS	The number of active shell connections followed by the total number of successful shell connections.
	USERS	The number of active user connections followed by the total number of successful user connections.
	JOBS	The number of active jobs followed by the total number of jobs run.
	CMDS	The number of server-user transactions.
	UP	The elapsed time since the server started.
	REAL	The elapsed time during which the USER and SYS times were accumulated.
	USER	The accumulated user time for all jobs on all hosts.
	SYS	The accumulated sys time for all jobs on all hosts.
	CPU	The number of CPUs available on all connected hosts followed by the processor pool CPU limit plus the explicit host override count. An *override* host is a connected host that does not meet the processor pool idle time requirements.
	LOAD	The load average, averaged over all connected hosts.
	RATING	The host rating, averaged over all connected hosts.

u	List connected user status. The status fields are:	
	CON	The id assigned to the user connection by the server.
	PID	The user process id.
	JOBS	The number of jobs currently running on behalf of the user.
	TOTAL	The total number of jobs requested by the user.
	TTY	The user process stderr file name.
	label	The label assigned to the connection by the requesting user.
v	List the server version stamp.	

The interactive commands are useful in terms of tuning some global variable values. For example, one could set peruser to 10 and perserver to 40 using the coshell interactive command:

```
coshell> a local,peruser=10,perserver=40
```

The interactive commands may be used as options for non-interactive coshell queries. For example, coshell -sl produces a long shell status listing and coshell -c dodo closes the shell connection to the host dodo.

Examples

The following environment variables must be set if coshell is installed in a non-standard directory (not /bin, /usr/bin, or /usr/local/bin):

```
root=<coshell-installation-root-directory>
export PATH=$root/bin:$PATH
```

If coshell is dynamically linked, the LD_LIBRARY_PATH environment variable needs to be set.

```
export LD_LIBRARY_PATH=$root/lib:$LD_LIBRARY_PATH
```

The following two commands are used to generate the local network host information which is shared among all the coshell users and only needs to be generated once unless this information needs to be updated. If permission problems are encountered, contact the system administrator.

```
genshare > $root/lib/cs/share
genlocal > $root/share/lib/cs/local
```

The genshare command is run first to generate information on servers for the network. By default, this information is stored in $root/lib/cs/share. Based on this information, genlocal is run to generate the local host attribute file. By default, this information is stored in $root/share/lib/cs/local. If the share file generated by the genshare command is not stored in the default path, its path must be passed to the genlocal command using the -f option.

The generated files may also be modified to meet user needs.

A sample local host attribute file follows:

```
#
# local host attributes
#
local       pool=8      bias=4      busy=1m
server      type=sun4   rating=20
cruncher    type=mips   rating=30   cpu=20
station     type=sun3   rating=6    idle=15m
token       type=3b     rating=0.1  idle=15m
```

The "local" entry sets the processor pool size to 8, the local host bias to 4, and the busy host grace period to 1 minute. 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.

The following starts the coshell server. The processor pool size is taken from the local host attribute file.

```
coshell +
```

The following instructs programs using coshell(3) to use coshell rather than ksh or sh for command execution and sets the command execution concurrency level to 8.

```
export COSHELL=coshell
export NPROC=8
```

The shell function cosh provides a convenient interface for common coshell actions:

```
export FPATH=$root/fun:$FPATH
# start coshell, export COSHELL,NPROC, and set window title
cosh
coshell 9.1.2.1 (Alcatel-Lucent Bell Laboratories) 06/30/06 [first time only]
# run hostname on best host
on - hostname
dodo
```

```
# interact with server ...
cosh -
coshell>
```

Caveats

A coshell connect stream file is created in the /tmp/cs directory. Some systems:

1. do not update the times on the connect stream file when it is accessed
2. automatically remove stale files from /tmp
3. fail to generate a poll(2) or select(2) event when the connect stream file is removed
4. do not handle mounted streams or sockets.

In any of these cases, the environment variable CS_MOUNT_LOCAL needs to be set to another file system where all the users have read and write permissions. For example:

```
export CS_MOUNT_LOCAL=<coshell-installation-root-directory>/tmp
```

On some systems the server may not detect that its connect stream file has been unlinked, resulting in erroneous ‘server not running’ errors. To handle this situation the server checks and recreates the connect stream file on receipt of a SIGINT signal.

NFS cache inconsistencies may arise for files generated via NFS on remote hosts but serviced via the native file system on the local host. Running coshell from a diskless host avoids the problem.

Host load average and logged in user idle times are used to schedule hosts and jobs. Some terminal lock programs, e.g., xlock(1), inflate the load average, usually doing complex graphics operations on displays that have long since been blanked out by an independent screen saver. A simple lock program that blocks on a read request may open up idle cycles for better use.

Environment

COATTRIBUTES	Host attribute expression, **(**type@local**)** by default. Non-numeric valued attributes may 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(1) 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, and (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 is to support 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 in the nmake manual page) This variable may be referenced in .profile.
HOMEHOST	Set within each action to the name of the host executing coshell.
HOSTNAME	Set within each action to the name of the host executing the action. This variable may be referenced in .profile**.**
HOSTTYPE	Set within each action to the type (from the local coshell host attribute file) of the host executing the action. This variable may be referenced in .profile**.**
NPROC	Default command concurrency level

Files

share/lib/cs/local local network host attributes

Related Commands

3d(1), ksh(1), nmake, rsh(1), ss, coshell(3), cs(3)

cpp Command

cpp - stand-alone C preprocessor

Synopsis

/lib/cpp [*option* ... **] [** *ifile* **[** *ofile* **]]**

Description

cpp is the stand-alone preprocessor that is automatically invoked by the cc(1) command. Although primarily intended for the C language, there are several options that support preprocessing of other languages.

cpp accepts two optional file name arguments. *ifile* and *ofile* are respectively the input and output for the preprocessor, defaulting to standard input and standard output if not specified.

CAUTION:
This is a non-standard file argument syntax that is required by some C compiler implementations.

Dialects

cpp preprocesses one of two dialects:

ansi	The default dialect **ansi** provides a conforming implementation, with extensions, of the preprocessing translation stages defined in the ANSI C standard. Refer to this document for detailed descriptions. The macro __**STDC**__ is predefined in this dialect.
compatibility	This dialect provides almost complete compatibility with the old (Reiser) *cpp*. Non-supported features of old *cpp* appear in the diagnostics when possible. The macro __**STDC**__ is not predefined in this dialect. Refer to the *Compatibility* section below for a description of differences between this and the **ansi** dialect.

Each of these dialects has three modes - **strict**, **transition**, and **C++.** The dialects may have a strict interpretation, controlled by the **–D–S** option. When the **strict** mode is in effect, warning diagnostics are issued when dialect extensions are used in **non-hosted** files (see the **–I–H** option below).

The macro __**STDPP**__ is predefined to be 1 in all dialects.

Options

The options are processed in order from left to right. Options issuing directives are executed after pp_default.h is read. The options are:

–A*arg*	SVR4 assertion. *arg* is the name of assertion. This option is equivalent to the directive **#assert** *arg*.
–C	Do not strip comments from the input files.

–D*arg*	Additional cpp options accepted by most versions of cc(1). *arg* may be:	
	–C	Preprocess the **compatibility** dialect.
	–D*level*	Set the debug trace level to *level*. See the **pp:debug** pragma below.
	–F*filename*	Change the name of the input file reported in the line control information to *filename*.
	–H	Mark all files **hosted**. See the **–I–H** option below.
	–L[*line-directive*]	Change the directive name for line number control output directives to *line-directive*. See the **pp:lineid** pragma below.
	–M	Mark all files to be included only once. See the **allmultiple** and **multiple** pragmas below.
	–P	Enable the **passthrough** mode. Specifies the passthrough mode, especially useful for preprocessing other languages (e.g., nmake). This option is not suitable for C language processing. In this mode: • comments are removed • space characters are preserved on output • **#(...)** results are not enclosed in "" • \ escapes the special meaning of " and ’ • **\newline** is removed only in directives • " and ’ constants may contain **newline**s
	–Q	Produce a checkpoint dump of *ifile* in *ofile*. *ifile* will be searched for in all **-I** directories. A checkpoint dump file requires no preprocessing; it contains a fully preprocessed text section followed by a dump of the macro definitions. Checkpoint dump files may be subject to **#include** directives just as normal text files. Performance will most likely be improved for source files that reference large collections of common, stable header files.

–D*arg* (cont.)	**–R**	Preprocess for **transition** mode.
	–S	Provide a strict dialect interpretation. See the **pp:strict** pragma below.
	–T*test*	Enable the internal debugging test *test*. *test* may be **1**, **2** or **3** (both 1 and 2). A diagnostic indicating the behavior of selected tests is issued.
	–W	Produce additional compatibility diagnostics relating to the current dialect.
	–X*probe*	Set the default definition probe key to *probe*. In the absence of pp_default.h *probe* is used to generate the information using probe.
	–+	Preprocess for the C++ language. See the **pp:plusplus** pragma below.
	+*option*	Invert the sense of the corresponding **–***option* from above.
	:*option*[=*value*]	Equivalent to the directive **#pragma pp:***option value*.
	%*directive*	Enters the preprocessing directive *directive* from the command line.
	#*predicate*([*args*])	Equivalent to the directive **#define** #*predicate*([*args*]).
	name[=*value*]	Equivalent to the directive **#define** *name value*. *value* defaults to **1** if omitted.
–E	No effect. Provides compatibility with some versions of cc(1).	
–H	Print the included reference files in stderr. This is for System V compatibility.	

–I*arg*	Include file related options. *arg* may be:	
	–	Cause "" include files to be searched for in all **–I** directories and <> include files to be searched for only in the **–I** directories listed after **–I–**. The standard include directory is always searched last for both include file forms. This allows the *directory of the including file* (see the **–I***directory* option below) to be replaced by a list of directories for "" include files. To first search for "" include files in the directory of *ifile* specify **–I***directory(ifile)* **–I–***...*, where *directory(ifile)* is the directory portion of the *ifile* file name.
	–C	This is for C++. Any **-I***dir* after **-I-C** will be marked as C source. This is equivalent to **pp:cdir** pragma below.
	–D[*file*]	Use the default definitions from *file* rather than **pp_default.h**. If *file* is omitted then no default definitions file is read. *file* is read using the **–I–R** mechanism described below.
	–H[*directory*]	Include files found in **–I** directories listed after **–I–H** are marked **hosted**. If *directory* is speci-fied then all **–I** directories, including *directory* itself, listed after **–I***directory* are marked **hosted**. Most warning diagnostics are relaxed for **hosted** files. Files found in the standard include file directory (**–I–S** below) are marked **hosted** by default. If **–D–H** (**–D+H**) is specified then all files are marked **hosted** (**non-hosted**) regardless of the **–I–H** option.
	–I*directory*	Initialization files, (e.g., **pp_default.h,–I–R),** are searched for in *directory* before the default standard include directory.

–I*arg* (cont.)	**–I**[*file*|–*char suffix*]	Ignore **"..."** quoted **#include** file names listed in file *file*, or, if **–***char suffix* is specified, the file named by either changing characters following *char* in the input file base name to *suffix* or appending *char suffix* to the input file name if it does not contain *char*. **#include** on any of the listed files will be ignored. Only the last **–I–I** option takes effect. If *file* is omitted then the option is ignored.
	–R*file*	Read the contents of *file* using **#include** "*file*". *file* and all files included by *file* are marked **hosted** even if **–D+H** is specified. Line sync output is also disabled for these files.
	–S*dir*	Set the standard include file directory to *dir*. This directory is the last place searched for all include file forms. The default standard include file directory is /usr/include. Only one standard include directory may be specified.
	–T*file*	Read the contents of *file* before the input file. The output of *file* from *cpp* is used in the input file.
	+*option*	Invert the sense of the corresponding **–***option* from above
	dir	"" include files not beginning with **/** are first searched for in the directory of the including file, then in the directories named in **–I** options and finally the standard directory. <> include files use the same search order as "" files except that <> files are not searched for in the directory of the including file. The directory of the including file can be replaced by a possibly empty list of directories using the **–I–** option described above. **NOTE:** *dir* cannot start with the **–** character.
–M	Output the file dependencies in makefile assertion format. This is for BSD compatibility.	
–P	Preprocess without producing line control information.	

–T	Truncate macro names for compatibility with non-flexname (8-character) compilation systems. This is equivalent to **pp:truncate** pragma below.	
–U*name*	Equivalent to the directive **#undef** *name*.	
–V	Prints the cpp version in stderr.	
–X*dialect*	Set the System V standard preprocessing dialect to *dialect*. *dialect* may be one of:	
	a	The default **ansi** dialect.
	[Ac]	The **ansi** strict mode (ANSI conforming). **–D–S** is preferred.
	F	The C++ mode.
	f	The **compatibility** transition and C++ mode.
	[ks]	The **compatibility** strict mode.
	o	The old cpp.
	t	The **compatibility** transition mode. **–D–C** is preferred.
	The table below summarizes the mode(s) that each *dialect* represents.	
–Y*dir*	Set the standard include file directory to *dir*. **–I–S***dir* is preferred.	

The following table summarizes the mode(s) that each *dialect* represents:

dialect	**compatibility**	**transition**	**strict**	**C++**
a	0	1	0	0
A|c	0	0	1	0
F	0	0	0	1
f	1	1	0	1
k|s	1	0	1	0
o	1	0	0	0
t	1	1	0	0

Directives

cpp directives are a single line (after all **\newline** sequences have been removed) starting with # as the first non-space character on the line. A space character is any one of **space**, **tab**, **vertical-tab** or **formfeed**. **vertical-tab** and **formfeed** are not valid between the initial # and the terminating **newline**. Any number of **space** and **tab** characters may appear between the initial # and the directive name. All tokens in directives are significant; trailing tokens, sometimes used as commentary in other implementations, must be enclosed in **/*...*/**. The **#include**, **#if**, **#ifdef**, **#ifndef** and **#macdef** directives can be nested, although the nesting levels must balance within files. In the following an *identifier* matches the regular expression **[a–zA–Z_][a–zA–Z_0–9]*** and must not be immediately preceded by **[0–9]**.

The directives are:

#define *name token-string*	Replace subsequent instances of *name* with *token-string*.
#define *name(arg,...,arg) token-string*	Replace subsequent instances of *name* followed by a **(**, a list of comma-separated set of tokens, and a **)** by *token-string*, where each occurrence of an *arg* in the *token-string* is replaced by the expanded value of the corresponding set of tokens in the comma-separated list. The argument replaced *token-string* is then re-scanned for further macro replacement. Notice that there can be no space between *name* and **(** in the definition. Formal arguments appearing in single or double quoted strings are replaced by the corresponding unexpanded actual argument text only in the **compatibility** dialect. Macro recursion is inhibited by not expanding a macro name appearing in its own definition. If the last formal argument is followed by the **...** token then it is replaced by the expanded value of all remaining actual arguments and intervening **,** tokens from the macro call. If there is only one formal argument then the macro may be called with no actual arguments, otherwise there must be at least one actual argument for the last formal argument. The token # in *token-string* causes the immediately following formal argument to be replaced by the unexpanded value of the corresponding actual argument enclosed in double quotes. The token ## in *token-string* concatenates the space separated tokens immediately preceding and following the ## token. The resulting token is not checked for further macro expansions. Formal arguments preceding ## or following ## and # are replaced by the unexpanded value of the corresponding actual argument.

#define *#predicate***(***arg***)**	Makes the assertion *#predicate*(*arg*) that may be tested only in **#if** or **#elif** directive expressions. *predicate* must be an identifier and *arg* may be any balanced parenthesis sequence of tokens not containing **newline**. The assertion in no way conflicts with the **#define** macro name space. Space character sequences in *arg* are canonicalized into single **space** characters and macro expansion is inhibited on both *predicate* and *arg* during predicate assertion and evaluation. Within **#if** expressions, *#predicate(arg)* evaluates to 1 if **#define** *#predicate***(***arg***)** has been specified; otherwise it evaluates to 0. Likewise, *#predicate()* evaluates to 1 if any assertion has been made on *predicate*. Multiple assertions on *predicate* are allowed, with all such assertions evaluating to 1 in **#if** expressions.
#macdef *name...*	Defines the multi-line macro *name*. A matching **#endmac** ends the definition. Nesting is allowed. As with **#define**, *name* may have arguments. The definition body may contain directives; these directives are not executed until the macro is expanded.
#elif *constant-expression*	Allows multiple alternate branches for the **#if** directive. *constant-expression* evaluation is the same as for **#if**.
#else	Reverses the sense of the test directive matching this directive. If lines previous to this directive are ignored then the following lines will appear in the output.
#endif	Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**.
#error [*message*]	Emits *message* as a *cpp* error diagnostic and terminates preprocessing immediately. The default *message* is **user-error**.

#if *constant-expression*	Subsequent lines will appear in the output if and only if *constant-expression* evaluates to non-zero. All non-assignment C operators are valid in *constant-expression*. Operator precedence is the same as in the C language. Computations are done using **long int** and **unsigned long int** arithmetic on the host machine; floating point computations are not supported. Any macros in *constant-expression* are expanded before the expression is evaluated. The builtin predicate **defined(***name***)** tests if the macro name *name* has been defined. The builtin predicate **exists(***file***)** tests if *file* can be found using **#include** search rules. *file* must be enclosed in "" or <>. **exists** accepts additional optional quoted string arguments that are **:** separated lists of directories to search for the existence of *file*. The normal **#include** search rules are overridden in this case. The builtin predicate **strcmp(***token1,token2***)** compares the macro expanded string values of *token1* and *token2* using the *strcmp*(3) library function. Only these operators, functions, predicates, integer constants and names may be used in *constant-expression*. In particular, the **sizeof** and **,** (comma) operators are not valid in this context.
#ifdef *name*	The lines following will appear in the output if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.
#ifndef *name*	The lines following will not appear in the output if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.
#include <*filename*>	The standard include directories (see the **–I** option above) are searched for a header identified uniquely by *filename*, and the directive is replaced by the entire contents of the header. If the **allmultiple** pragma is off 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. Any macros on the directive line are first expanded before the directive is processed.

#include *"filename"*	The directory of the including file (see the **–I** option above) is searched for a source file identified uniquely by *filename*, and the directive is replaced by the entire contents of the source file. If the **allmultiple** pragma is off then subsequent **#include** references to the source file named by *filename* are ignored unless the source file contains a **#pragma multiple** directive that was processed during the first inclusion of the source file. Any macros on the directive line are first expanded before the directive is processed. If the source file search fails then the directive is reprocessed as if it were **#include**<*filename*>.
#let *identifier = constant-expression*	Defines the macro *identifier* to be the value of the evaluated *constant-expression*, where *constant-expression* is the same as for the **#if** directive. Macro redefinition diagnostics are suppressed for macros defined by **#let**.
#line *integer-constant* [*"filename"*]	Outputs line control information for the next pass. *integer-constant* is the line number of the next line and *filename* is the originating file. The current file name is set to *filename* if specified. Any macros on the directive line are first expanded before the directive is processed.

#pragma [*pass*:][**no**]*option* [*args* ...]	Sets preprocessor and compiler control options. Use of **#pragma** should be limited to **hosted** files as the interpretation varies between compiler implementations. A warning diagnostic is issued when **#pragma** directive is encountered in a **non-hosted** file. This directive is completely ignored for **non-hosted** files in the **strict ansi** dialect. If *pass* is **pp** then the option is used and verified and is not passed on, else if *pass* is omitted then the option is used and passed on, otherwise the option is passed on and not used. **#pragma** arguments are not checked for macro expansions. If **no** is present then *option* is turned off. Pass specific pragmas should not omit *pass:*. Options specified on the command line override options in the default include file. The *cpp* specific options are:	
	allmultiple	Marks all include files **multiple**. This is the default.
	builtin	Sets a mode that marks all macros defined by **#define** *builtin*. *builtin* macro definitions are not dumped by the **–D–Q** option.
	cdir	Marks include files after this pragma as C source. This is for C++.
	compatibility	Sets the **compatibility** dialect.
	debug *number*	Sets the debug trace level to *number*. Higher levels produce more output. Debug tracing is enabled only in debugging versions of the preprocessor.
	elseif	Allows **#else if**, **#else ifdef**, and **#else ifndef** directives.
	hostdir "*dir*"	Include files found in *dir* or after *dir* in the **–I** directory list are marked hosted.
	id "*string*"	Adds the characters in *string* to the set of characters that may appear within an identifier name. For example, **#pragma pp:id "$"** causes **sys$call** to be tokenized as a single identifier. *string* is currently limited to "**$**". Once added a character cannot be deleted from the identifier set.
	include "*dir*"	Equivalent to the **–I***dir* command line option.

#pragma [*pass*:][**no**]*option* [*args* ...] (cont.)	**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.
	linetype	Specifies that line number control output directives are to contain an additional include file type argument. The line number control output directives are of the form *#line-directive line "file" type* where *line-directive* is set by the **pp:lineid** pragma or **–D–L** option (null by default), and *type* is **1** for include file push, **2** for include file pop and null otherwise.
	load	Specifies that lines following this directive were produced by a **–D–Q** checkpoint dump. This option should not be used explicitly.
	macref *name type*	Specifies that macro reference pragmas are to be emitted. *name* is the macro name and *type* is **–2** for **undef**, **–1** for a reference in **#if**, **#ifdef** or **#ifndef**, **0** for macro expansion and **>0** (number of lines in the definition) for macro definition.

#pragma [*pass*:][**no**]*option* [*args* ...] (cont.)	**map** [*id* ...] "/*from*/[,/*to*/]" ["/*old*/*new*/ [*glnu*]" ...]	**map** allows unknown directives and **pragma** options to be mapped to other directives and rescanned. The optional *id*'s support the mapping of standard directives and options as well. In this case the standard directives and options may be accessed by __*id*__. Each unknown directive line is space canonicalized and placed in a buffer that is subject to **pp:map** editing. This buffer contains the initial **#** and omits the trailing **newline**. *from* is an *egrep*(1) style regular expression, with the addition of the identifier delimiter operators **<** and **>**, and the proviso that **^** and **$** match the beginning and end of string (rather than line). The expressions are delimited by **/** in the example, but any character may be used, as long as it is escaped within the expressions. The maps are searched, last in first out, for the longest *from* pattern that matches the unknown directive buffer. The ed(1) style substitute expressions for the longest *from* match are then applied left to right. The optional **g** substitutes all occurrences of *old* to *new*. **l** (**u**) converts *new* to lower (upper) case. **n** specifies that the substitute expression is to be applied only if all previous substitute expressions failed. The standard C escape sequences are recognized in all map patterns. In particular, **\n** translates to **newline**, allowing a single directive line to be mapped into many lines. After all substitutions have been applied the resulting buffer is pushed back onto the input token stream and rescanned. The original directive line number is preserved during the rescan. If any of the mapped lines start with ## then the text between ## and the next **newline** is copied verbatim to the output. If the resulting buffer is empty then the input directive is ignored.

#pragma [*pass*:][**no**]*option* [*args* ...] (cont.)	**map** [*id* ...] "/*from*/[,/*to*/]" ["/*old*/*new*/ [*glnu*]" ...] (cont.)	If *to* is also specified then the nested construct *from* to *to* is matched. For example, #pragma pp:map "/#(pragma)?ident>/" causes **#pragma ident ...** and **#ident ...** directives to be silently ignored and, #pragma pp:map "/#pragma lint:/" ",#pragma lint:(.*),##/*\1*/,u" maps **#pragma lint:argsused** to **/*ARGSUSED*/** on output.
	multiple	If the **allmultiple** option is on then the **#include** directive ensures that each header and source file is included at most one time. A **#pragma multiple** directive, when processed during the first inclusion of the header (source file), causes all subsequent **#include** directives on that header (source file) to re-read the contents of the header (source file).
	opspace	Allow whitespace between multi character assignment operators.
	pluscomment	Enable C++ comments.
	plusplus	Preprocess for C++. See *Additional Processing* below.
	plussplice	Allows non-standard backslash.
	predefined	Macros defined after this pragma are marked as predefined.
	prefix	Enable prefixinclude feature to look for nested include files using the directory prefix from the parent's include directive. This option is on by default when -I- is specified and supports **#include "file.h"** compatibility when using -I-. When enabled, if main.c uses **#include "dir/file1.h"** and file1.h uses **#include "file2.h"**, the preprocessor first searches for dir/file2.h and if not found searches for file2.h. The directory prefix may also be inherited if the parent file itself inherited the prefix. When not enabled no directory prefix is ever inherited.

#pragma [*pass*:][**no**]*option* [*args* ...] (cont.)	**reguard**	Output all #define statements to re-guard included header files.
	reserved *name*[=*value*] ...	The *name* arguments are marked as reserved keywords. *value* optionally specifies the keyword lexical value. This option is used when compiler front ends are linked directly with the preprocessor. The "classic" C and C++ keywords have predefined lexical values, as do **asm**, **const**, **enum**, **signed**, **void** and **volatile**. If *value* is omitted for a keyword that has no predefined lexical value then **NOISE** is assumed. *value* may be one of: **GROUP** A group noise token that may be followed by zero or one balanced **(...)** groups and zero or one balanced **{...}** groups (e.g., **asm**). **LINE** A line noise token terminated by the next **newline**. **NOISE** A noise token to be ignored (e.g., **near**). **STATEMENT** A statement noise token terminated by the next semicolon.
	spaceout	In the **ansi** dialect for the standalone cpp **spaceout** 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. This mode is required by some **asm** implementations that allow the assembly text to be preprocessed.
	splicecat	**\newline** line splicing may be used to concatenate tokens in macros definition.
	standard "*dir*"	Names *dir* as the standard include directory.
	strict	Set the strict interpretation mode.
	stringspan	Allow \newline line in strings.

#pragma [*pass*:][**no**]*option* [*args* ...] (cont.)	**suffixsunwcch**	Enable header file search rules emulating the standard header implementation of certain Sun C++ compilers operating under certain versions of the Solaris operating environment. In this mode, the normal include file search rules are modified for the standard C and C++ header files only, and only for included files which appear in angle brackets and without any path specified. To search for one of these header files, cpp first appends the suffix **.SUNWCCh** to the requested name, and searches for that name. If that file is found and is a symbolic link, cpp dereferences the link exactly once and uses the dereferenced path as the included file. If the search using the rewritten name fails, cpp searches again using the original file name as specified in the **#include** directive. For Sun C++ 6.0, this behavior is documented in the Sun C++ User's Guide, Chapter 5 ``Using Libraries,'' section 5.7.4, ``Standard Header Implementation.''
	text	**notext** suppresses output to *ofile* generated by input text. This allows files to be scanned for cpp directives without generating any text output. Note that the **–P** option must still be used to suppress line number information output.
	transition	Sets transition mode. Also sets the compatibility dialect.
	truncate	Truncate macro names for compatibility with non-flexname (8-characters) compilation systems.
	version	Outputs both **#pragma pp:version** *version-string* and **#pragma version**, allowing later passes to omit similar version pragmas.
	warn	Produce warnings about extensions used in non-hosted files in the strict dialect.
#rename *oldname newname*	Changes the name of the macro *oldname* to *newname*.	
#undef *name*	Remove the definition of the macro *name* (if any).	

#undef *#predicate(argument)* **#undef** *#predicate()*	The first form removes the assertion of *predicate(argument)*, if any, while the second form removes all assertions on *predicate*.
#warning [*message*]	Emits *message* as a cpp warning diagnostic and continues normal processing. The default *message* is **user warning**.

Builtin Macro

The builtin macro **#(***[op]identifier***...)** provides access to preprocess time symbols and definitions. The value of this macro is enclosed in "" unless otherwise noted. Just as with the # and ## operators, any macro formal arguments appearing within **#(...)** in a macro definition are copied without expansion on macro invocation. *arg* may be one of the following:

FILE	The current file name. __**FILE**__ is defined to be **#(FILE)** for ANSI conformance.
LINE	The current line number (not quoted). __**LINE**__ is defined to be **#(LINE)** for ANSI conformance.
DATE	The current month, day and year *(***MMMDDYYYY***).* __**DATE**__ is defined to be **#(DATE)** for ANSI conformance.
TIME	The current time *(***HH:MM:SS***).* __**TIME**__ is defined to be **#(TIME)** for ANSI conformance.
BASE	The base name of **FILE**.
PATH	The full path name of the most recent **#include** directive or **exists** predicate evaluation. __**PATH**__ is defined to be **#(PATH)** in deference to ANSI conformance.
VERSION	The *cpp* version stamp. __**VERSION**__ is defined to be **#(VERSION)** in deference to ANSI conformance.
ARGC	The number of arguments of a variable arguments macro. __**ARGC**__ is defined to be **#(ARGC)** for ANSI conformance.
getenv*name*	The value of *name* as returned by the getenv(3) library call.
getmac*name*	The definition of the preprocessor macro *name*. Notice that macro formal names appearing in macro definitions are replaced by internal format token sequences.
getopt*option*	The setting for the option or pragma *option*.
getprd*predicate*	The argument associated with *predicate* from the most recent assertion on *predicate*.

Default Definitions

#include "pp_default.h" is automatically executed before the first line of *ifile* is read using the **–I–R** mechanism described above. A file other than **pp_default.h** may be specified using the **–I–D** option. **pp_default.h** typically contains **#define** directives that describe the current hardware and software environment. By using the **–I***dir* or **–I–D***file* options different **pp_default.h** files may be referenced to support cross-compilation.

Proposed standard assertions for **pp_default.h** are:

system(*system-name***)**	Defines the operating system name. Example values for *system-name* are **unix**, **vms** and **msdos**.
release(*system-release***)**	Defines the operating system release name. Example values for *system-release* are **hpux**, **bsd**, **svr4**, **sun**, **uts**, and **xinu**.
version(*release-version***)**	Defines the operating system release version. Example values for *release-version* are **4.1c** and **4.3** for **release(bsd)**, **8** and **9** for **release(research)** and **3.0** etc. for **release(V)**.
model(*model-name***)**	Defines the hardware model or workstation name. Example values for *model-name* are **apollo**, **sun**, **ibm-pc** and **unix-pc**.
architecture(*architecture-name***)**	Defines the processor architecture name. Example values for *architecture-name* are **u3b**, **m68000**, **ibm**, **pdp11**, and **vax**.
machine(*architecture-version***)**	Defines the processor architecture version. Example values for *architecture-version* are **2**, **20** and **20s** for **architecture(3b)**, **70** etc. for **architecture(pdp11)** and **750**, **780** and **micro** for **architecture(vax)**.
addressing(*addressing-mode***)**	Defines the addressing mode, useful for PC compiler implementations. Example values for *addressing-mode* are: **small**, **medium**, **large**, **segmented** and **unsegmented**.

Additional Processing

Adjacent "" string constants appearing in the text are concatenated. However, strings are not concatenated in directives.

The new character escapes **\a**, **\v** and **\x...** are converted to octal notation for compatibility with older passes. This will disappear as ANSI C support becomes more pervasive.

The **::**, **.*** and **–>*** operators and **//**... comments are recognized for the C++ language.

Compatibility

The **compatibility** dialect supports pervasive *Reiserisms* that will be hard to shake out of old code as the ANSI standard arrives. Compatibility support includes:

- **#assert** and **#unassert** are supported by maps to **#define** # and **#assert#**.
- the *disappearing comments* trick used to concatenate tokens within macro definitions (this trick does not work outside of macro definitions, but at least a diagnostic is produced). If **pp:splicecat** is set then the line splice sequence **\newline** may also be used to concatenate tokens in macro definitions, otherwise **\newline** translates to space.
- **vertical-tab** is treated as **space** in directive lines
- **formfeed** is treated as a **newline** character (although the line count is not incremented by **formfeed**)
- macro formal arguments appearing within "" or ' ' quotes in macro definitions are replaced by the corresponding actual argument text
- macro call arguments are not expanded before being placed in the macro body text
- trailing characters in directives are silently ignored

Files

/usr/include	standard directory for **#include** files
/usr/local/include	sometimes searched after the standard directory during initialization
pp_default.h	predefined symbols and assertions

Related Commands

cc(1), egrep(1), m4(1), nmake, probe, proto, getenv(3), pp(3), strcmp(3)

See Also

American National Standard for Information Systems--Programming Language C, ANSI X3.159-1989.

Diagnostics

The error messages produced by cpp are intended to be self-explanatory. The line number and file name are printed along with the diagnostic.

predefined, **readonly** and **active** macro diagnostics may surprise old cpp users.

iffe Command

iffe – C compilation environment feature probe

Synopsis

iffe [-] [*op***[** *,op***...] [** *arg* **[** *,arg***...] [***reference***...]]] [:...]**

Description

iffe probes the C compilation environment for features. A feature is any file, option or symbol that controls or is controlled by the C compiler. iffe input is line oriented. Input lines may appear as command arguments with the argument **:** acting as a line delimiter. Unless specified by set out *file* the first *arg* determines the output file name FEATURE/*arg. The* **–** option sends the output to stdout.

Files

iffe/*op*.c	C program to compile and run for *op*. Execution output copied to the FEATURE output file.
iffe/*op*.sh	sh(1) script to run for *op*. Execution output copied to the FEATURE output file.
iffe/*op*	iffe override tests for *op*.

Related Commands

cc(1), cpp, nmake, probe

ignore Command

ignore – ignore exit status of shell commands

Synopsis

ignore *shell-command*

Description

ignore causes the exit status of shell commands to be ignored.

Related Command

silent

nmake Command

nmake – maintain and update files

Synopsis

nmake [[*option*] [*script*] [*target*] ...]

Description

nmake reads input makefiles and triggers shell actions to build target files that are out-of-date with prerequisite files. Most information used to build targets is contained in the global base rules that are augmented by user makefiles.

Each *option* argument is preceded by **–** or **+** and follows getopt(3) conventions, with the exception that an option may appear in any position on the command line (see the *Options* section). Each *script* argument is a non-option argument that contains at least one of **space**, **tab**, **newline**, **:**, **=**, **"** or **** and is read as if it were a complete, preprocessed makefile. The *target* arguments are made in order from left to right, overriding the default main targets defined in the makefiles (see the *Engine Control* section).

Makefiles

Makefiles are the main source of input to nmake. A makefile contains a sequence of assertions and variable assignments that describe target files and their prerequisites. All parsing is ordered from left to right, top to bottom.

At least one makefile must be specified. If no file options are present then nmake attempts to read, in order, Makefile or makefile. The first line of the first makefile determines the base rules context. If it is of the form

rules ["*base-rules*"]

then *base-rules* will be used rather than the default makerules. If "*base-rules*" is not specified then no base rules are used. Omitting rules is equivalent to specifying rules "makerules". The actual base rules file, which must be a compiled global makefile, is found by changing the *base-rules* suffix to .mo and binding the resulting name using the directories of .SOURCE.mk (see the *Binding* section). Also, for compiled makefiles, the $(MAKERULES) engine variable should be set to any alternate base-rules name (see the *Engine Control* section). The following makefile descriptions rely on nmake engine constructs and are independent of the base rules context.

If a makefile contains cpp directives then it is first passed through a modified ANSI cpp. Directives and C-style comments are treated as usual, but cpp macros are only expanded on directive lines. # preceded by zero or more space characters (**space** or **tab**) starting in column 1 is interpreted by cpp, otherwise text between # (preceded by at least one space character) and **newline** is treated as a comment by nmake. Blank lines are uniformly ignored.

After all the makefiles have been read in, and before any command line targets are made, the collective makefile information is automatically compiled into a single nmake object file. The next time nmake executes this object file is read in place of the makefiles. The object file is automatically regenerated whenever the makefiles or their prerequisites (.e.g., **include** dependencies) have changed. The nmake object file name is *base***.**mo where *base* is the base name of the first non-global makefile or null.mo if the first non-global makefile name is **–**. For better performance makefiles specified by the global option should be precompiled nmake object files. Refer to the compile option below for more information.

Variables

Variables are assigned by

variable = *value*

where *variable* may be any sequence of letters, digits, underscores and **dot**s. Depending on the context (e.g., **:** dependency, operator dependency, assignment, action), subsequent appearances of $(*variable*) expand to value and $$(*variable*) expands to $(*variable*) , otherwise $ is passed untouched. value is not expanded until $(*variable*) is encountered (see the *Variable Editing* section). *variable* := *value* causes *value* to be expanded before assigning it to *variable* and *variable* += *value* appends the expanded *value* to the current value of *variable* . *variable* == *value* assigns *value* to *variable* and also marks *variable* as a candidate implicit state variable prerequisite. variable &= value defines the hidden (auxiliary) value of *variable* . The auxiliary value is not saved in the statefile. The expansion of variable will include the primary and auxiliary values (see the V edit operator in the *Variable Editing* section)**.**

Variable assignments come from many sources. The precedence order (highest to lowest) is:

automatic	Variables maintained by nmake (see Automatic Variables)
dynamic	Assignments done while building targets
command line	Assignments in command line *scripts*
import variables	Colon separated environment variable names listed in the value of the MAKEIMPORT variable (see the *Environment* section)
makefile	Normal makefile assignments
environment	Variables defined in the environment (see env(1))

global makefile	Includes base rule assignments

Variable names containing **dot** cannot conflict with environment variables set by the shell; such variables are typically used by the base rules. To avoid base rule conflicts users should not define upper case variable names with **dot** as both the first and last character.

Assertions

Assertions specify dependency relationships between targets and prerequisites and provide actions that may be executed to build targets that are out-of-date with their prerequisites. An individual assertion is a list of target atoms, a *dependency operator* (see the *Assertion Operators* section), and an ordered list of prerequisite atoms for the targets. Subsequent lines with an indentation level greater than the first target comprise the action. The target list, the prerequisite list and the action may be empty. Variables in the target and prerequisite lists are expanded when the assertion is read, whereas variables in the action are not expanded until the action is executed. For portability only **tab** characters should be used for action indentation.

A single assertion with multiple targets associates the prerequisite list and action with the targets as if each target were in a separate assertion. Single assertions are more efficient in that the storage used by the action is shared among the targets.

An atom may appear as a target in more than one assertion. Prerequisites from successive assertions are simply appended to the prerequisite list of the target, with the exception that duplicate prerequisites are deleted (from the right) at assertion time. Only one action may be specified per target. The collection of all assertions for a given target is called the *rule* for that target. Atom names containing the special characters :, #, = and + must be enclosed in double quotes. For example:

```
target : ":file1" file2
	action
```

Binding

In the process of making a target atom nmake *binds* the atom to a *file*, a *virtual atom*, a *state variable* or a *label*. This binding remains in effect for the entire nmake execution.

A *state variable* is an atom associated with an nmake variable that holds the variable value and the time the variable last changed. This information is retained in the *statefile base*.ms where *base* is the base name of the first makefile. Statefiles are automatically generated and loaded using the nmake object file algorithms. The state variable atom for the nmake variable *variable* is **(***variable***)***.*

For example:

```
x.o : (DEBUG)
```

specifies that x.o depends on the definition of the variable DEBUG .
The contents of the statefile can be listed by executing

```
nmake –f base.ms –blr
```

Atoms are bound to files using the prerequisites of the special .SOURCE and .SOURCE.*pattern atoms* **(**see the *Special Atoms* section). The prerequisites of these atoms are ordered lists of directories to be scanned when searching for files.

A *virtual* atom binds neither to a file nor to a state variable. Virtual atoms are declared using the .VIRTUAL attribute**.** Virtual atom times are retained in the statefile.

Make Algorithm

The steps taken to make a target atom are:

1. bind the atom
   An atom retains its binding until nmake exits, unless it is explicitly unbound.
2. check if atom already made
   No work required for atoms that have already been made. If atom is active (action executing), block until its action completes before returning.
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. Notice that this type of rebuild may result in empty $(>) expansions in non-metarule actions.
4. check .INSERT and .APPEND
   Target prerequisites are modified according to these Special Atoms.
5. make explicit prerequisites
   The explicit prerequisites are (recursively) made from left to right. The most recent prerequisite time is saved.
6. make implicit prerequisites
   If the target has no explicit action and no explicit prerequisites, and a metarule can be applied to generate the target, the metarule prerequisites are (recursively) made from left to right. Again, the most recent prerequisite time is saved.
7. check if out-of-date
   If the most recent prerequisite is newer than the target, or if the target information is not consistent with the statefile, the target action is triggered to build the target.
8. make scan prerequisites
   If the target atom is a file with a .SCAN.*x* attribute (see the *Special Atoms* section), the scan prerequisites are (recursively) made from left to right. The scan prerequisites are determined either from the statefile information (if consistent) or by reading the contents of the file using scan strategy *.x*. The time noted for the target atom is the most recent of its own and all (recursively) of its scan prerequisite times. The scan prerequisites are saved in the statefile to avoid scanning during the next nmake execution.
9. sync statefile information
   When the target is built (action, if any, completed) its time, action, and prerequisites are saved in the statefile in preparation for the next nmake execution.

Engine Control

The global flow of control in the nmake engine is, in order:

1. read the args file
   The first file in $(MAKEARGS) (see the *Environment* section) that exists in the current directory is read, and the contents are inserted into the command-line argument list.
2. read the command-line arguments
   The *options* and command-line *script* arguments are parsed.
3. read the initialization script
   In addition to the variables listed in the *Environment* section, the Special Atoms .VIEW and .SOURCE.mk are initialized.
4. read the base rules
   Determine which base rules are to be used (either user-defined or default) by checking the first line of the first explicit makefile for the rules statement. The first explicit makefile is determined by first ckecking the command-line for the **–f** file option. If the **–f** option is not found, the first file listed in

$(MAKEFILES) (see the *Environment* section) that exists is checked. If the rules statement is not found in that file, the default base rules, $(MAKERULES) (see the *Environment* section) are used.

5. read the global makefiles
6. read the explicit makefiles
   If no file options are given, the files listed in $(MAKEFILES) (see the *Environment* section) are used.
7. compile makefiles to form the nmake object file
   If the nmake object file is out-of-date with the input makefiles, the makefiles are recompiled.
8. read the statefile
   The statefile is loaded as an nmake object file.
9. initialize .ARGS
   The prerequisites of .ARGS are set to the list of command-line *targets*.
10. make .MAKEINIT, if defined
11. make .INIT, if defined
12. check .ARGS
    If .ARGS has no prerequisites, append the prerequisites of .MAIN onto .ARGS.
13. make the prerequisites of .ARGS
    This constitutes the actions requested either on the command line or in the makefile, subject to the actions of .MAKEINIT and .INIT.
14. make .DONE, if defined
15. make .MAKEDONE, if defined

Programming Constructs

Stuctured programming constructs may appear inside .MAKE actions or outside assertions. The construct keywords must be the first word on a line. Atom names matching any of the keywords must be quoted (when the first word on a line) to avoid conflicts.

Each *expression* may be a combination of arithmetic and string expressions where the string expression components must be quoted. "..." strings use shell pattern matching (e.g., "*string*" == "*pattern*"). Empty strings evaluate to 0 in arithmetic expressions and non-empty strings evaluate to non-zero. Expression components may also contain optional variable assignments. All computations are done using signed long integers.

break	Breaks out of the **for** and **while** loops and resumes execution after the **end** statement.

error *level message*	*message* is output as an nmake error message with severity level *level*. *level* may be one of: < **0** debug trace -- output only if *level* <= **debug**. **0** information **1** warning **2** error -- no exit >= **3** error -- exit with code (*level*–2)
eval **...** **end**	The statements between **eval** and **end** are expanded an additional time. **eval ... end** pairs may nest to cause more than one additional expansion.

for *variable patterns ...*
...
end
	The statements between **for** and **end** are executed with *variable* assigned to the name of each atom matching one of the shell *patterns*. A **break** statement breaks out of the loop and resumes execution after the **end** statement.

if *expression*
...
elif *expression*
...
else
...
end
	Nested **if** conditional construct.

include [–] *path*	Include the file specified by *path*. If – is present, then the file is optionally included (no warning is generated if the file is not found).
let *variable=expression*	Sets the value of *variable* to the numeric value of *expression*.
local *var1 var2...* **local** *name=value*	Declares variables local to the current action.
print –n –u[*0-9***] –f** *format* **[+–]o** *file* **- -** *data*	**–n** adds a trailing null character at the end of the output string. **–u** sends output to the specified file descriptor (e.g., **0** is stdin, **1** is stdout, **2** is stderr). **–f** specifies the output *format*, where *format* is a C printf-like format string. **–o** *file* means to open *file* for writing, while **+o** *file* means to open *file* for appending. **- -** ends the arguments, anything after **- -** is considered *data*. *data* is the output string data.
read -i *file variable*	Read input from *file* and assign the contents of the file to *variable*.

return [*expression*]	Return from the action with results specified by *expression*: *omitted* If *expression* is omitted then 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. If the **rules** statement is specified, it must be the first line of the first makefile. It will overwrite the context of the default base rules. Omitting the **rules** statement is equivalent to specifying **rules "makerules"**. The **rules** statement with no file name specified is a request that no base rules file is to be used.
set [**no**]*name*=[*value*]	Sets options using option-by-name format (see the *Options* section)

while *expression*
...
end
	while loop construct. A **break** statement breaks out of the loop and resumes execution after the **end** statement.

Metarules

Based on metarule patterns, nmake can infer prerequisites for files that have no explicit actions or prerequisites. The prototype metarule pattern is *prefix%suffix* and is matched by any string containing the non-overlapping strings *prefix* at the beginning and *suffix* at the end. **%** matches the remaining characters and is called the *stem*.

The following makefile specifies that program depends on two files a.o and b.o , and that they in turn depend on .c *files and a common file* header.h .

```
program : a.o b.o
        cc a.o b.o libx.a –lm –o program
a.o : header.h a.c
        cc –c a.c
b.o : header.h b.c
        cc –c b.c
```

The *metarule* to create a file with suffix .s2 that depends on a file *(*with the same base name) with suffix .s1 is %.s2 : %.s1 . Any prerequisites following %.s1 are transferred to each target when the metarule is applied. For example, a metarule for making optimized *.o* files from *.c* files is

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

The $(<) variable is the current target, which is the generated .o file in this case. The $(>) variable is the primary metarule prerequisite, which is the .c source file. Note that $(>) has a different meaning in non-metarule rules (see Automatic Variables).

Notice that the .o targets also depend on the values of the CC and CCFLAGS nmake variables. If the current target is a.o then nmake infers the following from the %.o : %.c metarule:

```
a.o : a.c (CC) (CCFLAGS)
        cc –O –c –o a.o a.c
```

In this case the *stem* is a**,** $(CC) and $(CCFLAGS) expand to cc and –O**,** respectively.

Assuming the %.o : %.c metarule has been asserted, the example can be stated more briefly:

```
program : a.o b.o
        $(CC) $(*) libx.a –lm –o $(<)
a.o b.o : header.h
```

Metarules are applied according to the order of the patterns listed as the prerequisites of the .METARULE atom. Pattern order is significant; the first possible name for which both a file and a metarule exist is inferred.

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.

Metarules with the % target pattern are called **unconstrained** metarules and are subject to additional constraints that control metarule chaining. Unconstrained metarules with the .TERMINAL attribute are applied only when the prerequisite pattern matches an existing file. Otherwise an unconstrained metarule is applied only if there exists no other metarule (other than a unconstrained metarule) for which the target pattern matches the current target. For example

```
% : .TERMINAL RCS/%,v
        $(CO) $(COFLAGS) $(>)
```

is a metarule that generates source files from RCS version files in the RCS subdirectory.

Variable Editing

Edit operators allow variable values to be tested and modified during expansion. The expansion syntax is:

$(variable[:[@]op[sep arg]]...)

The operator groups, each preceded by **:**, are applied in order from left to right to each space separated token in the expanded variable *value*. The operator groups form a token pipeline where the output (*returned* or *selected* tokens) of any operator group becomes the input for the next operator group. **newline** is treated as a separate token. **"**, **'** and \ quote space characters, but they are considered part of the token and are not removed. If @ immediately precedes an *op*, then the entire *value* is treated as a single token for that operator. *sep* is usually **=**, but, where appropriate, some operators support **!=** and the arithmetic comparisons **<**, **<=**, **>=** and **>**.

The expansion algorithm first expands the variable name *variable* to determine the value to be edited. This value and the operator expressions ([:[@]*op*[*sep arg*]]...) are then expanded before the edit operators are applied. The ultimate expansion is formed by applying each operator to each token, separating adjacent results by a single **space** character.

$(*var1***|***var2*...) expands the value of the first (left to right) non-**null** valued variable. If the last variable name is enclosed in **""** then this string is used as the value if all preceding variables have **null** values. The standard **** character constants are interpreted within the string.

Some operators *select* tokens by simply returning the token value as the result; non-selected tokens produce **null** (the empty string). The operators are:

A[*sep expression*] A>*pattern*	Selects tokens having one or more attributes or prerequisites listed in *expression*. The tokens are treated as atoms. In case of *expression* being a space or vertical bar separated list of *attributes*, if *sep* is [!]=, then atoms that [*do not*] have any of the listed *attributes* are [*not*] selected (see the *Special Atoms* section) . *Attributes* defined through .ATTRIBUTE Special Atom may also be used. If *sep* is <, then the pattern association rule for the rule inferred from the input token and the pattern association rule base name specified in *attribute1* of *expression* is returned. For example, $("stdio.h":A<.SOURCE.) would give .SOURCE.%.LCL.INCLUDE. A alone expands to the list of applicable user-defined attributes. In the form A>*pattern*, selects target tokens with specific prerequisite patterns.

B*[=base]* D*[=directory]* S[=*suffix*]	As stated previously, the edit operators form token pipelines. The file component edit operators (D B S) are the exception to the rule; they do not form token pipelines because nmake treats the file component edit operators as a group (although separated by **:**). Each is applied as a single edit operation. Pathnames are partitioned into three (possibly **null**) components. *directory* includes all characters up to but not including the last ***/***. *base* includes all characters after the last ***/*** up to but not including the last **.**. *suffix* includes all characters from the last **.** on. Multiple **.**'s are treated as a single **.** and if **.** is the first character after the last ***/*** and is the only **.** then it is included in *base* rather than *suffix*. If the pipeline result is desired, extra **:** must be specified (e.g., :D::B will apply B edit operation to the result of D edit operation).	
C*<del>old<del>* *new<del>***[**G**]**	Similar to the ed (1) substitute command. Substitutes the first occurrence of the string *old* with the string *new* in each token. A trailing G causes all occurrences of *old* to be substituted. <*del*> may be any delimiter character. **C/** may be abbreviated as ***/***.	
E	Evaluates the complete input string as a logical, string, or integer expression.	
F=*format*	Converts tokens according to *format*. *format* may be a concatenation of the following	
	L	The token is converted to lower case.
	U	The token is converted to upper case.
	V	The token is converted to the valid nmake variable name.
	%[–][*n*][.*m*]*c*	printf(3) style formatting. Only the *d*, *e*, *E*, *f*, *F*, *g*, *G*, *o*, *s*, *u* and *x* conversions are supported.
F=*%(format)*T	Allows the user to display the *format* that will be used to display time.	
G=*pattern*	Selects token files that can build (generate) files that match the metarule pattern *pattern* using the metarules. For example, a token **x**.*y* is selected if a metarule **%**.*s***:%.y** has been asserted.	
H [*sep*][U]	Heap sorts the tokens in ascending (if *sep* is < or default) or descending (if *sep* is >) order. <= *sep* will give low to high numeric sort, and >= *sep* will give high to low numeric sort. If U is specified, the sort is unique (the duplicates are removed).	

I=*list*	*list* is expanded and directory tokens in *variable* that also appear in the expanded value of *list* are selected. Each selected directory will appear at most once in the return value. The directory path names are canonicalized before comparison.	
K=*pfx*	Splits long lines like xargs(1) into a number of tokens, applying optional *pfx* to each token.	
L [*sep* [<*pattern*>]]	Each token is treated as a directory name. For each token, the list of all files in the specified directories which match the optional *pattern* are returned. *sep* may be <= or >= and is used to specify that the list should be sorted in ascending or descending order respectively. The default order is unsorted.	
M[!]=*pattern*	Selects tokens [*not*] matching the egrep(1) style regular expression *pattern*.	
N[!]=*pattern*	Selects tokens [*not*] matching the sh(1) file match expression *pattern*. Multiple patterns separated by **|** denotes the inclusive or of the individual patterns.	
O[!]|[<*relop*><*n*>]	Each token is numbered by its left to right position, starting with 1. Tokens with positions satisfying the integer expression *position relop n* are selected. *relop* may be one of **<**, **<=**, **=**, **!=**, **>=** or **>**. If O is specified alone, the output is the number count of tokens, O! gives the string length of each token.	
P<relop>*op*	Treats the tokens as file path names and applies the path name operator *op*. Depending on the value of *op*, *relop* may be one of **<**, **<=**, **=**, **!=**, **>=** or **>**. *op* may be:	
	A	Returns the absolute path name for specified files.
	B	Returns physically bound tokens.
	C	Returns the canonicalized path name. **.**'s and redundant **/**'s are removed (unless **.** is the only remaining character). Each **..** cancels the path name component to its left (unless that component is also **..**); **..**'s are moved to the front of the name. **/..** forms are preserved in deference to some remote file system implementations.
	D	For each token that is a bound atom, the directory where the atom was bound is returned.

P<relop>*op* (cont.)	H[=*suffix*]	Generates 9 character hash file name (or up to 14 character hash file name if *suffix* is used — takes first 5 characters in *suffix* if it is greater than 5 characters) for the given token.
	I=*file*	Selects tokens which are existing files and which have the same device and inode numbers as *file*. If P!=I=*file* is used, then those tokens which are existing files and are not the same file as *file* are returned.
	L[*level*]	Selects tokens that are atoms bound in the level 0 [*level*] view. Views are defined using the .VIEW Special Atom.
	L*	Returns all the views of a bound file.
	L**!**	Returns the first occurrence of a bound file from all the views.
	P=*lang*	Returns the pathname for the probe information file for the language specified by *lang* and the language processor specified in the input token.
	R	For each token, treated as a directory path name, a path name that leads to the (possibly relative) root of the token is returned. The return value is either **.** or a path name consisting of **..** components.
	S	The bound name for each token that is bound to a file within a subdirectory of its view is returned.
	U	Returns the unbound name for each token.
	V	Returns the view directory path name of the current directory for each token bound to a file (see also .VIEW Special Atom). Tokens not bound expand to null**.**
	VL	Returns the view local path name for each token that is an atom bound to a file. Paths in the viewpath are canonicalized and converted to their corresponding local path relative to the current directory. Paths outside the viewpath are returned unmodified. Unbound paths return null.
	X	Returns each token that is the path name of an existing file. The tokens are not bound.

Q	Each token is quoted for literal interpretation by sh(1).	
R	Each token is parsed as a makefile, each in a separate context. **null** is returned.	

T=*type*
T=*type***?***return*
	The tokens are bound to atoms (unless stated otherwise) and are operated on according to *type*. **?***return* replaces the default non-**null** return value with the expanded value of *return* for selected atoms and **null** otherwise. *return* is only expanded (a *second* expansion: the operators have already been expanded once) if the test succeeds. If *type* is preceded by X, then no binding is done. *type* may be one of:	

	A	Returns the archive update action for each atom with the .ARCHIVE attribute. If the returned action is **non-**null then it must be executed for any archive that has been modified or copied before the archive is used by the compilers or loaders (e.g., ranlib(1) on BSD-based systems).
	D	The *cc*(1) style definition of each token that binds to a state variable is returned. Given: DEBUG = TEST = 1 SIZE = 13 STATEVARS = (DEBUG) (TEST) (SIZE) $(STATEVARS:T=D) expands to -DTEST -DSIZE=13.
	E	Similar to T=D except that the expanded definitions are *name*=*value* pairs. Given the above example, $(STATEVARS:T=E) expands to DEBUG= TEST=1 SIZE=13.
	F	Each atom that binds to a file is selected. The bound atom name is returned.

T=*type*
T=*type***?***return*
(cont.)
	G	Each atom bound to a file that has been built (generated) is selected. The bound atom name is returned.

	I**[–]**	Each input token is the name of an input file to be read. The contents of all the files are returned (including any newlines). A – following the **I** inhibits the expansion of variable names in the file.
	M	Generates the parentage chain of each active token.
	N	If the unbound value of *variable* is **null** then **1** is returned, otherwise **null** is returned.
	O[*mode*][=*text*]	Each input token is the name of the file to be written according to the specified *mode .* If no *mode* is specified, *text* overrides the contents of each file. By default, the new line separator is attached to the end of the *text.* This edit operator is the complement to T=I**[–].** *mode* may be: **+** Appends *text.* **–** Does not put the new line separator at the end of *text.*
	P	Each atom that binds to a file and is also not a symbolic link is selected. The bound atom name is returned. If symbolic links (link(2)) are not implemented then :T=P: is equivalent to :T=F:.
	Q	Each atom that exists is selected.
	QV	Returns defined variables.
	R	Each token is treated as an atom and the relative time (number of seconds since the Epoch) of each atom is returned. The form $("":T=R) operates on the current date.

T=*type*
T=*type***?***return*
(cont.)87
	S[*conv*[*data*]]	Converts each atom to a new atom type specified by *conv* and *data* (**null** is returned when the conversion specifies an undefined atom). *conv* may be:
		A Given the internal name for a state rule, convert it to the original name.
		F Forces creation of a new atom. Used in conjunction with the other conversion operators.
		M Returns the metarule name that generates files matching the metarule pattern *data* from files matching the metarule pattern named by each token.
		P Returns the alternate prerequisite state atom.
		R Returns the primary state atom. This is the default when *conv* is omitted.
		V Returns the state variable atom for each token that names a variable.

	T	The format for this operator is T*relop*T*atom*. A comparison is done between the time associated with *atom* and the times associated with the atoms in the token list. The tokens whose times are *relop atom* are returned. *relop* can be <, >, =, <=, >=, or !=. If *atom* is not specified, it defaults to the current target. For example, $(*:T>T) lists the prerequisites whose times are newer than the time for the current target.
	U	Returns the atom or variable name for each state atom token.
	V	If the unbound value of *variable* is non-**null** then **1** is returned, otherwise **null** is returned.
	W	If a type operator is forcing a file to be bound, and the W operator is applied, do not wait for the bind to complete.
	X	Binding should not be done. Used in conjunction with other type edit operators.
	Z[CERS]	Returns the cancel (C), event (E), relative(R), or state(S) time for the input token.

V[*type*]	The *value* of *variable* is used without expansion. *type* may be:	
	A	Expands to the auxiliary (assigned to *variable* with &= assignment operator) *value* of *variable* (see the *Variables* section).
	P	Expands to the primary *value* of *variable.*
X=*list*	The directory cross product of the tokens in *variable* and the tokens in the expanded value of *list* is returned. The first **.** *lhs* operand produces a **.** in the cross product. All *rhs* absolute path (rooted at ***/***) operands are collected, in order, after all other products have been computed, regardless of the *lhs* operands.	
Y*<del>non-null<del>null<del>*	If *value* is **null** then *null* is expanded and returned, otherwise *non-null* is expanded and returned. <*del*> may be any delimiter character. Y? may be abbreviated as ?.	

To illustrate some of the above operators:

```
FILES = a.h b.h c.h x.c y.c z.c
HEADERS = $(FILES:N=*.h)
$(HEADERS:/^/-I/)      →  -Ia.h -Ib.h -Ic.h
$(FILES:N=*.c:/ /:/G)  →  x.c:y.c:z.c
```

Automatic Variables

The following variables are automatically defined and updated by nmake:

$(–)	The current option settings suitable for use on nmake command lines. The options are listed using the –o option-by-name style and only those option settings different from the default are listed
$(–*option*)	**1** if the named *option* is set and non-zero, otherwise **null**.
$(+)	The current option settings suitable for use by **set**. Only those option settings different from the default are listed.
$(+*option*)	The current setting for the named *option*, suitable for use by **set**.
$(=)	The list of command line *script* arguments and assignments for variables that are prerequisites of the atom .EXPORT.
$(<)	The current target name.

$(>)	In regular rules $(>) is the list of all explicit file prerequisites of the current target that are out-of-date with the target or new. It may be null even if the current target action has triggered. In metarules $(>) is the first %-pattern prerequisite, also called the primary metarule prerequisite, and is never null.
$(%)	The *stem* of the current metarule match, or the arguments of a function (see .FUNCTION in the *Special Atoms* section).
$(*)	The list of all explicit file prerequisites of the current target.
$(~)	The list of all explicit prerequisites of the current target.
$(&)	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 list of all implicit and explicit file prerequisites of the current target. Implicit file prerequisites are generated by the language dependent scan.
$(?)	The list of all prerequisites of the current target. This includes all implicit and explicit state variable and file prerequisites.
$(#)	The number of actual arguments in the following constructs: local -[n] arg ... local (formal ...) actual
$(@)	The action for the current target.
$(^)	If the current target has been bound to a file in other than the top view then $(^) is set to the original (lower view) binding and $(<) is set to new (top view) binding, otherwise $(^) is **null**.
$(. . .)	Represents all the atoms, rules and state variables used when making .MAKEINIT.
$(;)	If the current target is a state variable then $(;) is the state variable value, otherwise it is the unbound atom name.

Each repeated occurrence of the automatic variable name character in the variable expansion causes the *parent* of the current target to be used as a reference. For example, $(<<) is the name of the parent of the current target and $(**) is the list of all its prerequisites. $(c*atom*) references information for *atom* instead of the current target. Notice that .IGNORE, .STATE, .USE and .VIRTUAL atoms are not included in $(<), $(>), $(*), or $(!) automatic variable expansions.

Assertion Operators

Assertion operators provide fine control over makefile assertions. Each operator is an atom whose action is executed whenever the atom appears as an operator in an assertion. Assertion operators are defined by assertions:

```
":operator:" : .OPERATOR
        action
```

where the operator name syntax is :: and :*identifier*: and the operator name must be quoted in its defining assertion. An operator is activated by an assertion of the form:

```
lhs  :operator:  rhs
        commands
```

The operator action is executed with $(<) set to *lhs*, $(>) set to *rhs*, and $(@) set to *commands*. No variable expansion is done on *lhs*, *rhs*, or *commands*.

Special Atoms

The Special Atoms defined by nmake all have the form .ID, where "ID" is any string of capital letters, dots, or numbers. Users can define their own Special Atoms, which do not have to begin with dot (.). The following atoms are special to nmake and fall into one or more of these types, depending on the context:

Action Rule	Action used for nmake control.
Assertion Attribute	Provides fine control over **:** dependency operator assertions.
Dynamic Attribute	Assigned to target atoms by listing attribute names as prerequisites in assertions. Dynamic atom attributes may be tested using the A**=***attribute***:** edit operator.
Dynamic List	Prerequisites are used to control nmake actions.
Immediate Rule	Invokes an nmake action when appearing as a target in assertions *at assertion time .* The prerequisites and actions are cleared after the assertion.
Pattern Association	The meaning of the attribute is applied to the atoms matching the specified *pattern*. *pattern* is given in one of the following formats: string match (SOURCE.%.c, .BIND.-l%), suffix match (.SOURCE.c), or attribute match (.INSERT.%.ARCHIVE).
Readonly Attribute	Maintained by nmake and may not be explicitly assigned.
Readonly List	Prerequisites maintained by nmake.

Sequence	Sequence atoms are made at specific times during nmake execution and are ignored if not specified. Sequence atom shell actions are always executed in the foreground. Assert intermediate rules (with actions) and append these to the sequence atom prerequisite list to avoid attribute or base and global rule clashes.

The Special Atoms are:

.ACCEPT [dynamic attribute]	Only file prerequisites can make .ACCEPT atoms out-of-date.
.ACCEPT [immediate rule]	The prerequisite atoms are accepted as being up to date. However, subsequent file prerequisites may make the atoms out of date.
.ACTIVE [readonly attribute]	Marks each target that is being made, i.e., currently active. The attribute is assigned to the target for the duration it takes to make it and its prerequisites.

.ACTIONWRAP [action rule]	If defined, the .ACTIONWRAP rule action is expanded in place of each shell action prior to command execution. The expansion takes place in the original rule context so that automatic variables refer to the original rule values. In particular, $(@) expands to the action for the original rule. For example,
	`.ACTIONWRAP :`
	`        echo making target $(<)`
	`         $(@)`
	causes each triggered shell action block to expand to:
	`echo making target <original target name>`
	`<expanded original action>`
	with the result that a message including the original rule target name would be output as the first step in each triggered shell action.

.AFTER [dynamic attribute]	.AFTER prerequisites are made after the action for the target atom has completed.
.ALARM [immediate rule]	Sets an alarm signal in seconds during nmake execution.
.ALWAYS [dynamic attribute]	The noexec option inhibits the execution of shell actions. However, shell actions for .ALWAYS targets are executed even with noexec on.

.APPEND *.pattern* [pattern association]	The prerequisites of .APPEND*".pattern"* are appended to the prerequisite list of each target atom which matches *pattern* immediately before the target prerequisites are made. Notice that *pattern* must be %.ARCHIVE for .ARCHIVE targets and %.COMMAND for .COMMAND targets.
.ARCHIVE [dynamic attribute]	Target atoms with the .ARCHIVE attribute are treated as archives **(**see ar(1)). Binding a .ARCHIVE target atom also binds the archive members to the target. .ARCHIVE may be used as a pseudo-suffix for .APPEND and .INSERT.
.ARGS [dynamic list]	The prerequisites of .ARGS are the command line target arguments. If, after making the .INIT sequence atom .ARGS has no prerequisites then the prerequisites of .MAIN are copied to .ARGS. As each prerequisite of .ARGS **i**s made it is removed from the .ARGS prerequisite list
.ATTRIBUTE [dynamic attribute]	Marks the target as a named attribute. Named attributes may be tested using the A**=***attribute***:** edit operator. A maximum of 32 named attributes may be defined. The prerequisites of .ATTRIBUTE constitute the list of all named attributes.
.ATTRIBUTE *.pattern* [pattern association]	When an atom which matches .pattern is bound it inherits the attributes of .ATTRIBUTE*.pattern*.
.BEFORE [dynamic attribute]	.BEFORE prerequisites are made just before the action for the target atom is executed.
.BIND [immediate rule]	The prerequisite atoms are bound.
.BIND ***.****pattern* [pattern association]	Specifies binding rules to be used when an atom cannot be bound using the normal rules. The return value is new binding.
.BOUND [readonly attribute]	Marks bound atoms.
.BUILT [readonly attribute]	Marks state rule atoms corresponding to atoms that have been built.
.CLEAR [assertion attribute]	Clears the attributes, prerequisites and action for the targets in the assertion.
.COMMAND [dynamic attribute]	Marks target atoms that bind to executable command files. .COMMAND may be used as a pseudo-suffix for .APPEND and .INSERT.

.COMPDONE [sequence]	This atom is made right after the makefile is compiled. It is skipped if the compiled makefile, *.mo, is not changed.
.COMPINIT [sequence]	This atom is made when the input makefiles are (re)compiled, just before the make object file is written.
.DONE [sequence]	This is the last atom made before nmake terminates execution.
.DONTCARE [dynamic attribute]	If a .DONTCARE target cannot be made, nmake continues as if it existed; otherwise an error is issued and nmake either discontinues work on the target parent and siblings if keepgoing is on, or it terminates processing and exits.
.ENTRIES [readonly attribute]	Marks scanned directory or archive atoms that have entries (members).
.ERROR [sequence]	This atom is executed when an error is encountered and the compile option is off. If the .ERROR make action block returns 0 (default), control returns to the point after the error. If the action block returns –1, then the error processing continues.
.EXISTS [readonly attribute]	Marks atoms that have been successfully made.
.EXPORT [dynamic list]	The prerequisites are treated as *variable* names to be included in $(=) automatic variable expansions.
.FAILED [readonly attribute]	Marks atoms that have been unsuccessfully made.
.FILE [readonly attribute]	Marks atoms bound to existing files.
.FORCE [dynamic attribute]	An atom with this attribute is always out-of-date the first time it is made during a single nmake execution.
.FOREGROUND [dynamic attribute]	The target action blocks until all other actions have completed. Normally nmake makes future prerequisites while concurrent actions are being executed, however, a .FOREGROUND target causes nmake to block until the corresponding action completes.

.FUNCTIONAL [dynamic attribute]	A functional atom is associated with a variable by the same name. Each time the variable is expanded the corresponding atom is made before the variable value is determined. For example,
	
	`src : .MAKE .FUNCTIONAL`
	`        return $(*.SOURCE:L<=*.c)`
	In this example, $(src) evaluates to all the .c files in the directories specified as prerequisites of .SOURCE (listed in the ascending order). Also, .FUNCTIONAL Special Atom provides argument support. For example,
	
	`V : .MAKE .FUNCTIONAL`
	`        return $(%:T=F)`
	one can call it by $(V a1 ... an), where a1 ... an is the list of arguments that can be referenced by $(%).

.GLOBALFILES [readonly list]	The prerequisites are the list of global makefiles specified by the global option.

.IGNORE [dynamic attribute]	Prevents any parent targets from becoming out-of-date with the target, even if the target has just been built. This allows initialization sequences to be specified for individual atoms:
	
	`main : init header`
	`    echo "executed if header is newer than main"`
	`init : .IGNORE`
	`    echo "always executed for main"`

.IMMEDIATE [dynamic attribute]	An atom with this attribute is made immediately after each assertion of the atom.
.IMMEDIATE [immediate rule]	The prerequisites and action are made each time this atom is asserted.
.IMPLICIT [dynamic attribute]	This target attribute causes the implicit metarules to be applied even if there is no action but prerequisites have been specified for the target. Otherwise the implicit metarules are only applied to targets with no explicit actions and prerequisites. This attribute turns the .TERMINAL attribute off at assertion time.
.INIT [sequence]	Made after the .MAKEINIT sequence atom.
.INSERT [assertion attribute]	Causes the prerequisites to be inserted before rather than appended to the target prerequisite list.

.INSERT *.pattern* [pattern association]	The prerequisites of .INSERT.*pattern* are inserted onto the prerequisite list of each target atom which matches *.pattern* immediately before the target prerequisites are made. Notice that *.pattern* must be .ARCHIVE for .ARCHIVE targets and .COMMAND for .COMMAND targets.
.INTERNAL [readonly list]	This atom is used internally and appears here for completeness.
.INTERRUPT [sequence]	This atom is made when an interrupt signal is caught. The engine state may become corrupted by actions triggered while .INTERRUPT is active. If the interrupt occurs while .QUERY is being made and set nointerrupt is executed by .INTERRUPT then after .INTERRUPT is made nmake continues from the point where the interrupt occurred, otherwise nmake exits with non-zero status.
.INTERRUPT.<*signal*> [signal]	This atom controls actions for the specified *signal*.
.JOINT [dynamic attribute]	The action causes all targets on the left hand side to be jointly built with respect to the prerequisites on the right.
.LOCAL [dynamic attribute]	An atom with this attribute is made on the local machine after each assertion of the atom when the coshell is active, no-op otherwise (see the coshell manual page). For .MAKE assertions, causes assignments in the action to be placed in the scope of the parent rule.
.MAIN [dynamic list]	If, after the .INIT target has been made, .ARGS has no prerequisites then the prerequisites of .MAIN are appended onto .ARGS. If not explicitly asserted in the input makefiles then the first prerequisite of .MAIN is set to be the first target in the input makefile that is not marked as .FUNCTIONAL, .OPERATOR, or .REPEAT, and is neither a state variable nor metarule.
.MAKE [dynamic attribute]	Causes the target action to be read by nmake rather than executed by the shell. Such actions are always read, even with noexec on.
.MAKE [immediate rule]	The prerequisites and action are made each time this atom is asserted.
.MAKEDONE [sequence]	Made after the .DONE sequence atom.
.MAKEFILES [readonly list]	The prerequisites are the list of makefiles specified by the file option.

.MAKEINIT [sequence]	This target is made just after the statefile has been loaded. The base rules typically use this atom to initialize the nmake engine. For this reason the user should not redefine the .MAKEINIT action or attributes. However, it is safe to insert or append prerequisites onto .MAKEINIT.
.MAKING [readonly attribute]	Marks each atom whose action is executing.
.MEMBER [readonly attribute]	Marks an atom that is a member of a bound archive.
.METARULE [readonly list]	The ordered list of LHS metarule patterns for all asserted metarules excluding the **%** match-all metarules. This list determines the metarule application order.
.MULTIPLE [dynamic attribute]	Normally each assertion removes duplicate prerequisites from the end of the target atom prerequisites list. .MULTIPLE atoms are allowed to appear more than once in a prerequisite list.
.NOTYET [readonly attribute]	Marks atoms that have not been made.
.NULL [assertion attribute]	Assigns the null action to the target atoms. Used when an action must be present, usually to force source files with prerequisites to be accepted.
.OPERATOR [dynamic attribute]	This attribute marks the target as an *operator* to be applied when reading makefiles. Assertion operator names must match either **::** or **:***identifier***:**
.OPTIONS [readonly list]	The prerequisites are the options declared by the option option.
.PARAMETER [dynamic attribute]	State variables with the .PARAMETER attribute are not expanded by the T=D and T=E edit operators.
.QUERY [immediate rule]	Full atom and state information is listed for each prerequisite.

.QUERY [action rule]	.QUERY *[action rule]*
	Making .QUERY places nmake in an interactive loop. Input entered at the make> prompt may be any valid makefile input. However, parsing readahead requires that a blank line follow an interactive assertion before it takes effect. If a list of atoms is entered without an assertion or assignment operator then the atoms are listed as if they were prerequisites of the .QUERY *immediate rule*. The interactive loop is exited by entering **control-D** (or **quit**) from the keyboard. The assertion
	.INTERRUPT : .QUERY
	causes the interactive loop to be entered on interrupts. The interactive loop can also be invoked by specifying the
	.QUERY Special Atom on the command line. The following command line will invoke the interactive loop:
	`$ nmake -f nmake_filename .QUERY`
	
	or, more commonly,
	`$ nmake -f nmake_filename query`

.READ [dynamic attribute]	The standard output of the action is read and interpreted as nmake statements.
.REBIND [immediate rule]	Each prerequisite is unbound and bound again as if it had already been made.
.REGULAR [readonly attribute]	Marks atoms that are bound to regular files
.REPEAT [dynamic attribute]	By default an atom is made at most once per nmake invocation. .REPEAT marks atoms that are to be made repeatedly. .FORCE is required to force the action to be triggered each time the atom is made.
.REQUIRE.*pattern* [pattern association]	Modifies the binding algorithm to allow a bound atom to map to a list of bound atoms. For example: .REQUIRE.–lcs : .FUNCTION return –lcs –lnsl –lsocket will return the entire list specified above every time –lcs is called: –lcs —> –lcs –lnsl –lsocket
.RETAIN [immediate rule]	The prerequisites are variable names whose values are to be retained in the statefile.

.SCAN [dynamic attribute]	Used for defining scanning rules for a project specific language that are not defined in the default base rules.
.SCAN .*x* [dynamic attribute]	Marks an atom (when bound to a file) to be scanned for implicit prerequisites using the *.x* scan strategy.
.SCAN .*c* [dynamic attribute]	Use C language scan strategies.
.SCAN .*f* [dynamic attribute]	Use FORTRAN scan strategies.
.SCAN .*F* [dynamic attribute]	Use SQL scan strategies.
.SCAN .*m4* [dynamic attribute]	Use m4 scan strategies.
.SCAN .*nroff* [dynamic attribute]	Use nroff/troff scan strategies.
.SCAN .*r* [dynamic attribute]	Use RATFOR scan strategies.
.SCAN .*sh* [dynamic attribute]	Scan for state variable references.
.SCAN .*sql* [dynamic attribute]	Use SQL scan strategies.
.SCAN.IGNORE [dynamic attribute]	Inhibits scanning.
.SCAN.NULL [dynamic attribute]	Inhibits scanning. Used to override .ATTRIBUTE.*pattern* scan strategies.
.SCAN.STATE [dynamic attribute]	Marks candidate state variables for scanning.
.SCANNED [readonly attribute]	Marks archive and directory atoms that have been scanned for members (implicit prerequisites) and records this information in the *statefile*. The .SCANNED attribute may be removed from the *statefile* with –SCANNED assertion, thus causing nmake to rescan the file being scanned at a later time. For example, t :: t.c t.c : -SCANNED t.c file is being scanned by nmake using .SCAN**.*c*** scan strategy and it is marked with .SCANNED attribute. t.c : -SCANNED nullifies this action.

.SEMAPHORE [dynamic attribute]	Limits the number of executing shell actions for target atoms having the same .SEMAPHORE prerequisite**.** Each .SEMAPHORE within an assertion increments the semaphore count by 1. The maximum semaphore count is 7. The following example makes the shell actions for a and b mutually exclusive. set jobs=10 all : a b .sema : .SEMAPHORE a b : .sema action
.SOURCE [dynamic list]	The prerequisites of .SOURCE are directories to be scanned when searching for files. The (left to right) directory order is important; the first directory containing the file is used. The **.** directory is always searched first.
.SOURCE *.pattern* [pattern association]	The prerequisites of .SOURCE*.pattern* are directories to be scanned when searching for files which match suffix *.pattern*. If the file is not found then the directories specified by the prerequisites of .SOURCE are checked. The (left to right) directory order is important; the first directory containing the file is used. The directory **.** is always searched first. The implicit dependency scan strategy (see .SCAN) may augment or override the default .SOURCE search for individual atoms.
.SPECIAL [assertion attribute]	Target atoms in the assertion are not appended to the prerequisites of .MAIN and multiple action diagnostics are inhibited.
.STATE [dynamic attribute]	Non-.STATEVAR atoms with this attribute are treated as state variables with no implied connection to an nmake variable.
.STATE [immediate rule]	The prerequisites are variable names that are marked as candidate implicit state variable prerequisites (see the *Variables* section).
.STATERULE [readonly attribute]	Marks internal state rule atoms that are used to store state information.
.STATEVAR [readonly attribute]	Marks state variable atoms.
.SYNC [immediate rule]	Synchronizes the statefile.
.TARGET [readonly attribute]	Marks atoms that appeared as the target of an assertion.

.TERMINAL [dynamic attribute]	This attribute allows only .TERMINAL metarules to be applied to the target. Otherwise metarules are applied to targets with no explicit actions and prerequisites. This attribute turns the .IMPLICIT attribute off at assertion time. For metarule assertions, .TERMINAL marks **unconstrained** metarules that may be applied only to targets or non-generated source files. .TERMINAL attribute may also be applied to directories that do not have subdirectories. If a .SOURCE directory has .TERMINAL attribute, nmake will not search that directory for subdirectories. This can be used as an optimization technique when there are many .SOURCE directories that do not have subdirectories and there are many files with the directory prefixes (saves on unnecessary filename look-ups).
.TMPLIST [readonly list]	This atom is used internally and appears here for completeness.
.TRIGGERED [readonly attribute]	Marks atoms whose actions have triggered during the current nmake execution.
.UNBIND [immediate rule]	Each prerequisite is unbound as if it had not been bound.
.USE [dynamic attribute]	Marks the target as a .USE atom. Any target having a .USE atom as a prerequisite will be built using the .USE atom action and attributes. The leftmost .USE prerequisite takes precedence.
.VIEW [readonly list]	The prerequisites are view directory pathnames. **.** is by default the first view directory. A view directory is a directory from which nmake could be executed. .VIEW is initialized from the MAKEPATH and VPATH environment variables
.VIRTUAL [dynamic attribute]	A .VIRTUAL atom never binds to a file. .VIRTUAL atom times are saved in the statefile.
– [dynamic list]	Used to control synchronization when making a list of prerequisites. A – in a prerequisite list causes nmake to wait until the preceding prerequisite's actions are complete before continuing.

Command Execution

Each shell action is sent as a unit to sh via coshell(3). sh echoes commands within actions as they are executed unless the silent option is on. Since actions are sent as a unit, special shell constructs (**case**, **if**, **for**, **while**) may cross **newline** boundaries without **newline** escapes.

Commands within actions returning nonzero status (see intro(1)) cause nmake to stop unless the ignore or keepgoing option is on.

nmake works only with Bourne-based shells such as sh(1) and ksh(1). nmake is optimized to work with ksh(1). If set, the COSHELL environment variable must point to ksh, sh, or the network shell server coshell. ksh will be used by default.

Special Commands

ignore *shell-command*	Causes the exit status of *shell-command* to be ignored.
silent *shell-command*	Prevents *shell-command* from being printed by the shell, if possible. silent must precede ignore if both are to be used. set +x prevents subsequent commands from being printed by the shell up to and including the next set –x.

Jobs

The jobs option allows nmake to build many targets concurrently. The builds are synchronized using the target dependency graph. Actions for targets with the .FOREGROUND attribute block until all other jobs have completed. Prerequisites with the .SEMAPHORE attribute are used for mutual exclusion.

All actions should be written with concurrency in mind. Most problems occur when commands generate files with fixed names, such as yacc(1) and lex(1).

Conventions

nmake attributes match the regular expression .[.A-Z][.A-Z0-9]*. User attributes match the regular expression .[.A-Z][.a-z0-9]*. Intermediate targets match the regular expression .[.a-z][.a-z0-9]*. Use $COTEMP (see the *Environment Variables* section) to generate temporary file names. To avoid conflicts with the base rules the user should not define upper case atom or variable names with **.** as the first character.

Options

Options are preceded by **–** or **+** and may appear anywhere on the command line. **+** turns the options off. As with assignments, command line options override makefile option settings. The files Makeargs and makeargs (current directory only; no viewpathing) are checked in order for additional options. Only one file is read; each line is interpreted as a single argument and is inserted before the first argument of the original command line. # as the first character in a line denotes a comment, otherwise all lines (including blank lines) are significant.

All options have a string name and can be set using the **–o[no]***name***[=***value***]** option-by-name form. The popular options also have a corresponding flag letter; in some cases using the flag letter is the inverse of using the option name. The default values are **off** or **0** unless otherwise noted. *[inverted]* denotes that the flag letter has the opposite meaning of the string name (i.e., **-n** is noexec). The option names and corresponding flags (if any) are:

accept	**–A**	Accept any existing targets that are newer than their corresponding file prerequisites as being up-to-date.
base	**–b**	Must be set along with compile in order to compile base or global make-files.
believe	**-B** **-o believe**=*level*	Believe statefile information for views >= *level*.
byname	[**no**]*name*=[*value*] … **–o** [**no**]*name*=[*value*] ...	Set option *name* by name.
compatibility	**–C**	Disable compatibility warning mes-sages
compile	**–c**	Force the makefiles to be compiled into a single nmake object file. nmake exits after the files are compiled. To compile a global makefile **–b –f** *makefile* must be specified.
debug=*level*	**–d** *level*	Provide a debug trace of nmake's actions. The number argument *level* selects the debug level; higher levels produce more output. Levels 4 and higher are only enabled in debugging versions of nmake (currently all distrib-uted versions). To avoid unnecessary noise during the engine bootstrap: if *level* is less than 20 then all levels are disabled during the early bootstrap and if *level* is 1, then debug *level* 1 is not turned on until after the .INIT sequence atom has been made.
errorid=*id*	**–E** *id*	*id* is appended to the error and debug trace identification. Useful for tracing recursive nmake actions.

exec	**–n***[inverted]*	noexec causes shell actions to be traced and printed but not executed and also inhibits makefile and statefile compilations. Shell actions for targets with the .ALWAYS attribute, however, are executed even with noexec.
expandinclude	**-p**	Causes files referenced by the nmake 'include' statement to be textually expanded into the including file. Output written to a .mi file
expandview	**–x**	Used with 3D File System on main machine and coshell (see the coshell manual page) to access non-3D machines. The expandview option forces nmake to generate full path names for viewpathing.
explain	**–e**	Prints an explanation for actions taken.
file=*makefile*	**–f** *makefile*	Reads the descriptions in *makefile*. If *makefile* is **–**, then the default makefile names are not checked. More than one file option may appear; the files are read in order from left to right.
force	**–F**	Force all active targets to be out-of-date.
global=*makefile*	**–g** *makefile*	Similar to the file option, except that *makefile* is treated as a global makefile. This means that the default makefiles will still be checked if no file options appear.
ignore	**–i**	Ignore error exit codes from shell action commands.
ignorelock	**–K**	Override locking protocols. Normally, only one nmake is allowed to execute on a given makefile.
include=*directory*	**–I** *directory*	Passed to cpp for makefile preprocessing.
interrupt		See .INTERRUPT above.

jobs=*level*	**–j** *level*	Specifies that nmake may execute up to *level* target actions concurrently (see also NPROC environment variable). *level*=0 causes nmake to block for the completion of each action before proceeding. The default *level*=1 allows nmake to determine the next out-of-date target while the previous action is executing.
keepgoing	**–k**	If the shell action for the current target returns error exit status, continue working on sibling targets that do not depend on the current target.
list	**–l**	List variable and rule definitions after the makefiles are read and exit after the listing. The targets are neither checked nor updated.
mam	**M[static|dynamic| regress][:*file*[:[*label*] [:*root*]]][,noport] [,dontcare]**	This option generates the make abstract machine language equivalent of the current makefile and writes it either to the standard output or a user-specified file. With the **-M** option, static is for generating mamfiles, dynamic is for tracing nmake execution steps, and regress is for canonicalizing regression testing on different hosts. If using neither dynamic nor label, file implies that output appends to file, otherwise it implies overwrite. label must be numeric (usually a process ID), and root specifies that all files listed are relative to this root directory. noport inhibits porting hints, while dontcare lists targets that contain the .DONTCARE Special Atom.
mismatch	**–X**	If the statefile version does not match the nmake version then run with readstate and accept on.
never	**-N**, **-n**	Never execute. **-N** overrides .ALWAYS

option=*flag,name* [*type,function*]		Define a new option *name*. *flag* is the command-line option letter; *name* is the command-line option name. [*type*] overrides the default Boolean option type. *type* may be **b** *name* is a Boolean option **n** *name* is a numeric option **s** *name* is a string-valued option. **x** *name* is not to be expanded in **$(–)**. **x** type may be combined with any of the other types. The optional *function* is the name of a function in the makefile that acts based on the value of *name*.
override		Override explicit actions by applying matching metarule actions if possible. Also inhibits statefile updates.
preprocess	**–P**	Force makefiles to be preprocessed (deprecated).
readstate=*[level]*	**–S** level	Ignore the previous state by not reading any statefiles. The number argument *level* specifies which statefiles are read from the viewpath. The statefiles in all the viewpath levels up to and including *level* are read. The rest are ignored. If *level* is omitted, then all statefiles are ignored.
reread=*level*		Force the input makefiles to be read rather than loading the corresponding nmake object files. This option is generated automatically when nmake object file inconsistencies are detected. *Level* is used as an internal recursion check (nmake automatically re-execs itself with the reread option when it determines that the makefile needs to be recompiled)

ruledump	**–r**	List the detailed status of each rule after all targets have been made. If list is also set then the listing occurs before any targets are made and nmake exits immediately after the listing.
scan noscan		Inhibits the implicit file dependency scan algorithms controlled by the .SCAN and .SCAN.*suffix* attributes. scan is the default.
silent	**–s**	Execute but do not print shell actions.
strictview	**–V** *[inverted]*	This option is on by default when viewpathing is used. nostrictview enables non-strict viewpathing. The difference between strictview and nostrictview is in the order in which the .SOURCE[.pattern] directories are searched. For example: If *viewpath* is "v0:v1" and .SOURCE : A B, the strictview interpretation is: v0 v1 v0/A v1/A v0/B v1/B The non-strictview interpretation is: v0 v0/A v0/B v1 v1/A v1/B
targetcontext		Expands actions in the target directory when the target name contains a directory prefix.
test=*bits*	**–T** *bits*	Enable the internal debug tests specified by the bit vector *bits*. Some tests enable verbose tracing while others change the internal algorithms. This option should be avoided unless the user knows what the tests do.
tolerance=*seconds*	**–z** *seconds*	Times that differ by less than *seconds* seconds are considered equal. This option may be necessary for certain network file system implementations that fail to maintain clock integrity between systems on the network.
touch	**–t**	Touch the modify date of targets, bypassing the actions that build them. Only existing targets are touched.

vardump	**–v**	List variable assignments. Useful in conjunction with the debug option.
warn	**–w**	Enable more detailed source file warning messages.
writestate		nowritestate prevents the state file from being updated on exit.

Environment

The file names and directories used by the nmake engine are parameterized using variables.

Engine Variables

These variables are assigned default values by the nmake engine itself. Some of the values are determined at installation time while others are determined by each invocation environment.

COTEMP	The COTEMP environment variable is generated and set to a different value for each shell command. It is 10 characters long and can be used for temporary file names, so names of the form <pfx>$COTEMP<sfx> will be up to 14 characters to satisfy the system V file name length restriction.
MAKE	The path name of the current nmake program suitable for use as a shell command or in an execvp(3) system call.
MAKEARGS=*Makeargs***:***makeargs*	A colon-separated list of the candidate argument file names.
MAKEFILE	The name of the first makefile.
MAKEFILES=*Makefile***:***makefile*	A colon separated list of candidate implicit makefile names.
MAKELEVEL	Current makefile recursion level.
MAKELIB	The directory for nmake related files, e.g., base rules.
MAKEPP=*$(MAKELIB)/../cpp*	The name of the (deprecated) makefile preprocessor.

MAKERULES=*makerules*	The name of the default base rules. The base rules file must be a compiled make object file and is named by changing the suffix in $(MAKERULES) to .mo and binding the result using the directories in .SOURCE.mk. *makerules* may be an absolute path name.
MAKERULESPATH=*$(LOCALRULESPATH):$(MAKELIB):/usr/local/lib/make:/usr/lib/make*	Used to initialize .SOURCE.mk. All makefiles, including compiled make object files and files included by makefiles, are bound using the directories in .SOURCE.mk.
MAKEVERSION	The nmake engine version stamp.
OLDMAKE	Executed via execvp(3) when the first input makefile is not a valid nmake makefile.
PWD	The absolute pathname of the current directory.
VOFFSET	Set to the path that goes from the viewpath node root to the current working directory.
VROOT	Set to the relative path that gets from the current working directory to the viewpath node root.

Uninitialized Variables

These variables have **null** default values and are used only when defined.

COSHELL	The name of the shell used to execute shell actions. execvp(3) is used to execute the shell. If COSHELL is not defined, then ksh, sh, and /bin/sh are tried in order.
EXTRASTATE	Used by the :cc: operator to refer to the name of the makefile whose statefile information is needed to complement the statefile information of the primary statefile, null.ms
MAKECONVERT	A space separated pair of the build tool script (other than nmake) and the conversion tool. Used for compile time makefile conversion. For example: `export MAKECONVERT='Make.src "nmakegen $(>)"'`
MAKEEXPORT	A colon separated list of variable names, whose values are set in the nmake environment (by the user) and exported in the shell coprocess environment (by the nmake engine). They can be referenced in the shell action block as a shell variable.

MAKEIMPORT	A colon separated list of environment variable names, the values of which override any makefile variable assignments.
MAKEPATH	A colon separated list of directory names from which nmake could be run on the current makefile. These directories are used by the initialization script to initialize the .VIEW Special Atom. **.** is always the first view by default.
MAKE_OPTIONS	This variable sets the makerules variables. Multiple variables can be set. For example: MAKE_OPTIONS="ancestor=2 cctype=/bin/cc"
NPROC	The maximum number of coshell processes that are to be executed simultaneously (the environment variable equivalent of -j).
PROBEPATH	A colon-separated list of alternate probe hierarchy root paths. When used in conjunction with the localprobe base rules variable, this variable can be used to determine the search path order for local probe information.
UMASKCHANGE	May be 0/NO/no to supress changing the umask to match the current directory permissions, or 1/YES/yes/null to change the umask. For compatibility the umask change is enabled by default. Any other value generates a warning and keeps the umask change enabled.
VERSION_ENVIRONMENT	A colon-separated list of environment variables for probe to consider when generating the filename hash for the probe configuration file.
VPATH	A colon separated list of viewpath node names. The node names are converted to MAKEPATH directories that are used by the initialization script to initialize the .VIEW Special Atom. VPATH is ignored if nmake is not executing within the first viewpath node.

Files

Makefile, makefile	default makefiles, tried in order
Makeargs, makeargs	default argument files, tried in order
$MAKELIB/Makeinit.mk	engine initialization script
$MAKELIB/makerules.mo	default compiled base rules
base.ml	make lock file
base.mo	make object file
base.ms	make statefile

Diagnostics

nmake diagnostic messages are preceded by make:. If a diagnostic is produced while reading a makefile (or executing a .MAKE action) then the makefile (or atom) name and line number are included. Debug trace messages are preceded by debug-*level***:** where *level* is the message trace level. *panic* diagnostics include the nmake version, source file and source line number where the error was detected. There is no graceful recovery from a *panic* (i.e., the statefile usually gets trashed); mail the panic message with a brief description of the context to the nmake provider.

See Also

S. I. Feldman, *Make – A Program for Maintaining Computer Programs*
E. G. Bradford, *An Augmented Version of Make*
V. B. Erickson, J. F. Pellegrin, *Build – A Software Construction Tool*
G. S. Fowler, *The Fourth Generation Make*

Related Commands

3d(1), ar(1), cc(1), coshell, cpp, lex(1), libcoshell(3), make(1), probe, sh(1), xargs(1), yacc(1)

Caveats

Any variables that are expanded while reading makefiles are *frozen* into the corresponding nmake object file. If the value of a *frozen* variable changes from one invocation of nmake to the next, either in a command line definition or in the environment, then a warning is issued and the makefile is automatically recompiled. Most frozen variable expansions can be deferred by entering $(*var*) as $$(*var*).

Some commands return nonzero status inappropriately. Use ignore *command* to overcome any difficulties.

nmake only detects source files that it builds or those that exist before nmake is executed.

nmakelog Command

nmakelog – run nmake producing serialized build log

Synopsis

nmakelog [nmake-arguments**]**

Description

nmakelog runs nmake (1) configured to produce a serialized build log. nmakelog addresses a problem that occurs when more than one shell action executes concurrently under the control of nmake (1), (e.g. when the value of the **-j** option is greater than 1, or when coshell (1) is used). In this case, each line of output from multiple concurrently executing commands appears in the output log in the order that it is written, with the result that output of different commands is intermingled, making it very difficult to scan and interpret.

nmakelog serializes the output of nmake (1): it writes a log in which the output of each command appears contiguously and in the correct order in the log file; additionally, at higher levels the output of each recursive make (which may itself be running concurrently with other shell actions and recursive makes, and itself be subordinate to higher level recursive makes) appears contiguously and correctly ordered in the output file.

The nmakelog command line is identical to the nmake (1) command line; use nmakelog as you would nmake (1). As usual, during the build the unserialized output appears on standard output as it is generated; in addition, serialized output appends to file $makelog, by default makelog.

nmakelog may be used with distributed (coshell (1) based) or non-distributed builds.

Example

```
nmakelog -j4 -f a.mk
```

Files

logfilter	executable used to reorder log file
taglines	executable used to insert tags on build log lines

Related Commands

nmake(1), makerules(1), coshell(1)

mam Translator Commands

mamdep, mamdag, mamdot, mamdt – translate MAM ouput to specified formats

Synopsis

mamdep | mamdag | mamdot | mamdt [-a "shell_pattern" -t "shell_pattern"**]**

Description

mamdag, mamdot and mamdt translate nmake MAM output to formats compatible with dag(1), dot(1), and dt(1), respectively. mamdep translates MAM output to a simple textual format indicating both the depth and the name of each atom. All MAM output translators take a mamfile as input, and output the supplied data in an acceptable format for each tool. In addition to basic conversion, all translators also support the capability to "ignore" specific portions of the build hierarchy based on command-line options, which allow an arbitrary ksh(1) pattern.

The **-a** option indicates that atoms matching the given shell pattern, along with the direct descendents of each matched atom, should be ignored. The **-t** option indicates that all atoms with attributes matching the given shell pattern, along with the direct descendents of each matched atom, should be ignored. For example,

```
mamdep -a"*.m[os]" -t"implicit*" < mamoutput
```

processes the MAM output file mamoutput, ignoring all atoms ending in .mo or .ms, along with atoms having the "implicit" attribute, and prints the results on standard output. If an atom is ignored based on a user-specified pattern, the entire subtree based on that atom is also ignored.

MAM output files can be generated with nmake's **-M** command line option. For use with the mam translators, dynamic mode mam output will list the set of atom dependencies associated with a given build.

Example

The following example illustrates the usage of a mam translator command.

```
$ nmake -x -Mdynamic:mamfile::/
```

```
+ ppcc -i /tools/nmake/nmake3.1.2/lib/cpp cc -O -I-D/tools/nmake/nmake3.1.2/lib/probe/
C/pp/BFE0A4FEobincc -I- -c hello.c
```

```
+ ppcc -i /tools/nmake/nmake3.1.2/lib/cpp cc -O -o hello hello.o
```

The above call to nmake uses dynamic mam mode to generate mam output to file *mamfile*. The '/' specification in the mam option specifies that files listed are relative to the root directory, implying full directory listing. This specification can be combined with the -x (expandview) option to generate full directory paths in the mam output.

```
$ mamdag <mamfile >mamdag.out
```

The above call translates the mam output file to a format suitable for use with dag(1).

pax Command

pax – portable archive interchange

Synopsis

pax [–r] [–v] [–f *archive* **] [–s** */old/new/***[glpsu]] [–z** *base* **]**
[*pattern* **...]**
pax –w [–a] [–v] [–f *archive* **] [–s** */old/new/***[glpsu]]**
[–x *format* **] [–z** *base* **] [** *pathname* **...]**
pax –rw [–v] [–s */old/new/***[glpsu]] [** *pathname* **...]** *directory*

Description

pax reads and writes archive files in various formats. There are four operation modes controlled by combinations of the **–r** and **–w** options.

pax **–w** writes the files and directories named by the *pathname* arguments to the standard output together with pathname and status information. A directory *pathname* argument refers to the files and (recursively) subdirectories of that directory. If no *pathname* arguments are given then the standard input is read to get a list of pathnames to copy, one pathname per line. In this case only those pathnames appearing on the standard input are copied.

pax **–r** reads files from the standard input that is assumed to be the result of a previous pax **–w** command. Only files with names that match any of the *pattern* arguments are selected. A *pattern* is given in the name-generating notation of *sh*(1), except that the ***/*** character is also matched. The default if no *pattern* is given is *****, which selects all files. The selected files are conditionally created and copied relative to the current directory tree, subject to the options described below. By default the owner and group of selected files will be that of the current user, and the permissions and modify times will be the same as those in the archive. If the **–r** option is omitted then a table of contents of the selected files is listed on the standard output.

pax **–rw** reads the files and directories named in the *pathname* arguments and copies them to the destination *directory*. A directory *pathname* argument refers to the files and (recursively) subdirectories of that directory. If no *pathname* arguments are given then the standard input is read to get a list of pathnames to copy, one pathname per line. In this case only those pathnames appearing on the standard input are copied. *directory* must exist before the copy.

The standard archive formats are automatically detected on input. The default output archive format is implementation defined, but may be overridden by the **–x** option described below. *pax* archives may be concatenated to combine multiple volumes on a single tape or file. This is accomplished by forcing any format prescribed pad data to be null bytes. Hard links are not maintained between volumes, and delta and base archives cannot be multi-volume.

A single archive may span many files/devices. The second and subsequent file names are prompted for on the terminal input. The response may be:

!*command*	Execute *command* via system(3) and prompt again for file name.
EOF	Exit without further processing.
CR	An empty input line retains the previous file name.
pathname	The file name for the next archive part.

Basic Options

a	For **–w**, append files to the end of the archive.	
f *archive*	*archive* is the pathname of the input or output archive, overriding the default standard input for **–r** and **–rw** or standard output for **–w**.	
s /*old*/*new*/[**glpu**]	File names and symbolic link text are mapped according to the ed(1) style substitution expression. Any non-null character may be used as a delimiter (**/** shown here). Multiple **–s** expressions may be specified; the expressions are applied from left to right. A trailing **l** converts the matched string to lower case. A trailing **p** causes successful mappings to be listed on the standard error. A trailing **s** stops the substitutions on the current name if the substitution changes the name. A trailing **u** converts the matched string to upper case. File names that substitute to the null string are ignored on both input and output. The **–o physical** option inhibits symbolic link text substitution.	
v	Produces a verbose table of contents listing on the standard output when both **–r** and **–w** are omitted. Otherwise the file names are listed on the standard error as they are encountered.	
x *format*	Specifies the output archive *format*. If specified with **–rw** then the standard input is treated as an archive that is converted to a *format* archive on the standard output. The input format, which must be one of the following, is automatically determined. The default output format, named by **–**, is currently **cpio**. The formats are:	
	ansi	ANSI standard label tape format. Only regular files with simple pathnames are archived. Valid only for blocked devices.
	asc	The **s5r4** extended cpio(5) character format.
	aschk	The **s5r4** extended cpio(5) character format with header checksum. This format is misnamed **crc** in the **s5r4** documentation.
	binary	The cpio(5) binary format with symbolic links. This format is obsolete and should not be used on output.
	cpio	The cpio(5) character format with symbolic links. This is the default output format.
	ibmar	EBCDIC standard label tape format. Only regular files with simple pathnames are archived. Valid only for tape devices.

x *format* (cont.)	**posix**	The IEEE 1003.1b-1990 interchange format, partially compatible with the X3.27 standard labeled tape format.
	portarch	The s5r2 portable object library format. Valid only on input.
	randarch	The BSD ranlib object library format. Valid only on input.
	tar	The tar(5) format with symbolic links.
	ustar	The POSIX IEEE Std 1003.1-1988 tar format.
	vdb	The *virtual database* format used by cia(1) and cql(1).
	vmsbackup	ANSI standard label VMS backup savset tape format. Valid only for input tape devices.
	format	Formats can be defined as extensions to the builtin formats. For any non-builtin format *format*, can be

Extended Options

All options have long string names specified using **–o** [no]*name*[=*value*];

append	(**–a**) For **–w**, append files to the end of the archive.
atime	Preserve the access time of all files.
base=*path*	(**–z**) **b***blocking* Set the output blocking size. If no suffix (or a **c** suffix) is specified then *blocking* is in 1 character units. A **b** suffix multiplies *blocking* by 512 (1 block), a **k** suffix multiplies *blocking* by 1024 (1 kilobyte) and an **m** suffix multiplies *blocking* by 1048576 (1 megabyte). *blocking* is automatically determined on input and is ignored for **–rw**. The default *blocking* is **10k** for block and character special archive files and implementation defined otherwise. The minimum *blocking* is **1c**.
exact (**–n**)	For **–r** the pattern arguments are treated as ordinary file names. Only the first occurrence of each of these files in the input archive is read. *pax* exits with zero exit status after all files in the list have been read. If one or more files in the list is not found, *pax* writes a message to standard error for each of these files and exits with a non-zero exit status. The file names are compared before any pathname transformations are applied.

m	File modification times are not retained.
o	Restore file ownership as specified in the archive. The current user must have appropriate privileges.

Compatibility Options

These options provide functional compatibility with the old cpio(1) and tar(1) commands.

c	Complement the match sense of the *pattern* arguments.
d	Intermediate directories not explicitly listed in the archive are not created.
i	Interactively *rename* files. A file is skipped if a null line is entered and *pax* exits if **EOF** is encountered.
l	For **–rw**, files are linked rather than copied when possible.
p	Preserve the access times of input files after they have been copied.
t *device*	*device* is an identifier that names the input or output archive device, overriding the default standard input for **–r** or standard output for **–w**. Tape devices may be specified as *drive*[*density rewind*] where *drive* is a drive number in the range [0–7], *density* is one of **l**, **m** and **h** for **low** (800 bpi), **medium** (1600 bpi – default) and **high** (6250 bpi) tape densities and *rewind* is **n** to inhibit rewinding of the tape device when it is closed. Other forms for *device* are implementation defined.
u	Copy each file only if it is newer than a pre-existing file with the same name. This option implies **–a**.
y	Interactively prompt for the disposition of each file. **EOF** or an input line starting with **q** causes *pax* to exit. Otherwise an input line starting with anything other than **y** causes the file to be ignored.

Extended Options

These options provide fine archive control, including delta archive operations.

e *filter*	Run the *filter* command on each file to be output. The current name of the file to be output is appended to the filter command string before the command is executed by the shell.
h	Inhibit archive heading and summmary information messages to stderr.
k	For **–r** continue processing the archive after encountering an error by attempting to locate the next valid entry. This is useful for archives stored on unreliable media.

z *base*	Specifies the delta base archive *base* that is assumed to be the result of a previous **pax –w** command. For **–w** the input files are compared with the files in *base* and file delta information is placed in the output archive using the delta algorithm. For **–r** the delta information in the input archive is used to update the output files with respect to the files in *base*. For **–rw** the delta information in the archive on the standard input is used to generate an archive on the standard output whose entries are updated with respect to the files in *base*. If *base* is **–** or an empty file then the input files are simply compressed. **–z -** must also be specified to produce a compressed archive for **–rw**.	
B *count*	Sets the maximum archive part output character count. *pax* prompts for the next archive part file name. Valid only with **–w**.	
C	Archive entries smaller than **–B** *maxblocks* must be contained within a single part. Valid only with **–B**.	
L	Copy a logical view of the input files. Symbolic links are followed, causing the pointed to files to be copied rather than the symbolic link information. This is the default.	
M *message*	Set the *end of medium* prompt to *message*. This message is used to prompt interactively for the next tape reel or cartridge in cases where the tape runs out before all files have been copied. *message* may contain one printf(3) style integer format specification that is replaced with the next part number.	
P	Copy a physical view of the input files. Causes symbolic link information to be copied as opposed to the default (logical view) action of following symbolic links and copying the pointed to files.	
R *option*[*value*][,*option*[*value*]...]	Set record oriented format options. Multiple options may be concatenated using **,**. Some options may be fixed for some formats. The options are:	
	c	Record data is subject to character set conversions.
	f *format*	Set the output record format to *format*. The supported record formats are:
		D Variable length with 4 byte record header. The record size default is 512.
		F Fixed length with no record header. The record size default is 128.
		S Spanned variable length with 4 byte record header. The record size default is 0 (no limit).

R *option*[*value*][,*op tion*[*value*]...] (cont.)	**f** *format* (cont.)	**U** Variable length with no record header. The output block size matches the size of each output record. The record size default is 512.
		V Spanned variable length with binary 4 byte record header. The record size default is 0 (no limit). The **D** format is preferred.
	m *pattern*	Only those files with input record format matching *pattern* are processed.
	p	Partial output blocks are padded to the full blocksize.
	s *size*	Set the output record size to *size*. *size* should divide the output blocking.
	v *label*	Set the output volume label to *label*. Some formats may truncate and/or case-convert *label*.
S	Similar to **–l** except that symbolic links are created.	
U *id*	Set file ownership to the default of the user named *id*. Valid only for the super-user.	
V	Output a '.' as each file is encountered. This overrides the **–v** option.	
X	Do not cross mount points when searching for files to output.	

Diagnostics

The number of files, blocks, and optionally the number of volumes and media parts are listed on the standard error. For **–v** the input archive formats are also listed on the standard error.

Examples

pax –w –t 1m .

Copies the contents of the current directory to tape drive 1, medium density.

mkdir *newdir*
cd *olddir*
pax–rw . *newdir*

Copies the *olddir* directory hierarchy to *newdir*.

Related Commands

ar(1), cpio(1), find(1), ksh(1), tar(1), tw(1), libdelta(3), cpio(5), tar(5)

Bugs

Special privileges may be required to copy special files.
Each archive format has a hard upper limit on member pathname sizes.
Device, user-id and group-id numbers larger than 65535 cause additional header records to be output. These records are ignored by old versions of cpio(1) and tar(1).

probe Command

probe – probe language processors for tool information

Synopsis

lib/probe/probe [–adgklsv] [-p <path>] *language tool processor*

Description

probe provides command support for the libast(3) routine pathprobe. It maintains *tool* specific information for *language* processors. lib/probe/probe controls the language and tool specific probe command lib/probe*/language/tool*/probe and performs consistency and security checks to ensure that the tool specific command generates valid information.

The information for the *language* processor *cmd* for *tool* resides in the file lib/probe*/language/tool/XXXXXXXXPPPPPP* where *XXXXXXXX* is a hex hash value for the fully qualified pathname for *cmd* and *PPPPPP* is the last six characters of the fully qualified pathname for *cmd* with */* deleted and left filled with . for short paths. lib/probe/probe **–k** *language tool cmd* lists this pathname on the standard output. The first line of the information file must end with the fully qualified pathname of *cmd*. This is used to detect pathname hash collisions.

The pathprobe library routine calls lib/probe/probe only when new information must be generated or the current information is older than the tool specific probe command or the valid probe_hints file. The out-of-date test compares the information file ctime with the probe command file or probe_hints script mtime. By default the information files have write permission disabled. To handle cases where the automatic probe information is incorrect or insufficient, lib/probe/probe will not overwrite an information file if owner write permission is enabled. The corrected probe information should be forwarded to the *tool* provider so that the tool specific probe command can be updated.

The options are provided for administrative purposes.

–a	List the probe attribute definitions. The attributes are passed as a string to the tool specific probe command.
–d	Delete the current probe information file. This is restricted to the owner of *pro-cessor* or the owner of lib/probe/probe.
–D	This option is for debugging. It shows the steps that probe used in generating the configuration information for the specified language processor.
–g	Generate the information file only if necessary.

–k	List the fully qualified information file pathname.
–l	List the information file contents.
-p <path>	Specify an alternate probe hierarchy path. This option allows custom probe directories to be used for both the lib/probe/language/tool/probe generation shell and the resulting *lib/probe/language/tool/XXXXXXXXPPPPPP* probe information file. The consistency checks between the generation shell and the information file match those of the normal case. In addition, the mtime of the generation shell is also verified against the mtime lib/probe/probe to limit version incompatibilities. If an executable generation shell is not found in *<path>/lib/probe/language/tool/probe* then the non-alternate probe generation shell is used. The set-uid permissions of *lib/probe/probe* are disabled in this mode, and the alternate directory *<path>/lib/probe/language/tool/* must be writable by the gid of the executing user. The resulting probe information file will reside in *<path>/lib/probe/language/tool/XXXXXXXXPPPPPP*.
–s	Suppress information messages.
–t	Test – generate the information on the standard output. The current information file is unaffected.
–v	Verify the current information file by checking the fully qualified language process pathname.

Examples

To list the C language processor cc(1) information for nmake:

```
lib/probe/probe -l C make cc
```

To modify the cpp information for gcc(1):

```
info=$(lib/probe/probe -k C pp gcc)
        chmod u+w $info
edit $info
```

Files

lib/probe/*language*/*tool*/	*tool* specific information directory for *language* processors

Caveats

lib/probe/probe must be set-uid to the owner of the lib/probe directory hierarchy. This allows probe information to be generated and shared among all users.

Related Commands

cc(1), cpp, nmake, libast(3), pp(3)

See Also

Appendix A of the *Alcatel-Lucent nmake Product Builder User's Guide*

proto Command

proto – make prototyped C source compatible with K&R, ANSI and C++

Synopsis

proto [–c *name=value* **] [–fhinprs]** *file* **...**

Description

proto converts ANSI C prototype constructs in *file* to constructs compatible with K&R C, ANSI C, and C++. Only files with the line #pragma prototyped in the first 64 lines are processed; other files are silently ignored. If **–f** is specified then all files are processed. If **–h** is specified then the *proto* inline header definitions are omitted. If **–n** is specified then the output will contain #line cpp directives to maintain line number compatibility with the original source file. If **–p** is specified then ignored file contents are passed untouched to the output. If **–s** is specified then #include <prototyped.h> is output instead of the inline header definitions. If **–r** is specified then the files are modified in place; otherwise the output is placed on the standard output.

–i specifies inverse proto: classic function definitions are converted to ANSI prototype form and non-ANSI directives are commented out. In this case files with the line #pragma noprototyped in the first 64 lines are silently ignored. If **–h** is specified with **–i** then only extern prototypes are emitted for each non-static function. This option requires that all classic function formal arguments be declared, i.e., omitted declarations for **int** formals will not generate the correct prototype. Typical usage would be to first run proto –ih *.c to generate the external prototypes that should be placed in a **.h** file shared by ***.c** and then proto –ir *.c to convert the function prototypes in place. Note that prototype code conditioned on __STDC__ will most likely confuse **–i**.

–c *name=value* generates a copyright notice comment at the top of the output controlled by one or more *name=value* pairs. If *value* contains space characters then it must be enclosed in either "..." *or '...'* quotes. *name* may be:

type	The copyright type. **proprietary** (default) and **nonexclusive** are supported.
corporation	They own it all, e.g., *Alcatel-Lucent*
company	Within the corporation, e.g., *Bell Laboratories*.
organization	Within the company, e.g., *Software Engineering Research Department*.

license	External reference to the detailed license text, e.g., http://www.bell-labs.com/project/nmake/licenseagreement.html.
contact	External reference for more information, e.g., nmake@alcatel-lucent.com.

The copyright notice text can be generated by:

```
proto -hfc "..." /dev/null | egrep -v '^$| : : .* : :'
```

All conversion is done before macro expansion, so programs that disguise C syntax in macros lose. A single prototype must not span more than 1024 bytes.

ANSI <stdarg.h> variable argument constructs are recognized. va_start(must appear within 1024 bytes of the enclosing function prototype.

The ANSI macro operators # (stringize) and ## (concatenate) are recognized, but note that old K&R preprocessors cannot be coaxed to expand arguments to # or ##.

For C++, extern and references are placed in the "C" linkage class. Although C allows extern in function bodies, C++ restricts linkage class specifications to file scope.

The macros __PROTO__, __OTORP__, __PARAM__, __MANGLE__, __STDARG__, __V_, and __VARARG__ must not be used in the input source files.

The pp(3) based cpp uses the same transformations to pre-pre-process #pragma prototyped files. The transformations are disabled when pp(3) is preprocessing for ANSI. If the pp(3) cpp is used for all C compiles then *proto* need only be used when transporting files to places lacking pp(3).

Caveats

proto is not an ANSI C compiler.

Bugs

1) Function-returning-function prototypes and casts are not handled -- nested typedefs get around the problem.

2) *proto -i* uses the order of the K&R-style argument declarations rather than the order specified in the argument list. To work-around this problem, declare the arguments in the same order they appear in the argument list.

For example, say

```
fn(a,b,c)
int a;
char *b;
char *c;
{
}
```

rather than

```
fn(a,b,c)
char *c;
char *b;
int a;
{
}
```

Related Commands

cpp, pp(3)

silent Command

silent – prevent a shell command from being printed by the shell

Synopsis

silent *shell-command*

Description

silent prevents shell commands from being printed by the shell, if possible. silent must precede ignore (see ignore) if both are used.

Related Command

ignore

ss Command

ss – network system status

Synopsis

ss [–cilrtu] [*host*... **]**

Description

ss lists the system status for hosts on the local network. The status information is generated by a system status demon ssd(8) running on each of the hosts. This information is also available through the cs(3) routine csstat.The system status information is:

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 moused.
%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**).

If *host* arguments are specified then the listing is restricted to those hosts. If no options are specified then the listing is sorted by host name. Any other sorting is done in order from best to worst, e.g., the best load averages are small, the best idle times are large. The option order determines the sort order from highest to lowest precedence. The 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

Caveats

A system will be reported as being down until ssd(8) is up and running. The initial report is also subject to network file system latencies.

Files

share/lib/ss/.log	demon error log
share/lib/ss/*host*	status for *host*

Related Commands

coshell(3), cs(3), ssd(8)

Command-Line Options

2

When the nmake command is run, a series of options can be specified on the command line. This section contains manual pages for all the nmake command-line options.

Conventions

The following conventions are used:

- One or more command-line options can be used each time the nmake command is run.
- Option values are saved in the statefile. Once an option is turned on, it stays on every time nmake is executed until it is explicitly turned off.
- Options can be specified on the nmake command line by using the -o [no]*name*[=*value*] format. For example, to run nmake with the accept option turned on, use the following nmake command line:

  ```
  nmake -o accept
  ```
- To run nmake with the accept option turned off, use the following nmake command line:

  ```
  nmake -o noaccept
  ```
- Values can be specified with some options. For example, the following nmake command line can be used to set the jobs option to a level of 3:

  ```
  nmake -o jobs=3
  ```

- Although command-line options can be placed anywhere on the command line, the argument for an option must follow the option immediately. For example, if the **-f** option is specified, the *filename* must immediately follow the option.
- Some of the more popular command-line options can be run with a corresponding flag letter. For example, to use the accept option, the command line:

  ```
  nmake -A
  ```

  can be used instead of using the following command line:

  ```
  nmake -o accept
  ```

  The flag letter must be preceded by a minus (-) to turn the option on or a plus (+) to turn the option off. Most of the command-line options have flag letters; however, some do not.
- In the case of exec (flag letter **n**), **-n** means noexec, rather than exec.
- In the case of strictview (flag letter **V**), **-V** means nostrictview, rather than strictview.
- All the command-line options are turned off by default except in cases: strictview and writestate, which are turned on by default. For those options that can accept a value, the default value is set to 0, except the jobs option which has a default value of 1.
- Options can be specified in the makefile by using set *option* (e.g., set accept).
- Command-line options override options specified in the makefile.
- It is possible to declare and specify additional options with the option option, and display and list them with the .OPTIONS Special Atom.
- If a file named either Makeargs or makeargs exists in the current directory, it is checked (in the order given) for additional options. If both files exist, only the Makeargs file is used. Each line of the file is interpreted as a single argument and is inserted before the first argument of the original command line. All lines in the file are significant except comment lines that begin with a pound sign (#). (See Chapter 2, *Understanding Makefiles*, in the *Alcatel-Lucent nmake Product Builder User's Guide* for a discussion of the risks involved in using # comment lines.)

NOTE:
Makeargs is not viewpathed.

accept Option

accept - accept targets as up-to-date

Synopsis

```
nmake -A
nmake -o [no]accept
```

Description

This option treats any existing targets that are newer than their file prerequisites as up-to-date; these targets will not be made. This option also changes the time-stamp of statefiles.

Example

This example runs nmake three times on the same makefile. The first time, nmake is invoked normally, and the target hello is built.

The second time, the time-stamp of the target is changed before running nmake. Although this makes the target newer than its prerequisite, the target is still rebuilt because its time-stamp is newer than that of the statefile, Makefile.ms.

The third time, the time-stamp of the target is changed again, but this time thte target is not rebuilt, because the -A option is specified on the nmake command line.

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Contents of Makefile

```
FILES = hello.c
hello :: $(FILES)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

Change time-stamp of target

```
touch hello
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -o hello hello.o
```

Change time-stamp of target

```
touch hello
```

Run nmake

```
$ nmake -A
```

Output

base Option

base - compile a base or global makefile

Synopsis

nmake -b
nmake -o [no]base

Description

This option is used with the compile option (-c) and the file option (-f) to compile base or global makefiles.

Example

In this example, the -b option is used to compile a global makefile (g.mk).

Run nmake

```
$ nmake -bc -f g.mk
```

Output

Produces g.mo

Related Options

compile
file

believe Option

believe - believe statefile information

Synopsis

nmake -B
nmake -o believe=*level*

Description

This option tells nmake to believe the statefile information. When it is used in the form -o believe=*level*, it tells nmake to believe the statefile information for views greater than or equal to *level*. This provides an optimization method for projects that can certify that official files in views greater than or equal to *level* are up-to-date and have not been changed. In cases where there are a large number of files on the lower levels and only a few on the top levels, this option may save significant time by making it unnecessary to make stat(2) system calls on the lower-level files.

byname Option

byname - set an option by name

Synopsis

nmake -o byname=[[no]*name*=*value***]...**
nmake -o [[no] *name*=*value***]...**

Description

This option is used to set the option specified in *name*. If *name* is preceded by the word no, the option is turned off. A value can be given in the form *name*=*value* where appropriate.

Example

In this example, the -o option is used to set nmake's debugging trace to a level of 3. In practice, the byname form is not used, since entering

```
$ nmake -o byname=debug=3
```

is the same as entering

```
$ nmake -o debug=3
```

compatibility Option

compatibility - disable compatibility warning messages

Synopsis

nmake -C
nmake -o [no]compatibility

compile Option

compile - compile makefiles

Synopsis

```
nmake -c
nmake -o [no]compile
```

Description

This option is used to compile the makefiles into a single nmake object file and exit after compilation. The object file is not executed. To compile a global makefile, specify the -bc -f *makefile* options.

Example

In the first example, the command line compiles the makefile g.mk, creating the make object file g.mo. No actions specified in the makefile are performed, because g.mo is not executed.

Run nmake

```
$ nmake -bc -f g.mk
```

Output

Produces g.mo

In the second example, the command line compiles the makefiles x.mk and y.mk, creating the make object file x.mo. The x.mo makefile contains the object code for both the x.mk and y.mk makefiles. Again, no actions specified in the makefiles are performed, because x.mo is not executed. Because x.mk and y.mk are not global makefiles, the -b option is not used.

Run nmake

```
$ nmake -c -f x.mk -f y.mk
```

Output

Produces x.mo

Related Options

base

debug Option

debug - provide a debug trace

Synopsis

nmake -d*level*
nmake -o debug=*level*

Description

This option is similar to the .QUERY Special Atom in that it is used to provide a trace listing of nmake activity. However, unlike the .QUERY Special Atom, this option does not allow the user to debug makefiles interactively.

If *level* is used, debug information will be provided for all levels less than or equal to *level* . Higher levels generate more debug information.

If the debug leve is 1, debug activity is disabled until after the .INIT target has been made. If the debug level is less than 20, all levels are disabled during early bootstrap.

Level	Kind of Debugging Trace
0	No trace
1	Basic nmake trace (targets, prerequisites, triggered actions)
2	Major actions (reading statefile, binding information, scan strategy information)
3	Option settings, make object files
4	State variables
5	Directory and archive scan
6	coshell commands and messages
7	Makefile input lines
8	Lexical analyzer states
9	Metarules applied
10	Variable expansions
11	bindfile ( ) trace
12	Files added to hash
13	New atoms
14	Hash table usage
20	Bootstrap trace

Example

In the first example below, nmake is run without the -d option. This is equivalent to specifying a debug level of 0; no trace is proveded.

In the second example, nmake is run with a debug level of 1, which provides basic nmake trace information. An abbreviated version of the actual output is provided.

Contents of Makefile

```
FILES = hello.c
hello :: $(FILES)
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- \
-c hello.c
+ cc -O -o hello hello.o
```

Run nmake

```
$ nmake -d1
```

Output

```
make: debug: make(hello) [No such file or directory]
make: debug: time(hello) = Jan 10 12:20:48 1998
make: debug: make(hello.o)
make: debug: time(hello.o) = Jan 10 12:20:47 1998
make: debug: make(hello.c)
make: debug: time(hello.c) = Jan 10 12:20:26 1998
make: debug: make(.SCAN.c)
make: debug: time(.SCAN.c) = not found
make: debug: time(.SCAN.c) = Jan 10 12:23:47 1998
make: debug: time(hello.c) = Jan 10 12:23:47 1998
make: debug: make(hello.c)
make: debug: time(hello.c) = Jan 10 12:23:47 1998
make: debug: make((CC))
make: debug: time((CC)) = Jan 10 12:23:46 1998
make: debug: make(.CC.PROBE.)
make: debug: time(.CC.PROBE.) = not found
make: debug: make(/nmake3.1.2/lib/probe/C/\
make/361FBDCFrucbcc)
```

```
make: debug: time(/nmake3.1.2/lib/probe/C/\
make/361FBDCFrucbcc) = Dec 17 20:29:53 1997

make: debug: time(.CC.PROBE.) = Jan 10 12:23:47 1998
make: debug: make((CPP))
make: debug: time((CPP)) = Jan 10 12:23:46 1998
make: debug: make(.CC.PROBE.)
make: debug: time(.CC.PROBE.) = Jan 10 12:23:47 1998
make: debug: time((CPP)) = Jan 10 12:23:47 1998
make: debug: time((CC)) = Jan 10 12:23:47 1998
make: debug: make((CCFLAGS))
make: debug: time((CCFLAGS)) = Jan 10 12:23:46 1998
make: debug: time((CCFLAGS)) = Jan 10 2:23:46 1998
make: debug: triggering hello.o action using %.c>%.o
make: debug: time(hello.o) = Jan 10 12:23:48 1998
make: debug: make((CCLD))
make: debug: time((CCLD)) = Jan 10 12:23:47 1998
make: debug: time((CCLD)) = Jan 10 12:23:47 1998
make: debug: make((CCFLAGS))
make: debug: time((CCFLAGS)) = Jan 10 12:23:46 1998
make: debug: make((F77FLAGS))
make: debug: time((F77FLAGS)) = Jan 10 12:23:47 1998
make: debug: time((F77FLAGS)) = Jan 10 12:23:47 1998
make: debug: make((LDFLAGS))
make: debug: time((LDFLAGS)) = Jan 10 12:23:47 1998
make: debug: time((LDFLAGS)) = Jan 10 12:23:47 1998

make: debug: triggering hello action
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
make: debug: time(hello) = Jan 10 12:23:51 1998
make: debug: waiting for hello
+ cc -O -o hello hello.o
make: debug: normal cleanup
make: debug: jobs 2 user 0.58s sys 6.43s
make: debug: normal exit [No such process]
```

disableautomaticprobe Option

disableautomaticprobe - prevent the compiler from being automatically probed

Synopsis

nmake -m
nmake -o disableautomaticprobe

Description

This option prevents nmake from automatically triggering a probe of the $(CC) compiler. When the option is in effect, nmake will not trigger a probe under any condition. The option allows users of build rules that do not involve $(CC) to conveniently bypass automatic probe and allow nmake to continue. For example, users of :JAVA: may not even have a C compiler and need a convenient way to suppress $(CC) based probe. The option is also useful to administrators who want to prevent end users from kicking off a probe.

The optional variable disable_probe_message can be set as "*ERRORLEVEL custom_message*" to further control the behavior when probe is needed but is disabled. If *ERRORLEVEL* is 2 or less then *custom_message* is issued as a warning and processing is continued. If *ERRORLEVEL* is greater than 2 then *custom_message* is issued as an error and nmake exits with an exit code of *ERRORLEVEL*-2. The default value of disable_probe_message is "3 probe file non-existent or out-of-date - contact nmake administrator".

Setting the CC variable to null is equal to setting disableautomaticprobe and disable_probe_message to null.

Example

The following example shows how setting the option in a project global makefile can be used to enforce a policy where users cannot run probe.

Excerpt of global makefile

```
set disableautomaticprobe
disable_probe_message = 3 probe needed, contact local administrator x1234
```

Run nmake

```
nmake
```

Output

```
make: ".LOCALPROBE", line 17: probe needed, contact local administrator x1234
```

errorid Option

errorid - append an ID to an error

Synopsis

nmake -E *id*
nmake -o errorid=*id*

Description

This option allows the user to provide a string (*id*) to be appended to the error and debug trace identificatifier, e.g., make. Such a string is especially useful for tracing recursive nmake actions.

Example

In this example, nmake is run twice, once with and once without the -E command-line option. In the first case, nmake reports an error because the file myinclude.h cannot be found. In the second case, the error message is associated with the id demo, provided on the command-line by the -E option.

Contents of Makefile

```
FILES = hello.c
hello :: $(FILES)
```

Contents of hello.c

```
#include "myinclude.h"
main()
{
printf("Hello World\n");
}
```

Run nmake

```
nmake
```

Output

```
make: don't know how to make hello\
 : hello.o : hello.c : myinclude.h
```

Run nmake again

```
$ nmake -E demo
```

Output

```
make-demo: don't know how to make \
hello : hello.o : hello.c : myinclude.h
```

exec Option

exec - trace and print shell actions

Synopsis

```
nmake -n
nmake -o noexec
```

Description

When nmake is run with the -o noexec option, shell actions are traced and printed, but not executed. (exec is on by default.) Also inhibited are makefile compilations and updating of the statefile. However, shell actions for targets with the .ALWAYS Special Atom are executed even if -o noexec is specified.

Example

In this example, the shell actions are traced and printed, but not executed; therefore, the output does not include the message "Hello World.".

Contents of Makefile

```
FILES = hello.c
hello :: $(FILES)
```

Contents of hello.c

```
main ()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake -n
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

expandinclude Option

expandinclude - make include expansion

Synopsis

```
nmake -p
nmake -o [no]expandinclude
```

Description

expandinclude causes files referenced by the nmake *'include'* statement to be textually expanded into the including-file (similar to the #include preprocessor directive for C/C++ files). The textually expanded information is placed in the file **$(MAKEFILE:B).mi**. So, if the name of the first makefile nmake reads is Makefile or myrules.mk, then the information is placed in the file Makefile.mi or myrules.mi, respectively.

NOTE:
When used with other options expandinclude does not disable the other options - they will be obeyed as usual.

Example

The following example illustrates its use and how it may be used a debugging aid. (The 'make' probe information file may also be contained in the .mi file as a consequence of being included by the base rules, Makerules.mk).

Contents of Makefile

```
include rules.mk
$(TARG) :: $(FILES)
```

Contents of rules.mk

```
CC = /tools/sun_workshop/SUNWspro/bin/cc
FILES = hello.c
TARG = hello
```

Run nmake

```
$ nmake -p
```

Output

```
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp /tools/sun_workshop/SUNWspro/bin/cc -O -I-D/
tools/nmake/sparc5/lu3.3/lib/probe/C/pp/BFE0A4FEobincc -I- -c hello.c
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp /tools/sun_workshop/SUNWspro/bin/cc -O -I-D/
tools/nmake/sparc5/lu3.3/lib/probe/C/pp/BFE0A4FEobincc -I- -o hello hello.o
```

Makefile.* created

```
$ ls -l Makefile.*
-rw-r--r--     1   nmake                nmake   1481 Jun23 2000 Makefile.mi
-rw-r--r--     1   nmake                nmake   5356 Jun23 2000 Makefile.mo
-rw-r--r--     1   nmake                nmake   13945 Jun23 2000 Makefile.ms
```

Contents of Makefile.mi

```
/*Beginning Makefile*/
include rules.mk

/*Beginning rules.mk
CC = /tools/sun_workshop/SUNWspro/bin/cc
FILES = hello.c
TARG = hello
Ending rules.mk*/

$(TARG) :: $(FILES)

/*Beginning /tools/nmake/sparc5/lu3.3/lib/probe/C/make/BFE0A4FEobincc
CC.CC = /tools/sun_workshop/SUNWspro/bin/cc
CC.ALTPP.FLAGS =
CC.ALTPP.ENV =
CC.DIALECT = ANSI DOTI DYNAMIC
CC.DYNAMIC = -dy
CC.LD = ld
CC.MEMBERS = `$(NM) $(NMFLAGS) $(*:N=*$(CC.ARCHIVE):O=1) | $(SED)
$(NMEDIT) -e "s/^/-u /"`
CC.NM = nm
CC.NMEDIT = -e '/[  ][TDBC][                    ][          ]*[_A-Za-z]/!d' -e 's/.*[][TDBC][
][                 ]*//'
CC.NMFLAGS = -p
CC.PIC = -KPIC
CC.READONLY =
CC.SHARED = -G
CC.STATIC = -dn
CC.STDINCLUDE = /tools/sun_workshop/SUNWspro/SC5.0/include/cc /usr/include
CC.STDLIB = $(LD_LIBRARY_PATH:/:/ /G) /apps/tools/sparc5/SUNWmotif/lib /usr/ccs/lib /
tools/sun_workshop/SUNWspro/SC5.0/lib /usr/lib /lib
CC.SUFFIX.ARCHIVE = .a
CC.SUFFIX.COMMAND =
CC.SUFFIX.OBJECT = .o
CC.SUFFIX.SHARED = .so
CC.SUFFIX.SOURCE = .c
CC.SUFFIX.STATIC =
CC.SYMPREFIX =
CC.INSTRUMENT =
CC.PREROOT =
CC.UNIVERSE = ucb
Ending /tools/nmake/sparc5/lu3.3/lib/probe/C/make/BFE0A4FEobincc*/

/*Ending Makefile*/
```

Example

Let’s say that we changed our Makefile to not "include rules.mk" but to accept it using the -g (global makefile) option -- rules.mk remaining the same. The only noticeable difference in the Makefile.mi is the content of the global makefile appearing before the Makefile’s content, as the engine normally reads the former before the latter.

Contents of Makefile

```
$(TARG) :: $(FILES)
```

Run nmake

```
$ nmake -p -g rules.mk
```

Output

```
<same as before>
```

Makefile.* created

```
$ ls -l Makefile.*
<same as before>
```

Content of Makefile.mi

```
/*Beginning rules.mk
CC = /tools/sun_workshop/SUNWspro/bin/cc
FILES = hello.c
TARG = hello
Ending rules.mk*/

/*Beginning Makefile*/
$(TARG) :: $(FILES)

/*Beginning /tools/nmake/sparc5/lu3.3/lib/probe/C/make/BFE0A4FEobincc
CC.CC = /tools/sun_workshop/SUNWspro/bin/cc
CC.ALTPP.FLAGS =
                .
                .
                .
                .
CC.INSTRUMENT =
CC.PREROOT =
CC.UNIVERSE = ucb
Ending /tools/nmake/sparc5/lu3.3/lib/probe/C/make/BFE0A4FEobincc*/

/*Ending Makefile*/
```

expandview Option

expandview - generate full path names for viewpathing

Synopsis

```
nmake -x
nmake -o [no]expandview    .
```

Description

expandview generates the full path of a file rather than the simple file name for viewpathing.

Example

In this example, VPATH is set to $PWD/dev:$PWD/ofc. In the ofc directory, there are two files, Makefile and hello.c. The dev directory is empty.

Contents of Makefile

```
hello :: hello.c
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Change directory

```
$ cd $PWD/dev
```

Run nmake

```
$ nmake -Mdynamic -K
```

Output

```
info mam dynamic 00000 07/17/94 Alcatel-Lucent nmake (Bell Labs) alu3.9 07/05/2007
info start 922213434
info pwd /home/bugati/pete/lu3.2/src/cmd/nmake/dev
make Makefile
make /tools/nmake/sparc5/v3.1.2/lib/make/makerules.mo
done /tools/nmake/sparc5/v3.1.2/lib/make/makerules.mo
make /tools/nmake/sparc5/v3.1.2/lib/probe/C/make/BFE0A4FEobincc
done /tools/nmake/sparc5/v3.1.2/lib/probe/C/make/BFE0A4FEobincc dontcare
done Makefile
make Makefile.mo
exec - : compile into make object
done Makefile.mo
```

```
setv INSTALLROOT $HOME
make hello
make hello.o
make hello.c
make /usr/include/stdio.h implicit
make /usr/include/sys/va_list.h implicit
done /usr/include/sys/va_list.h dontcare
make /usr/include/sys/feature_tests.h implicit
done /usr/include/sys/va_list.h dontcare
make /usr/include/sys/feature_tests.h implicit
done /usr/include/sys/feature_tests.h dontcare
done /usr/include/stdio.h
done hello.c
make hello.c
done hello.c
prev /tools/nmake/sparc5/v3.1.2/lib/probe/C/make/BFE0A4FEobincc
exec - ppcc  -i  /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O  -I-D/tools/nmake/spar
c5/v3.1.2/lib/probe/C/pp/BFE0A4FEobincc  -I-   -c /home/bugati/pete/lu3.2/src/cm
d/nmake/ofc/hello.c
done hello.o virtual
+ 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 /home/bugati/pete/lu3.2/src/cmd/nmake/ofc
/hello.c
code hello.o 0 922213435
exec - ppcc  -i  /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O    -o hello hello.o
done hello virtual
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o hello hello.o
code hello 0 922213435
make Makefile.ms
exec - : compile into make object
done Makefile.ms
info finish 922213435 0
```

Run nmake again

```
$ nmake -Mdynamic -K -x
```

Output

```
info mam dynamic 00000 07/17/94 Alcatel-Lucent nmake (Bell Labs) alu3.9 07/05/2007
info start 922213518
info pwd /home/bugati/pete/lu3.2/src/cmd/nmake/dev
make /home/bugati/pete/lu3.2/src/cmd/nmake/ofc/Makefile
make /tools/nmake/sparc5/v3.1.2/lib/make/makerules.mo
done /tools/nmake/sparc5/v3.1.2/lib/make/makerules.mo
make /tools/nmake/sparc5/v3.1.2/lib/probe/C/make/BFE0A4FEobincc
done /tools/nmake/sparc5/v3.1.2/lib/probe/C/make/BFE0A4FEobincc dontcare
done /home/bugati/pete/lu3.2/src/cmd/nmake/ofc/Makefile
make Makefile.mo
exec - : compile into make object
done Makefile.mo
setv INSTALLROOT $HOME
make hello
make hello.o
make hello
make hello.o
make /home/bugati/pete/lu3.2/src/cmd/nmake/ofc/hello.c
make /usr/include/stdio.h implicit
make /usr/include/sys/va_list.h implicit
done /usr/include/sys/va_list.h dontcare
make /usr/include/sys/feature_tests.h implicit
done /usr/include/sys/feature_tests.h dontcare
done /usr/include/stdio.h
done /home/bugati/pete/lu3.2/src/cmd/nmake/ofc/hello.c
make /home/bugati/pete/lu3.2/src/cmd/nmake/ofc/hello.c
done /home/bugati/pete/lu3.2/src/cmd/nmake/ofc/hello.c
prev /tools/nmake/sparc5/v3.1.2/lib/probe/C/make/BFE0A4FEobincc
exec - ppcc  -i  /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O  -I-D/tools/nmake/spar
c5/v3.1.2/lib/probe/C/pp/BFE0A4FEobincc  -I-   -c /home/bugati/pete/lu3.2/src/cm
d/nmake/ofc/hello.c
done hello.o virtual
+ 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 /home/bugati/pete/lu3.2/src/cmd/nmake/ofc
/hello.c
code hello.o 0 922213520
exec - ppcc  -i  /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O    -o hello hello.o
done hello virtual
exec - ppcc  -i  /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O    -o hello hello.o
done hello virtual
+ ppcc -i /tools/nmake/sparc5/v3.1.2/lib/cpp cc -O -o hello hello.o
code hello 0 922213520
make Makefile.ms
exec - : compile into make object
done Makefile.ms
info finish 922213519 0
```

explain Option

explain - print explanation for actions

Synopsis

```
nmake -e
nmake -o [no]explain
```

Description

This option prints an explanation for actions taken as nmake executes.

Example

In this example, nmake is run twice, the first time without the -e option, the second time with it. Both runs use the same makefile and build the same target. When the -e option is specified, explanations are printed for nmake's actions.

Contents of Makefile

```
hello :: hello.c
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

Change time-stamp of hello.c

```
$ touch hello.c
```

Run nmake again

```
$ nmake -e
```

Output

```
hello.c [Jan 10 12:20:26 1998] has changed [Jan 10 12:17:54 1998]
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

file Option

file - specify makefile

Synopsis

nmake -f *makefile*
nmake -o file=*makefile*

Description

This option specifies the file or files to be used as makefile(s). If more than one name is suppied, the files are read in order from left to right.

Example

In the example below, nmake is run with the -f command-line option. This forces nmake to read the specified makefile.

Contents of mymake.mk

```
FILES = hello.c
hello :: $(FILES)
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake -f mymake.mk
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

force Option

force - make all active targets out-of-date

Synopsis

```
nmake -F
nmake -o [no]force
```

Description

This option forces all active targets to be out-of-date and causes them all to be remade.

Example

In this example, nmake is run three times; the first two times without the -F command-line option. In the first run, nmake creates the target hello. In the second, it does nothing because the target is up-to-date. In the third run, nmake is invoked with the -F option, which forces the target hello to be made.

Contents of mymake.mk

```
FILES = hello.c
hello :: $(FILES)
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

Run nmake again

```
$ nmake
```

Output

Run nmake a third time

```
$ nmake -F
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

global Option

global - specify a global makefile

Synopsis

nmake -g *makefile*
nmake -o global=*makefile*

Description

This option specifies the file (*makefile*) to be used as a global makefile. Global makefiles are used to expand nmake's standard knowledge and allow default base rules or specified makefiles to be used at the same time. Rules included in a global makefile with the same target as default base rules take precedence over the default base rules. Rules specified in the local makefile with the same target as the global makefile take precedence over the global rules.

Global makefiles can be source or object files written in the same manner as any other makefile. They are generally used to define a standard project environment to be used when executing nmake commands.

nmake always uses one global makefile containing the default base rules. Any number of other global makefiles can be specified with the -g option on the command line. Administrators of Alcatel-Lucent nmake can use a Korn shell alias to ensure that all users in one project use the global makefile, e.g., projectmake='nmake -g global.mk'.

ignore Option

ignore - ignore exit codes from shell commands

Synopsis

```
nmake -i
nmake -o [no]ignore
```

Description

This option causes nmake to ignore error exit codes from shell action commands.

Example

In this example nmake is run twice, once without the ignore option, and once with it. In the first case, nmake receives a shell error code of 2 and then stops. In the second case, the same exit error code is received, but, because the -i option is in effect, it is ignored and nmake continues processing.

Contents of Makefile

```
testignore:
        cp $(>) $(<)
        : after copying the file over
```

Run nmake

```
$ nmake
```

Output

```
+ cp testignore
cp: Insufficient arguments (1)
Usage:    cp f1 f2
     cp f1 ... fn d1
make:*** exit code 1 making testignore
```

Run nmake again

```
$ nmake -i
```

Output

```
+ cp testignore
cp: Insufficient arguments (1)
Usage:          cp f1 f2
                cp f1 ... fn d1
+ : after copying the file over
```

ignorelock Option

ignorelock - override locking protocols

Synopsis

nmake -K
nmake -o [no]ignorelock

Description

This option overrides statefile locking protocols, thus allowing multiple occurrences of nmake to execute on a given makefile simultaneously.

CAUTION:
Use this option with care. It should be used only if the lock is wrong. (Some implementations drop a lock file.)

Example

If more than one invocation of nmake in the same directory is trying to access a given makefile, only the first nmake process can execute the makefile. A message similar to the following appears on all the other nmake instances when ignorelock is not used:

```
make: warning: another make has been running on makefile in . \
for the past 2.00s by user richb from host jaguar

make: use -K to override.
```

When using ignorelock, the following message is displayed when the first nmake process completes:

```
make: warning: the statefile lock on makefile has been overridden
```

include Option

include - pass directory to cpp for preprocessing

Synopsis

nmake -I *directory*
nmake -o [no]include

Description

This option passes the directory name, *directory*, as a value to Alcatel-Lucent nmake's cpp for makefile preprocessing.

Example

In this example, nmake is run twice, once without the include option, and once with it. In the first case, nmake cannot build the target x because it does not know where to find the file g.mk, which contains the relative paths to the files jk.h and jk1. (The file is located in the directory one level up.) In the second case, nmake is invoked with the include option, which tells nmake to search the two included directories for files it needs.

Contents of Makefile

```
include "g.mk"
x :: x.c
```

Contents of x.c

```
#include <time.h>>
#include "jk.h"
#include "jk1"
main()
{
}
```

Contents of g.mk (one level up)

```
.SOURCE.h : incl incl1
```

Run nmake

```
$ nmake
```

Output

```
make: "makefile", line 1: g.mk: cannot read include file
make: don't know how to make x : x.o : x.c : jk1
```

Run nmake again

```
$ nmake -I ..
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/ \
lib/probe/C/pp/361FBDCFrucbcc -I/usrs8/bob/incl \
-I/usrs8/bob/incl1 -I- -c x.c
+ cc -O -o x x.o
```

jobs Option

jobs - limit number of concurrent target actions

Synopsis

nmake -j*level*
nmake -o jobs=*level*

Description

This option specifies that nmake can execute a number of target actions concurrently. The value (*level*) specifies how many actions can be performed concurrently. For example, jobs=10 allows up to 10 actions to execute at the same time.

jobs=0 causes nmake to block other processing until the completion of each action before proceeding.

jobs=1 (default) allows nmake to determine the next out-of-date target while the previous action is executing.

Example

In this example, nmake is invoked with the job level set to 3, which allows three target actions to be executed concurrently.

Run nmake

```
$ nmake -o jobs=3
```

Result

nmake can process three actions concurrently.

keepgoing Option

keepgoing - continue processing

Synopsis

```
nmake -k
nmake -o [no]keepgoing
```

Description

This option causes nmake to continue working on targets that do not depend on the current target, even if the action for the current target returns a nonzero status, which indicates an error.

Example

In this example, nmake is run twice, once with and once without the -k command-line option. Both cases use the same makefile. The target testkeepgoing contains two prerequisites, target1 and target2. In the first run, the shell gives an error message, because there are no prerequisites specified for target1, and $(>) evaluates to null. A nonzero status is returned, nmake halts, and target2 is never made. In the second run, the shell gives an error message and a nonzero status is returned, but, because the -k option is used, nmake continues processing, that is, it tries to make target2.

Contents of Makefile

```
testkeepgoing: target1 target2
target1:
        cp $(>) $(<)
        : after copying the file over
target2:
        >$(<)
```

Run nmake

```
$ nmake
```

Output

```
+ cp target1
cp: Insufficient arguments (1)
Usage:    cp f1 f2
     cp f1 ... fn d1
make: *** exit code 1 making target1
```

Run nmake again

```
$ nmake -k
```

Output

```
+ cp target1
cp: Insufficient arguments (1)
Usage:      cp f1 f2
        cp f1 ... fn d1
make: *** exit code 1 making target1
+ 1> target2
make: *** 1 action failed
```

list Option

list - list variable and rule definitions

Synopsis

```
nmake -l
nmake -o [no]list
```

Description

This option lists variable and rule definitions included in the makefile. After reading the makefile and producing the listing, nmake stops processing. Targets are neither checked nor updated.

Example

This example displays an excerpt from a listing file produced by using the -l option.

Run nmake

```
$ nmake -l
```

Output

```
/* Variables */
PPCC = $(MAKELIB:D:D)/bin/ppcc
CC = cc
VGRIND = vgrind
VOFFSET = .
".FUNCTION" =
_version_ =
IGNORE = ignore
GREP = grep
MAKEARGS = Makeargs:makeargs
VROOT = .
                    l
                    l
                    l
.MULTIPLE : .MULTIPLE
%.c :
```

mam Option

mam - generate mamfiles

Synopsis

nmake -M{static|dynamic|regress} [:*file*[:[*label*][:*root*]]][,noport]
[,dontcare]
nmake -o nomam
nmake -o mam={static|dynamic|regress} [:*file*[:[*label*][:*root*]]]
[,noport]\[,dontcare]

Description

This option generates the make abstract machine language equivalent of the current makefile and writes it either to the standard output or a user-specified file. With the -M option, static is for generating mamfiles, dynamic is for tracing nmake execution steps, and regress is for canonicalizing regression testing on different hosts. If using neither dynamic nor label, file implies that output appends to file, otherwise it implies overwrite. label must be numeric (usually a process ID), and root specifies that all files listed are relative to this root directory. noport inhibits porting hints, while dontcare lists targets that contain the .DONTCARE Special Atom.

If label is used, recursive child builds utilizing $(-) will have label specified as the process id of the parent.

When a mam output file is specified in dynamic mode, the output file name for recursive child builds that do not use :MAKE: (but utilize $(-)) defaults to <output_file>.pid, where *pid* is the process id of the parent.

When a mam output file is specified in dynamic mode, the output file name for recursive child builds that use :MAKE: with either a directory specification or makefile name defaults to <output_file>.<name>, where *<name>* is either the directory name or the makefile associated with the recursive child build. The naming convention generalizes for recursive builds of arbitrary depth, such that <output_file>.<name> would imply <output_file>.<name>.<name2> for the file name of a recursive build initiated from the recursive build within *<name>*, using :MAKE:.

mismatch Option

mismatch - versions of nmake and statefile differ

Synopsis

nmake -o [no]mismatch

Description

This option tells nmake to run with the readstate and accept options on when the statefile version does not match the nmake version.

never Option

never - trace and print shell actions

Synopsis

nmake -N
nmake -o never

Description

Like the exec option, the -N or -o never option, will trace and print the shell actions for targets and not execute the action, along with not compiling the makefile or updating the statefile. However, unlike the exec option, shell actions for targets with the .ALWAYS Special Atom are not executed. So, the difference between this option (**-N**) and the exec option (**-n**) is that **-N** overrides .ALWAYS.

option Option

option - define a new option name

Synopsis

nmake -o option=*flag,name***[,sbn][x] [,***function***] -o** *name***[=***value***]**
nmake -o option=*flag,name***[,sbn][x] [,***function***]** *-flag*

Description

This option allows the user to define a new option called *name*. The user must specify both a *flag* to be used on the nmake command line and a *name* to be used as an argument to the -o command-line option.

The *type* is optional; the default is Boolean (**b**). The types allowed are:

b	*name* is a Boolean option
n	*name* is a numeric option
s	*name* is a string-valued option
x	do not expand *name* in $(-)

The optional *function* is the name of a function in the makefile that acts based on the *value* of *name*.

NOTE:
No spaces are allowed around the equal sign between *name* and *value*. Do not use quotation marks in the assignment.

The value can be used on the command line only in the -o*name*=*value* format. If *name* is a numeric option and, if no value is assigned, the default value is 0.

On the command line, -*flag* is used to invoke *name*; +*flag* is used to remove *name*.

For example, if the user uses the option option to define:

-o option=u,unconditional,bx

the user can use either -u or -o unconditional on the command line. b specifies that unconditional takes Boolean values, and x specifies that they are not to be expanded in $(-).

Example

The following example shows how to define and use a new option.

Contents of Makefile

```
.FUNC : .FUNCTION .MAKE
        set $(%:/0/no/:/1//)force

set option=u,unconditional,bx,.FUNC
set unconditional

mark : .MAKE
        if "$(-unconditional)"
        print $(-unconditional)
        end
        if "$(-a)" == "zoo"
        print mark is $(-mark)
        end
        print $(-:@/ /=/)
```

Run nmake

```
$ nmake -o option=a,mark,s -o mark=zoo
```

Output

```
1
mark is zoo
-o=' force option=a,mark,s, mark=zoo '
```

override Option

override - override explicit actions

Synopsis

nmake -o [no]override

Description

This option causes the explicit actions to be overridden by applying matching metarule actions if possible. It also inhibits updates to the statefile.

Example

In this example, nmake is run twice, once with and once without the -o command-line option. Both instances use the same makefile. The makefile specifies that s.o is to be built from s.c using the explicit action : local rule, which simply displays the string local rule. In the first run, nmake is executed without the override command-line option. In this case, nmake performs the action specified in the makefile. In the second run, because the -o command-line option is specified, nmake applies an implicit metarule (%.o : %.c), instead of performing the action specified in the makefile.

Normally, rules specified in assertions take precedence; however, when the -o override option is specified, the matching metarule actions take precedence. Recall that the rule order of precedence is makefile rules, global rules, and default rules.

Contents of Makefile

```
s.o : s.c .IMPLICIT
        : local rule
```

Run nmake

```
$ nmake
```

Output

```
+ : local rule
```

Run nmake again

```
$ nmake -o override
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c s.c
```

Related Special Atom

.IMPLICIT

preprocess Option

preprocess - preprocess makefiles

Synopsis

nmake -o [no]preprocess

Description

This option forces makefiles to be preprocessed by Alcatel-Lucent nmake's cpp.

readstate Option

readstate - do not read statefile

Synopsis

nmake -S*level*
nmake readstate[=*level*]

Description

This option directs nmake to ignore the previous state by not reading some or all statefiles. When *level* is specified, the statefiles in all the viewpath levels up to and including *level* are read. The remaining statefiles are ignored. If *level* is omitted, all statefiles are ignored.

For example, -S and -S0 ignore all state files; -S1 reads only the statefile in the current viewpath node.

Example

In this example nmake is run three times, twice without the readstate option, the third time with it. All runs use the same makefile. In the first run, nmake is invoked to build the target hello. In the second run, nmake determines that all targets are up-to-date. In the third run, all the statefiles are ignored and the target hello is built.

Contents of Makefile

```
FILES = hello.c
hello :: $(FILES)
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

Run nmake again

```
$ nmake
```

Output

Run nmake a third time

```
$ nmake -S
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o
```

reread Option

reread - read input makefiles

Synopsis

nmake -o reread
nmake reread[=*level*]

Description

This option forces nmake to read the input makefiles, rather than loading the corresponding nmake object files. This option is generated automatically when inconsistencies are detected in nmake object files.

level specifies the number of times the makefile is to be reread. The default is 1.

Example

In this example, nmake is forced to read the input makefiles. Because everything is consistent with the statefile, nothing is rebuilt.

Run nmake

```
$ nmake -o reread
```

Output

ruledump Option

ruledump - list status of rules

Synopsis

```
nmake -r
nmake -o [no]ruledump
```

Description

This option lists detailed information about the status of each rule after all targets have been made.

Example

This example displays a short excerpt of a listing produced by using the ruledump option.

Contents of Makefile

```
FILES = hello.c
hello :: $(FILES)
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake -r
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/lib/probe/C/\
pp/361FBDCFrucbcc -I- -c hello.c
+ cc -O -o hello hello.o

/* Rules */

"" : [really old] compiled IGNORE

.MEMBER : [not found] compiled unbound
%.cc>%.o : [not found] metarule target use compiled unbound
 prerequisites: (CC) (CCFLAGS)
 action:
 $(CC) $(CCFLAGS) -c $(>)
.TMPLIST : [not found] internal readonly compiled unbound
```

.FORCE : [not found] attribute force compiled unbound

l
l
l

$(CIAFLAGS:N=-d*:/-d/) : [not found] compiled unbound

.MULTIPLE : [not found] attribute multiple compiled unbound

%.y>%.o==%.c : [not found] metarule use compiled unbound

scan Option

scan - inhibit scan algorithms

Synopsis

nmake -o [no]scan

Description

noscan disables the implicit file dependency scan algorithms controlled by the .SCAN and .SCAN.*suffix* rules. scan is on by default.

Example

In this example, nmake is run twice, once with the noscan option and once with the scan option. In the first run, an error is diplayed. In the second run, no error is produced, because nmake is able to use the implicit file dependency scan algorithms and so finds the file j.h included in file j.c.

Contents of Makefile

```
j :: j.c
```

Contents of j.c

```
#include "j.h"
main()
{
}
```

Run nmake

```
$ nmake -o noscan
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c j.c
"j.c", line 1: j.h: cannot find include file
make: *** exit code 1 making j.o
```

Run nmake again

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I. -I- -c j.c
+ cc -O -o j j.o
```

silent Option

silent - do not print shell actions

Synopsis

```
nmake -s
nmake -o [no]silent
```

Description

When the -s option is specified, nmake executes the makefile and builds all the targets, but shell actions are not displayed.

Example

In this example, the target is built, but the shell actions are not displayed, so there is no output.

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Contents of Makefile

```
FILES = hello.c
hello :: $(FILES)
```

Run nmake

```
$ nmake -s
```

Output

strictview Option

strictview - control viewpathing

Synopsis

```
nmake -V
nmake -o [no]strictview
```

Description

This option is on by default. Therefore, this option enables nostrictview viewpathing. The difference between strictview and nostrictview is in the order in which the source directories are searched. Given:

VIEWPATH = /v2:/v1
.SOURCE.h = ./include ./include2

the strictview interpretation is:

```
/v2 /v1 /v2/include /v1/include \
/v2/include2 /v1/include2 /usr/include
```

while the nostrictview interpretation is:

```
/v2 /v2/include /v2/include2 /v1 \
/v1/include /v1/include2 /usr/include
```

Example

In the following example, VPATH is set to $PWD/dev:$PWD/ofc. In the ofc directory, there are two files: Makefile and j.c. $PWD/dev is empty.

Contents of Makefile

```
.Source.h : incl incl2
j :: j.c
.DONE :
        : $(*.SOURCE.h)
```

Contents of j.c

```
#include "j.h"
Main()
{
}
```

Change directory

```
$ cd $PWD/dev
```

Run nmake

```
$ nmake
```

Output

```
+ : . /home3/u1/ofc incl /home3/u1/ofc/\
incl incl2 /home3/u1/ofc/incl2 /usr/include
```

This output represents the default strictview search for j.h.

Run nmake again

```
$ nmake -V
```

Output

```
+ : . incl incl2 /home3/u1/ofc /home3/u1/\
ofc/incl /home3/u1/ofc/incl2 /usr/include
```

This output gives the nostrictview search for j.h.

targetcontext Option

targetcontext - expand actions in the target directory

Synopsis

nmake -o targetcontext

Description

This option expands actions in the target directory when the target name contains a directory prefix. For example, if a target has a directory prefix a/b/c.o, then this target will be generated in the context of the prefix directory as follows:

```
cd a/b
<action>
```

This allows assertions like

```
system :: a/b/c.c c/d/c.c
```

which would normally generate a conflict by trying to generate two c.o files in the current directory. But if this option is used, one c.o file will be generated in the a/b directory, the other in the c/d directory.

test Option

test - enable tests

Synopsis

nmake -o [no]test
nmake -T *bits*

Description

In the **-T** *bits* version, this option enables the internal debug tests specified by the bit vector bits. Some tests enable verbose tracing, while others change the internal algorithms.

CAUTION:
Do not use this option unless you know what the tests do.

tolerance Option

tolerance - accept clock discrepancies

Synopsis

nmake -z *seconds*
nmake -o tolerance=*seconds*

Description

This option sets the number of seconds that can be accepted between networked systems as being equal, and not affecting the time-stamps checked, when determining whether a file is out-of-date.

This option may be necessary for certain network file system implementations that fail to maintain clock integrity between systems on the network.

A number (*seconds*) is specified as the value to indicate the allowable time (number of seconds) difference between the two systems.

When a client machine connects with a server machine through RFS (Remote File Sharing), the clock of the client is compared with the clock of the server to compute a time skew between the client and server time-stamps. The time skew is computed as follows:

skew = clock(client) - clock(server)

The skew is computed once when the client-server connection is made. Whenever the client stat()'s a file on the server, the time skew is automatically added in.

st.st_mtime' = st.st_mtime + skew

If either the client or the server crashes or reboots, a new skew is computed when the two machines are reconnected. This skew may be (and most likely will be) different from the skews of previous client-server connections.

nmake records file time-stamps in the statefile, expecting that a file's time-stamp remains constant as long as the file has not been touched. RFS invalidates the statefile by allowing a file's time-stamp to change (via a changing skew) even though the file has not been touched.

nmake partially solves the problem by allowing some fuzz in the time-stamp comparisons. As long as the RFS skew is less than the fuzz, the nmake algorithms remain constant.

The tolerance command-line option can be used to specify the range in number of seconds to be permitted in the skew.

Example

In the example shown below, nmake will allow 5 seconds difference between any two systems on a network when deciding whether a file is out-of-date.

Run nmake

```
$ nmake -z 5
```

touch Option

touch - change target modification dates

Synopsis

```
nmake -t
nmake -o [no]touch
```

Description

This option touches the modify date of the targets without executing any actions. Only existing targets are touched. It also changes the time-stamps of statefiles.

Targets with state-variable prerequisites can be touched only by using this option.

Example

In this example, a header file is used by two source files (x.c and y.c). nmake is then invoked to build the target, prog. A change is then made to the header file that affects only one of the source files, x.c. nmake is then invoked with the -t option, which modifies all of the target time-stamps to the current time (including that of x.o). The object file x.o is then removed. When nmake is invoked, only x.c is recompiled (to include the changes made to the header file).

Contents of Makefile

```
prog :: x.c y.c main.c
```

Contents of header.h

```
#define a 1
#define b 2
```

Contents of x.c

```
#include "header.h"
x()
{
printf("a is %d\n",a);
}
```

Contents of y.c

```
#include "header.h"
y()
{
printf("b is %d\n",b);
}
```

Contents of main.c

```
main()
{
        x();
        y();
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I. -I- -c x.c
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I. -I- -c y.c
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c main.c
+ cc -O -o prog x.o y.o main.o
```

Modify header.h

```
#define a 2
#define b 2
```

Run nmake again

```
$ nmake -t
```

Output

```
touch x.o
touch y.o
touch prog
```

Remove x.o

```
rm x.o
```

Run nmake a third time

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.1.2/lib -I-D/nmake3.1.2/\
lib/probe/C/pp/361FBDCFrucbcc -I. -I- -c x.c
+ cc -O -o prog x.o y.o main.o
```

vardump Option

vardump - list variable assignments

Synopsis

```
nmake -v
nmake -o [no]vardump
```

Description

This option lists assignments made to variables while nmake is executing. It is useful when used with the debug option.

Example

The following example shows a small example of the output produced by using the vardump option. The output has been shortened for brevity.

Contents of Makefile

```
FILES = hello.c
hello :: $(FILES)
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake -v
```

Output

```
MAKEPATH [ free ] =
VROOT [ free ] = .
VOFFSET [ free ] =
VOFFSET [ free ] = .
VROOT [ free ] = .
MAKEFILE [ free ] = Makefile
T [ free ] =
T [ free ] =
MAKERULES [ free ] = makerules
```

l
l
l

```
.ORIGINAL.MAIN. [ free ] = hello
T1 [ free ] =
T1 [ free ] = 0
T2 [ free ] =
T2 [ free ] = .SEMAPHORE
T1 [ free ] = 1

/* Variables */

CC [ ] = cc
CCC [ ] = $(MAKELIB:D:D)/bin/ccc
VGRIND [ ] = vgrind
VOFFSET [ free ] = .
.FUNCTION [ functional ] =
_version_ [ ] =

                    l
                    l
                    l

NMFLAGS [ ] = $(CC.NMFLAGS)
ARFLAGS [ ] = r
.SHARED. [ functional ] =
CC.PROBE [ free ] = /nmake3.1.2/lib/probe/C/\
make/361FBDCFrucbcc
CC.OBJ [ free ] = o
.PTR.OPTIONS. [ functional ] =
.MAMNAME. [ functional ] =
```

Related Options

debug

warn Option

warn - enable detailed warning messages

Synopsis

nmake -o [no]warn

writestate Option

writestate - control updating of statefile

Synopsis

```
nmake -o nowritestate
```

Description

This option allows the statefile to be updated whenever nmake processes a makefile. The option is on by default. nowritestate prevents the statefile from being updated upon exit.

Example

In the example shown below, Makefile.ms is not created or updated.

Examine statefile

```
$ ls -l Makefile.ms
```

Output

```
-rw-r--r-- 1 nmake 9463 Jan 10 17:44 Makefile.ms
```

Run nmake

```
$ nmake -o nowritestate
```

Examine statefile

```
$ ls -l Makefile.ms
```

Output

```
-rw-r--r-- 1 nmake 9463 Jan 10 17:44 Makefile.ms
```

3 Assertion Operators

The assertion operators and variables defined in the default base rules provide high-level access to basic nmake functionality (as a high-level programming language relates to a lower-level programming language). The :*identifier*: assertion operators described in this section should be used as much as possible when developing new makefiles. They provide all of the functionality that most systems require using simple operators that are parameterized for easy customization.

In the following descriptions, *lhs* and *rhs* are the lists of atoms appearing on the left-hand side and the right-hand side of the assertion operator, respectively. The common actions defined in the default base rules can be applied to the :*identifier*: assertion operators.

This section contains manual pages for all the nmake assertion operators.

: Assertion Operator

: - dependency operator

Description

The : dependency operator is the primitive nmake assertion operator. Dependency assertion operators are used when the action required to build a target is specifically defined in the action block. If no action is defined, the dependency assertion operator merely sets up a dependency relationship between the target list and the prerequisite list.

:: Assertion Operator

:: - source dependency operator

Description

The source dependency assertion operator is the most commonly used assertion operator. The target specified on the lhs is an executable target, an object target, an archive library target, or a shared library target that is built from the source files specified on the rhs. Other targets may also be built using user-defined metarules.

If the lhs is an archive or shared library and the target is installed using the install common action, the following is automatically asserted by default:
$(LIBDIR) :INSTALLDIR: *lhs*
(see :INSTALLDIR: for more information.)

On win32, if the lhs is a .lib or .dll target both a .lib and corresponding .dll are generated. If the install common action is asserted the .lib file is installed in $(LIBDIR) and .dll file in $(BINDIR). The following is asserted by default:
$(BINDIR) :INSTALLDIR: *lhs.*dll
$(LIBDIR) :INSTALLDIR: *lhs*.lib

If the lhs is an executable and the target is installed using the install common action, the following is automatically asserted by default:
$(BINDIR) :INSTALLDIR: *lhs*

If the lhs is not specified, the rhs is appended to the list of files for the save-oriented common actions (cpio**,** pax**,** etc.).

Examples of rhs files are source files, object files and libraries (-lname syntax is preferred). On win32, .res files specified on the rhs will also be included when linking the lhs executable.

Note that explicit :INSTALLDIR: assertions must precede :: to allow control over install locations for targets defined in the :: assertion.

Example

In the following example, the libsub.a library is built from the list of files specified in the FILES variable. When nmake is executed with the install common action, the library, when it is built, is copied to the $(LIBDIR) directory. The default value for $(LIBDIR) is $HOME/lib. When this example was documented, $HOME was set to /usrs1/bob.

Contents of Makefile

```
FILES = sub1.c sub2.c sub3.c
```

```
libsub.a :: $(FILES)
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
probe/C/pp/361FBDCFrucbcc -I- -c sub1.c
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
probe/C/pp/361FBDCFrucbcc -I- -c sub2.c
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
probe/C/pp/361FBDCFrucbcc -I- -c sub3.c
+ ar r libsub.a sub1.o sub2.o sub3.o
ar: creating libsub.a
+ ignore ranlib libsub.a
+ rm -f sub1.o sub2.o sub3.o
+ mkdir -p /usrs1/bob/lib
+ 2> /dev/null
+ ignore cp libsub.a /usrs1/bob/lib/\
libsub.a
```

Related Assertion Operators

:LIBRARY:

:ALL: Assertion Operator

:ALL: - build all rhs targets and all targets of :: assertions

Description

The :ALL: assertion operator causes the rhs targets and all targets built from source dependency assertions to be made when no command-line targets are specified. The position of the :ALL: assertion operator in the makefile must be before any target (see example below).

Example

The following example demonstrates that the target first (made with the : dependency assertion operator) and the targets target1 and target2 (made with the source dependency assertion operator) are built when nmake is invoked without specifying any targets on the nmake command-line and the :ALL: assertion operator is used.

Contents of Makefile

```
:ALL: first

target1 :: x.c

target2 :: y.c

first :
        : this is a test
second :
        : this is another test
```

Run nmake

```
$ nmake
```

Output

```
+ : this is a test
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c x.c
+ cc -O -o target1 x.o
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c y.c
+ cc -O -o target2 y.o
```

:COMMAND: Assertion Operator

:COMMAND: - lhs is variable resulting from shell command substitution

Description

The lhs is a variable whose value is the result of shell command substitution on the action with respect to prerequisites on the rhs.

Example

In the following example, the target tst displays the value of the present working directory. When this example was developed, the present working directory was /usrs1/bob/nmake.

Contents of Makefile

```
PRES_DIR :COMMAND:
        pwd

tst :
        : $(PRES_DIR)
```

Run nmake

```
$ nmake
```

Output

```
+ pwd
+ : /usrs1/bob/nmake
```

:COPY: Assertion Operator

:COPY: - copy file on rhs to file on lhs

Description

The :COPY: assertion operator copies the file on the rhs to the file on the lhs. The :COPY: assertion operator also marks the lhs with the .SPECIAL Special Atom, which prevents the lhs from being appended to .MAIN (.MAIN contains the list of all the targets that are to be built). Therefore, the lhs is not made unless it is needed by other assertions.

Examples

In the following example, since b1.c has been marked with the .SPECIAL Special Atom and b1.c is not appended to .MAIN, nmake generates an error message. (Compare this example to the second example.)

Contents of Makefile

```
b1.c :COPY: b.c
```

Run nmake

```
$ nmake
```

Output

```
make: Makefile: a main target must be specified
```

In the following example, b.c is the only source file in the current directory. When b.c is needed to build b1.o, nmake links or copies b.c to b1.c.

Contents of Makefile

```
b1.c :COPY: b.c
b1.o :
```

Run nmake

```
$ nmake
```

Output

```
+ cmp -s b.c b1.c
+ rm -f b1.c
+ cp b.c b1.c
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- \ -c b1.c
```

:FUNCTION: Assertion Operator

:FUNCTION: - define lhs as a functional variable

Description

The lhs is defined as a functional variable whose value is determined by the action at execution time. The format is:

```
var :FUNCTION: prerequisites
        action
```

Example

In the following example, we create a functional variable view whose function is to return the bound or bindable tokens of its argument, specified as $(%).

Contents of Makefile

```
/*
test functional operator and with arguments
*/
.SOURCE : src
t:
        : $(view x.c)

view :FUNCTION:
        return $(%:T=F)
```

Run nmake

```
$ nmake
```

Output

```
+ : src/x.c
```

:INSTALL: Assertion Operator

:INSTALL: - install file

Description

If the lhs is omitted, the rhs is added to the list of targets made by the install common action. Otherwise, the lhs is installed by the install common action by copying the file on the rhs. If both the lhs and rhs are omitted and an *action* is specified, this action overrides the default install action. The default install action saves old install targets in *target*.old. If the $(clobber) base rule variable is non-null, the default install action does not move old install targets to *target*.old.

If the target is already installed, it is reinstalled only if it changes. The UNIX cmp command is used to detect a difference between the installed target and the new file. The compare operation can be disabled by setting the $(compare) base rule variable to 0.

The user ID, group ID, and mode of the file can be set during the installation. (See :INSTALLDIR: for details.)

Examples

In the following example, when the install common action is specified on the command-line, the file1.c prerequisite is copied to the file1 target.

Contents of Makefile

```
file1 :INSTALL: file1.c
```

Run nmake

```
$ nmake install
```

Output

```
+ ignore cp file1.c file1
```

In the following example, when the install common action is specified, file2 is built from file2.c.

Contents of Makefile

```
:INSTALL: file2
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- \ -o file2 file2.c
```

In the following example, the :: operator sets up a default install assertion. The BINDIR variable is set to ../bin so that prog will be installed there instead of in the default value of BINDIR, which is $HOME/bin.

Contents of Makefile

```
BINDIR = ../bin

prog :: a.c b.c
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c a.c
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c b.c
+ cc -O -o prog a.o b.o
+ mkdir -p ../bin
+ 2> /dev/null
+ : new install rule
+ ignore cp prog ../bin/prog
```

Related Assertion Operators

:INSTALLDIR:, :INSTALLMAP:

:INSTALLDIR: Assertion Operator

:INSTALLDIR: - install file in directory

Description

When this assertion operator is used in an assertion, the files on the rhs are installed into the directory specified on the lhs by the install common action. If the lhs is omitted, the rhs will not be installed. If the directory specified on the lhs does not exist, the directory will be created before the files are installed.

The user ID, group ID, and the mode of a file can be changed during the installation. The second example below shows how these attributes can be changed.

:INSTALLDIR: makes directories to any depth; the third example shows the creation of such directories.

Examples

In the following example, two directories are created when nmake is executed with the install common action. The directories created are $HOME/include and $HOME/lib. When the makefile is processed, the header file headerX.h is copied to the $HOME/include directory and the library libsub.a is built and copied to the $HOME/lib directory. (The sub1.c file includes the header file headerX.h.) By default, $(INCLUDEDIR) is $(INSTALLROOT)/include and $(LIBDIR) is $(INSTALLROOT)/lib.

The .SOURCE.h Special Atom informs nmake where header files are located, specifically, those files that contain .h suffixes. The FILES variable contains a list of files that are to be used in the build. The HEADERS variable contains the header file that is to be used in the build. The source dependency assertion builds the libsub.a library target out of the prerequisite files specified in $FILES. When this example was documented, $HOME was /usrs1/bob.

Contents of Makefile

```
.SOURCE.h : inc
FILES = sub1.c sub2.c sub3.c
HEADERS = headerX.h

$(INCLUDEDIR) :INSTALLDIR: $(HEADERS)
libsub.a :: $(FILES)
```

Run nmake

```
$ nmake install
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -Iinc \ -I- -c sub1.c
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- \ -c sub2.c
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- \ -c sub3.c
+ ar r libsub.a sub1.o sub2.o sub3.o
ar: creating libsub.a
+ ignore ranlib libsub.a
+ rm -f sub1.o sub2.o sub3.o
+ mkdir -p /usrs1/bob/include
+ 2> /dev/null
+ ignore cp inc/headerX.h /usrs1/bob/include/headerX.h
+ mkdir -p /usrs1/bob/lib
+ 2> /dev/null
+ ignore cp libsub.a /usrs1/bob/lib/libsub.a
```

In the following example, the assertion installs cmda to $(BINDIR) and then changes the user ID to root, the group ID to adm, and the mode to 0750.

Contents of Makefile

```
$(BINDIR) :INSTALLDIR: user=root\
group=adm mode=0750 cmda
```

Run nmake

```
$ nmake install
```

Output

```
+ ignore cp cmda /home/u1/uname1/bin/cmda
+ chgrp adm /home/u1/uname1/bin/cmda
+ true
+ chmod 0750 /home/u1/uname1/bin/cmda
+ chown root /home/u1/uname1/bin/cmda
```

The following example demonstrates the creation of directories at several depths.

Contents of Makefile

```
$(INCLUDEDIR)/subA/suba :INSTALLDIR: myfile
```

Results

Assuming they do not already exist, nmake will create the following directories:

```
$(INCLUDEDIR)
$(INCLUDEDIR)/subA
$(INCLUDEDIR)/subA/suba
```

Output

```
+ mkdir -p /home/u1/uname1/include/subA/suba
+ 2> /dev/null
+ ignore cp myfile /home/u1/uname1/include/subA/suba/myfile
```

Related Assertion Operators

:INSTALL:, :INSTALLMAP:

:INSTALLMAP: Assertion Operator

:INSTALLMAP: - install selected files in directory

Description

The :INSTALLMAP: assertion operator can be used for specifying installation directories for all files that match an edit operation. The edit operation is applied to the set of all known atoms. The syntax for this assertion operator is:

directory :INSTALLMAP: *edit_operation*

The files that match the edit operation specified on the rhs are installed in the directory specified on the lhs. For example,

$(FORMDIR) :INSTALLMAP: A=.SQLFORM:T=G

installs in $(FORMDIR) all generated files with the user-defined attribute .SQLFORM.

Because this assertion operator defers evaluation of the edit operation until the install target is being made, it is useful for installing files that are not known about until run time (i.e., implicit prerequisites).

Example

In the following example, the first line in the makefile defines .MAP as an attribute. The next two lines assign the .MAP attribute to the a.c and b.c files. A list of files is assigned to the FILES variable. In the assertion, all the file names that contain the characters a–z and have the .MAP attribute are copied to the bin directory. Note in the output that the bin directory did not exist and therefore was made before the files were copied.

Contents of Makefile

```
.MAP: .ATTRIBUTE

a.c : .MAP
b.c : .MAP

FILES = a.c b.c main.c x.c

bin :INSTALLMAP: A=.MAP:N=[a-z]*.c
```

Run nmake

```
$ nmake install
```

Output

```
+ mkdir -p bin
+ 2> /dev/null
+ ignore cp b.c bin/b.c
+ ignore cp a.c bin/a.c
```

Related Assertion Operators

:INSTALL:, :INSTALLDIR:

:INSTALLPROTO: Assertion Operator

:INSTALLPROTO: - install proto output files

Description

This assertion operator enables $(PROTO) output of rhs files into the lhs directory by the install common action.

$(PROTO) is a conversion program that converts prototyped C files to files compatible with the ANSI, K&R, and C++ dialects of C. Each file to be converted must contain #pragma prototyped as the first line and be in valid ANSI C.

PROTO is set to proto in the Alcatel-Lucent nmake bin directory by default.

Example

The following example shows how file.c is converted to a compatible C file and installed in directory protodir.

Contents of Makefile

```
.SOURCE: src
protodir :INSTALLPROTO: file.c
```

Run nmake

```
$ nmake install
```

Output

```
+ mkdir -p protodir
+ 2> /dev/null
+ proto -p -s -cAlcatel-Lucent Bell Labs src/file.c
+ 1> 1.00128131.x
+ cmp -s protodir/file.c 1.00128131.x
+ mv 1.00128131.x protodir/file.c
```

:JOINT: Assertion Operator

:JOINT: - build targets jointly

Description

The :JOINT: assertion operator jointly builds all targets on the lhs with respect to the prerequisites on the rhs. :JOINT: is not needed for metarules.

Example

In the following example, g.c, c_g.c, and g.h are jointly built from g.sch using the :JOINT: assertion operator. Then the generated C files are used to build the library liba.a. The action block of the :JOINT: assertion is a set of shell commands to generate lhs targets using the rhs source.

Contents of Makefile

```
liba.a :: g.c c_g.c

g.c c_g.c g.h :JOINT: g.sch .TERMINAL
        cat $(>) > $(>:B:S=.c)
        echo "#include '$(>:B).h'" > \
            c_$(>:B:S=.c)
        cat $(>) >> c_$(>:B:S=.c)
        echo "#define $(>:B)_x 1" > $(>:B:S=.h)
```

If a metarule is desired in this case, :JOINT: is not needed. For example,

```
%.c c_%.c %.h : %.sch
        shell actions
liba.a :: g.sch
```

can be used instead.

Run nmake

```
$ nmake
```

Output

```
+ cat g.sch
+ 1> g.c
+ echo #include "g.h"
+ 1> c_g.c
+ cat g.sch
+ 1>> c_g.c
+ echo #define g_x 1
+ 1> g.h
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c g.c
```

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I. -I- -c c_g.c
+ ar r liba.a g.o c_g.o
ar: creating liba.a
+ ignore ranlib liba.a
+ rm -f g.o c_g.o
```

:LIBRARY: Assertion Operator

:LIBRARY: - create library from source files

Description

This assertion operator provides an alternate, portable way of creating a library (shared or archive) from specified source files.

The syntax is:

name [*major.minor*] :LIBRARY: *source* [*required-libs*]

where *name* is the name of the library (e.g., x for libx.a), *major.minor* is the version number for shared libraries (if supported), *required-libs* are the libraries that must be linked, if they exist, in conjunction with the current target library when the library is in use, and *source* is the list of source files. The file *name* containing the information on the required libraries is generated and installed in $(LIBDIR)/lib and is used to include the *required-libs* when linking applications with -l*name*.

required-libs must be specified as -l*library_name*.

The following should be noted about the :LIBRARY: operator:

- The $(CCFLAGS) variable determines the name of the library archive. If $(CCFLAGS) is set to generate code for symbolic debugging (e.g., -g), the library is named *libname*-g.a (e.g., libx-g.a). If $(CCFLAGS) is set to generate code for profiling (e.g., -p), the library is named *libname*-p.a (e.g., libx-p.a).
- If the install action is requested the archive and shared libraries are installed in $(LIBDIR). On win32 the .lib is installed in $(LIBDIR) and the .dll is installed in $(BINDIR).
- The shared version is built **only if** $(CCFLAGS) is set to generate position-independent code (e.g., -pic) AND either the makefile contains the :ALL: assertion operator or the install common action is invoked on the command line (e.g. $ nmake install). Otherwise, only the archive version is created. However, as a side-effect, the archive is created and used in generating the shared version. Note, CCFLAGS+=$$(CC.PIC) can be used for portability to add the appropriate position independent code flag for the current compiler to CCFLAGS. (CC.PIC is defined by probe.)
- The default behavior in creating a shared library is to create one with the specified major.minor version number (when not specified it defaults to *name*.1.0.$(CC.SUFFIX.SHARED) and *name*.1.0.$(CC.SUFFIX.DYNAMIC) on win32, and lib*name*.$(CC.SUFFIX.SHARED).1.0 on other platforms). If the install action is also requested, on win32 the .lib library is installed

without the version string, on other platforms the library is installed and symbolically linked to one without the version string, lib*name*.$(CC.SUFFIX.SHARED). However, this default behavior may be modified with the following base rule variables:

   - **sharedliblinks** -- This variable controls the creation of the symbolic links during the installation of the target library. By default, it is set to "1" which will cause the links to be made as usual. If set to "0", then the generation of the symbolic link is turned off.
   - **sharedlibvers** -- This variable controls the creation of a shared library with a major.minor version number. By default, this is set to "1" which will cause a shared library with a version number to be created, as usual. However, when set to "0", even if the *major.minor* string is specified in the assertion, a shared library with a version number will NOT be created.

- It can be used to forge a dependency between the target library and the list of required prerequisite libraries specified as -l*library_name*. Since system libraries vary on different platforms only the required libraries that exist are tracked, others are ignored. This allows libraries for all platforms to be specified but only the libraries that exist on the current platform will be included when linking with the target library.

  For example, to build a library libcs.a from source.c that should be used in conjunction with libnsl.a and libsocket.a, one would specify:

  cs 2.0 :LIBRARY: source.c -lnsl -lsocket.

  In addition to generating the target library, nmake will generate the file cs that contains the required library information -lcs -lnsl -lsocket (or whichever of -lnsl and -lsocket exist on the current platform) in the $(LIBDIR)/lib directory when the install common action is used. When -lcs is used as a prerequisite library in a makefile, nmake automatically expands it to -lcs -lnsl -lsocket.

Example

The following is a simple makefile that generates a shared library named libdb.so.1.0. To make the Makefile portable, the flag used to generate position-independent code was referenced from the probe variable CC.PIC. See Appendix A of the *Alcatel-Lucent nmake Product Builder User's Guide* for details.

NOTE:
To append to CCFLAGS, the CCFLAGS line in the following makefile would have to read CCFLAGS += $$(CC.PIC).

Contents of Makefile

```
.SOURCE : src
INSTALLROOT = .
CCFLAGS = $(CC.PIC)

:ALL:
db 1.0 :LIBRARY: a.c b.c
```

Run nmake

```
$ nmake
```

Output

```
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/a.c
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/b.c
+ ar r libdb.a a.o b.o
ar: creating libdb.a
+ rm -f a.o b.o
+ nm -p libdb.a
+ sed -e /[ ][TDBC][ ][         ]*[_A-Za-z]/!d -e s/.*[ ][TDBC][ ][ ]*// -e /_STUB_/d -e s/^/-u /
+ ld -G -o libdb.so.1.0 -u a -u b libdb.a
```

Re-run nmake (after clobbering), this time installing the shared library and using the sharedliblinks variable

```
$ nmake sharedliblinks=0 install
```

Output

```
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/a.c
make: warning: a.o file system time lags local time by at least 14.00s
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/b.c
+ ar r libdb.a a.o b.o
ar: creating libdb.a
+ rm -f a.o b.o
+ nm -p libdb.a
+ sed -e /[ ][TDBC][ ][         ]*[_A-Za-z]/!d -e s/.*[ ][TDBC][ ][ ]*// -e /_STUB_/d -e s/^/-u /
+ ld -G -o libdb.so.1.0 -u a -u b libdb.a
+ mkdir -p lib
+ 2> /dev/null
+ cp libdb.a lib/libdb.a
+ cp libdb.so.1.0 lib/libdb.so.1.0
```

Related Assertion Operators

::

:LINK: Assertion Operator

:LINK: - create hard/symbolic link to rhs file

Description

The lhs files are symbolic/hard links to the rhs file. If the base rule variable, symbolic_link is zero or null, an hard link is created, otherwise, a symbolic link is attempted. This assertion operator works in conjunction with the install common action.

rhs may be a file name, or a variable expansion evaluating to a file name. rhs is evaluated separately for each element of lhs. During each expansion $$(<) evalates to the current target.

Examples

In the following example, the assertion creates x, which is a hard link to y, for use by the install common action.

Contents of Makefile

```
x :LINK: y
```

Run nmake

```
$ nmake
```

Output

```
+ ln y x
```

In this example, the rhs is evaluated separately for each target.

Contents of Makefile

```
LINKS = link1.suf1 link1.suf2 link2.suf1 link2.suf2

:ALL: $(LINKS)
$(LINKS) :LINK: $$(<:B=orig:S)
```

Run nmake

```
$ nmake install
```

Output

```
+ ln orig.suf1 link1.suf1
+ ln orig.suf2 link1.suf2
+ ln orig.suf1 link2.suf1
+ ln orig.suf2 link2.suf2
```

:MAKE: Assertion Operator

:MAKE: - run nmake on rhs makefiles

Description

Tokens on the rhs can be subdirectories or a series of makefiles in the current directory or a shell pattern or a dash (–). Subdirectory names can contain path name prefixes, as in the following example:

```
:MAKE: ../d1 /actual/path/d2 d3/a file.mk
```

If the rhs is a subdirectory, a recursive nmake is run in that directory. If the rhs is not specified, a recursive nmake is run in all subdirectories containing a makefile from the list $(MAKEFILES:/:/ /G).

Targets may also be specified on the lhs and built recursively by asserting recurse before them to force recursion.

With :MAKE:, the following common actions are recursive by default: all, ciadag, ciadb, and install.
However, other common actions have to assert recurse before them to force recursion, for example,

```
$ nmake recurse clobber
```

Example

In the example that follows, the :MAKE: assertion 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 (–) (see the manual page on the – Special Atom) is used to synchronize processing; in other words, nmake must complete any processing in the lib directory before processing the makefile contained in the cmd directory.

Contents of Makefile

```
:MAKE: lib - cmd
```

Shell patterns can be used to specify the list of directories in which nmake should be run. By default, the rhs defaults to * (all subdirectories).

In the following example, nmake is recursively run in the libx directory. When nmake finishes processing in the libx directory, all of the remaining subdirectories are made.

Contents of Makefile

```
:MAKE: libx – *
```

As an example of a more complex selection process, consider the following example, which uses ksh pattern-matching facilities to include all directories except home, make, and nmake.

Contents of Makefile

```
:MAKE: probe - ccc cpp - !(home|?(n)make)
```

The following example demonstrates how to specify a lhs target and then recursively build it. Here, **targ1** will be built in directories a,b and c while directories d,e and f will not be visited.

Contents of Makefile

```
targ1 :MAKE: a b c
targ2 :MAKE: d e f
```

Run nmake

```
$ nmake recurse targ1
```

:PACKAGE: Assertion Operator

:PACKAGE: - find files and libraries in software package

Description

This assertion operator facilitates locating include files and libraries of a specified software package. With its associated environment variables, :PACKAGE: reduces the need for many .SOURCE.h and .SOURCE.a assertions. When :PACKAGE: is used, by default nmake searches for header and library files in directories respectively called include and lib under /usr/add-on/*package*, /usr/local/*package*, and /home/*package*, and adds -l*package* to LDLIBRARIES to link with :: executable targets. *package* is the name of the software package.

The syntax is:

:PACKAGE:[+-] *package*

When :PACKAGE: is used, INCLUDEDIR becomes $(INSTALLROOT)/include/*package* unless the optional - is specified.

:PACKAGE: can be defined more than once; the most recent definition takes precedence unless the optional + is specified, in which case it takes lowest precedence.

If there is no lib*package* library, use .DONTCARE to prevent nmake from complaining:

-l*package* : .DONTCARE

The following environment variables are used by :PACKAGE::

PACKAGE_*package*	alternate home for *package* if it is not installed in one of the three default directories
PACKAGE_*package*_INCLUDE	appends directories to .SOURCE.h ; needed if include files are not in one of the three default directories
PACKAGE_*package*_LIB	appends directories to .SOURCE.a ; needed if library files are not in one of the three default directories.
_PACKAGE_*package*	state variable defined to 1 by :PACKAGE:

Example

Let’s assume that we have installed a software package called ast. And suppose that we define the location of its executable (if it is not in a standard directory), its header files, and its library files, as follows:

```
export PACKAGE_ast=/home/build/ast
export PACKAGE_ast_INCLUDE=/home/build/ast/include
export PACKAGE_ast_LIB=/home/build/ast/lib
```

Contents of Makefile

```
:PACKAGE: ast
:PACKAGE: xast yast +
```

Output

When the makefile is processed, nmake looks for headers and libraries first in the ast package, then in the xast and yast packages.

:READONLY: Assertion Operator

:READONLY: - place tables and data in read-only text object section

Description

The rhs compilation is to place tables and/or data in a read-only text object section. Only the parse and pattern tables are placed in read-only for yacc(1) and lex(1) files.

Example

In the following example, y.c has only data. The :READONLY: assertion operator is used to put the data in a read-only text object section during the compilation of y.c.

Contents of y.c

```
int x;
```

Contents of Makefile

```
prog :: x.c y.c
:READONLY: y.c
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c x.c
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -R -c y.c
+ cc -O -o prog x.o y.o
```

:SAVE: Assertion Operator

:SAVE: - save rhs files

Description

The rhs files are generated but are still to be saved by the save-oriented common actions (cpio, pax, etc.).

Example

In the following example, nmake is run twice, once with and once without the :SAVE: assertion operator. Notice that x is saved in the cpio file when the :SAVE: assertion operator is specified in the makefile.

Contents of Makefile

```
x :: x.c
:SAVE: x
```

Run nmake

```
$ nmake cpio
```

Output (With the Line: :SAVE: x)

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c x.c
+ cc -O -o x x.o
+ cpio -o
+ echo x x.c Makefile
+ tr \012
+ 1> x.cpio
49 blocks
```

Comment Out Line

```
/* :SAVE: x */
```

Run nmake

```
$ nmake cpio
```

Output (Without the Line: :SAVE: x)

```
+ cpio -o
+ echo x.c Makefile
+ tr   \012
+ 1 x.cpio
1 block
```

:cc: Assertion Operator

:cc: - compile rhs files using $(cc)

Description

The list of files on the rhs are compiled using $(cc) instead of $(CC). $(cc) defaults to cc. This assertion operator is commonly used for compiling C and C++ files from the same makefile.

Example

In the following example, the same makefile is used for compiling C and C++ files from the same makefile.

The target program is built from the C source file called ccfile.c and the C++ source file called CCfile.c. The C source file ccfile.c is compiled using $(cc); the C++ source file CCfile.c is compiled using $(CC).

Contents of Makefile

```
CC = CC
cc = cc     /* default */

CPLUSFILES = CCfile.c
CFILES     = ccfile.c
program :: $(CFILES) $(CPLUSFILES)
:cc: $(CFILES)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\lib/probe/C/ \
pp/361FBDCFrucbcc -I- \-c ccfile.c
+ CC -O -I-D/nmake3.0/lib/probe/C/pp/A9E7CCBE.0.2CC -I- -c CCfile.c
+ cppC=/nmake3.0/lib/cpp
CC CCfile.c:
cc -c -O -I-D/nmake3.0/lib/probe/C/pp/A9E7CCBE.0.2CC -I- CCfile.c
+ CC -O -I- -o program ccfile.o CCfile.o
+ cppC=/nmake3.0/lib/cpp
cc -L/chome/sun4/3.0.2 -o program -O -I- ccfile.o CCfile.o -lC
```

Variables

4

This section contains manual pages for the Assignment Operators, the Automatic Variables, and the Base Rule Variables, and tables listing the Directory Variables, the Engine Environment Variables, and the Uninitialized Environment Variables. It also contains tables listing the Base Rule Variables that can be used in action blocks and the State Variables defined in the default base rules.

See also the *Alcatel-Lucent nmake User's Guide* Chapter 3, *Using Variables*, for more information on variables, including variable expansion, precedence and scope.

= Assignment Operator

= **-** replace variable by value

Synopsis

variable = value

Description

The = assignment operator replaces *variable* by its *value*. Variables on the rhs are expanded when the variable on the lhs is evaluated.

Example

```
X = 100
VALUE = $(X)
```

X is assigned the string 100. VALUE is assigned the string $(X). The value stored in X is expanded when $(VALUE) is encountered.

:= Assignment Operator

:= - expand value but don’t change variable

Synopsis

variable* := *value

Description

The := assignment operator expands *value* (if it contains another variable) at the time of assignment. Changes to the *value* variable do not affect *variable*.

Example

```
X=100
VALUE:=$(X)
```

X is assigned the string 100. VALUE is assigned the string 100.

+= Assignment Operator

+= - append values to current value list for variable

Synopsis

variable* += *value

Description

The += assignment operator appends *value* to the existing value of the variable. If value contains a variable, that variable is expanded and appended to the value of the variable.

Example

```
FILES = a.c b.c
FILES += x.c
```

$(FILES) is expanded to a.c b.c x.c. A space is inserted between the appended value and the original.

== Assignment Operator

== - declare value to be implicit state variable

Synopsis

variable == value

Description

The == assignment operator is the same as the = assignment operator, except that it declares the variable to be a candidate implicit state variable.

Example

Contents of Makefile

```
MESSAGE == "Hello World"
hello :: hello.c
```

Program hello.c

```
main()
{
printf("%s\n", MESSAGE);
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/lib/\
probe/C/pp/B49DF4E0.bincc -I- \
-DMESSAGE="Hello World" -c hello.c
+ cc -O -o hello hello.o
```

Note that the -DMESSAGE="Hello World" argument is automatically generated. This is an example of candidate implicit state variables. nmake automatically sets up a dependency between the source file and any of the current implicit state variables that it uses. In addition, nmake adds the -D arguments for each one used by the source module.

&= Assignment Operator

&= - add auxiliary value to a variable

Synopsis

variable* &= *value

Description

The &= assignment operator is used to add an auxiliary value to a variable. The auxiliary value is not saved in the statefile.

Example

In the following example:

```
X = dummy
X &= dummy2
```

dummy2 is the auxiliary value and dummy is the primary value. The auxiliary value is not saved in the statefile, nor is it saved from one assignment to another.

In the following example,

```
X = dummy
X &= dummy2
X &= dummy3
```

dummy3 is the only auxiliary value after the last assignment.

$(!) Automatic Variable

$(!) - all implicit and explicit file prerequisites of current target

Description

This variable is expanded to the list of all implicit and explicit file prerequisites of the current target. Implicit file prerequisites are generated by the nmake language-dependent scanning capability.

Example

In this example, hello.c includes stdio.h

```
hello :: hello.c
.DONE :
        : $(!hello)
```

In the example above, $(!hello) expands to hello.o hello.c /usr/include/stdio.h.

$(#) Automatic Variable

$(#) - number of actual arguments

Description

This variable is expanded to the number of actual arguments in the following constructs:

```
local -[n] arg ...
local (formal ...) actual
```

Example

The following makefile will return the number of actual arguments, i.e., 3 and 2. This can be used to get local function arguments (a b c) and set their values from the actual arguments in $(%).

Contents of Makefile

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

F2 : .FUNCTION
        local - a b
        print $(#)

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

$(%) Automatic Variable

$(%) - stem of current default base rule match

Description

This variable has four distinct meanings depending on context:

- It expands to the *stem* of the current default base rule match.
- It expands to targets for non-default base rule targets.
- It replaces the .SCAN. atom in release 2.2.
- It represents the arguments of a functional variable.

Example

In this example, the stem of x.o is x; $(%) expands to x. Thus, $(>) has the correct value, x.c.

Contents of Makefile

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

Run nmake

```
$ nmake
```

Output

```
+ : x
+ cc -O -Qpath /nmake3.3/lib -I_D/nmake3.3/lib/\
probe/C/pp/361FBICFrucbcc -I- -c x.c
```

$(&) Automatic Variable

$(&) - all implicit and explicit state variable prerequisites of current target

Description

This variable is expanded to the list of all implicit and explicit state variable prerequisites of the current target. Implicit state variable prerequisites are generated by the nmake language-dependent scanning capability.

Example:

Files in directory

```
1.c x.c y.c z.c
```

Contents of Makefile

```
FILES = y.c z.c
targ : 1.c (FILES) (CC) (CCFLAGS)
        : "$(&)"
```

Run nmake

```
$ nmake
```

Output

```
+ : (FILES) (CC) (CCFLAGS)
```

The quotation marks are needed around $(&) to prevent shell interpretation of its value.

$(–) Automatic Variable

$(-) - list of options for use with -o format

Description

There are three formats that can be used to specify options to nmake. For example, the keepgoing option can be specified by using the -o keepgoing format or -k on the command-line or by using set keepgoing in the makefile.

This variable expands to a list of the option names suitable for use with the -o format. Only those option settings different from the default are used.

Example

Contents of Makefile

```
target :
        : $(-)
```

Run nmake

```
$ nmake -f makefile -k
```

Output

```
+ : -o keepgoing
```

$(*) Automatic Variable

$(*) - all explicit file prerequisites of current target

Description

This variable is expanded to list all explicit file prerequisites of the current target.

Example

```
FILES = b.c c.c
product : a.c $(FILES) (CC) (CCFLAGS)
        $(CC) $(CCFLAGS) -o $(<) $(*)
```

$(*) expands to a.c b.c c.c.

$(+) Automatic Variable

$(+) - list of options for use with set format

Description

There are three formats that can be used to specify options to nmake. For example, the keepgoing option can be specified by using -o keepgoing or -k on the command-line or by using set keepgoing in the makefile.

This variable expands to a list of option names suitable for use with the set programming statement. Only those option settings different from the default are used.

Example

Contents of Makefile

```
target :
        : \$(+) "$(+)"
        : \$(-) "$(-)"
        : \$(-k) "$(-k)"
        : \$(+k) "$(+k)"
        : \$(-keepgoing) "$(-keepgoing)"
        : \$(+keepgoing) "$(+keepgoing)"
```

Run nmake

```
$ nmake -k
```

Output

```
+ : $(+) "keepgoing"
+ : $(-) "-o keepgoing "
+ : $(-k) "1"
+ : $(+k) "keepgoing"
+ : $(-keepgoing) "1"
+ : $(+keepgoing) "keepgoing"
```

Run nmake again

```
$ nmake +k
```

Output

```
+ : $(+) "nokeepgoing"
+ : $(-) "-o nokeepgoing "
+ : $(-k) ""
+ : $(+k) "nokeepgoing"
+ : $(-keepgoing) ""
+ : $(+keepgoing) "nokeepgoing"
```

Run nmake a third time

```
$ nmake -k -d
```

Output

```
+ : $(+) "nodebug keepgoing"
+ : $(-) "-o nodebug keepgoing "
+ : $(-k) "1"
+ : $(+k) "keepgoing"
+ : echo $(-keepgoing) "1"
+ : $(+keepgoing) "keepgoing"
```

$(+*option*) Automatic Variable

$(+*option*) - current setting for *option*

Description

This variable expands to the current setting for the named options, suitable for use with the set programming statement .

Example

Contents of Makefile

```
.INIT: .MAKE
  if "$(+keepgoing)" == "keepgoing"
              error 0 You have set the\
              keepgoing option.
 end
tst:
        : $(+keepgoing)
```

Run nmake

```
$ nmake -k
```

Output

```
You have set the keepgoing option.
+ : keepgoing
```

Run nmake again

```
$ nmake
```

Output

```
+ : nokeepgoing
```

In the example above, $(+keepgoing) expands to keepgoing when the option is specified and nokeepgoing when it is not specified.

$(...) Automatic Variable

$(...) - all atoms, rules, and state variables used when making .MAKEINIT

Description

This variable represents all the atoms, rules, and state variables used when making .MAKEINIT. Edit operators can be used to select tokens of interest. For example, $(...:A=.REGULAR) gives the existing files used or generated by nmake.

Example

Contents of Makefile

```
hello :: hello.c
.DONE : .QUERY
/* .QUERY puts user in query mode */
```

Contents of hello.c

```
main()
{
printf("Hello World\n");
}
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/B49DF4E0.bincc -I- \
-c hello.c
+ cc -O -o hello hello.o
make>
```

Entering $(...) at the make> prompt, causes all the atoms, rules, and state variables to be displayed on the screen.

Enter $(...:N=hello*) to see atoms related to hello.

```
make> $(...:N=hello*)

hello : [Jan 13 10:22:11 1998] must=2 \
command target compiled regular \
select0 triggered EXISTS
 prerequisites: hello.o .COMMAND.o \
(CCLD) (CCFLAGS) (F77FLAGS) (LDFLAGS)
```

```
()hello : [Jan 13 10:22:14 1998] \
event=[Jan 13 10:22:14 1998] force \
built compiled
 prerequisites: hello.o .COMMAND.o \
(CCLD) (CCFLAGS) (F77FLAGS) (LDFLAGS)

hello.c : [Jan 13 09:32:03 1998] .SCAN.c \
terminal compiled regular scanned \
select0 EXISTS

()hello.c : [Jan 13 09:31:55 1998] .SCAN.c \
event=[Jan 13 09:32:03 1998] compiled \
scanned

hello.o : [Jan 13 10:22:04 1998] .OBJECT \
must=2 implicit target compiled \
regular select0 triggered EXISTS
 prerequisites: hello.c (CC) (CCFLAGS)

()hello.o : [Jan 13 10:22:11 1998] .OBJECT \
event=[Jan 13 10:22:11 1998] force \
built compiled
 prerequisites: hello.c (CC) (CCFLAGS)
```

Exit query mode

```
make> q
```

$(;) Automatic Variable

$(;) - state variable value or unbound atom name

Description

This variable is expanded to the state variable value if the current target is a state variable; otherwise, it is expanded to the unbound atom name.

Example

In this example, the file x.c is copied from the current directory to a sibling directory, dir. The $(;) automatic variable in this example expands to the value of cpdir, which is dir; this variable does not expand, however, if there is no prerequisite (in this case if $(Files) does not exist).

Contents of Makefile

```
Files = x.c
cpdir = dir
all : (cpdir)
(cpdir) : $(Files)
        cp $(>) ../$(;)
```

Run nmake

```
$ nmake
```

Output

```
+ cp x.c ../dir
```

$(<) Automatic Variable

$(<) - current target name

Description

This variable is expanded to the current target name.

Example

In the following example, $(<) expands to product.

Contents of Makefile

```
FILES = a b c
product : $(FILES)
        cat $(*) > $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ cat a b c
+ 1> product
```

The next example shows that applying $(<) to a joint target will return all the joint target siblings of the specified target.

Contents of Makefile

```
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
```

Run nmake

```
$ nmake
```

Output

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

$(=) Automatic Variable

$(=) - all command-line script arguments and assignments for a variable

Description

This variable is expanded to the list of command-line script arguments and assignments for variables that are prerequisites of the .EXPORT atom.

Example

```
FILE = file1
.EXPORT : FILE
target  :
        : $(=)
```

In this example, $(=) expands to FILE=file1.

$(>) Automatic Variable

$(>) - all out-of-date or new explicit file prerequisites of the current target.
- primary metarule prerequisite.

Description

This variable has different meanings for metarules and regular rules.

In regular rules this variable is expanded to include all explicit file prerequisites of the current target that have a later date than the target or are new prerequisites. It may be null even if the current target action is triggered. For example, if the target is touched, bringing its time stamp out-of-date, then its action will be triggered but the file prerequisites will not have a later date than the target and the variable will be null.

In metarules this variable is always the first %-pattern prerequisite, also called the primary metarule prerequisite, and is never null.

Example

Makefile with a regular rule

```
libbc : b.c c.c
        $(CC) $(CCFLAGS) -c $(>)
        $(AR) $(ARFLAGS) $(<) $(>:B:S=.o)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -I- -c b.c c.c
b.c:
c.c:
+ ar r libbc b.o c.o
ar: creating libbc
```

$(>) expands to b.c and c.c

Touch b.c

```
touch b.c
```

Run nmake again

```
$ nmake
```

Output

```
+ cc -O -I- -c b.c
+ ar r libbc b.o
```

$(>) expands to b.c

Add d.c prerequisite to the Makefile

```
libbc : b.c c.c d.c
        $(CC) $(CCFLAGS) -c $(>)
        $(AR) $(ARFLAGS) $(<) $(>:B:S=.o)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -I- -c d.c
+ ar r libbc d.o
```

$(>) expands to d.c

Touch the target

```
touch libbc
```

Run nmake again

```
$ nmake
```

Output

```
+ cc -O -I- -c
usage: cc [ options] files.  Use 'cc -flags' for details
make: *** exit code 1 making libbc
```

$(>) expands to null

Makefile with a metarule

```
%.xyz : file.template %.x
        mycmd -t $(*:N=*.template) -f $(>) -o $(<)

:ALL: abc.xyz
```

Contents of directory

```
Makefile  abc.x  file.template
```

Run nmake

```
$ nmake
```

Output

```
+ mycmd -t file.template -f abc.x -o abc.xyz
```

$(>) expands to abc.x

$(?) Automatic Variable

$(?) - all prerequisites of the current target

Description

This variable is translated to the list of all prerequisites of the current target. All implicit and explicit state and file prerequisites are included.

This variable is the equivalent of the variables $(!) and $(&) together.

Example

Contents of Makefile

```
FILE = file1 /* hello.c includes stdio.h */
NOT_USED == file2

target : hello.c (FILE)
        : "All state prereqs:" "$(&)"
        : "All file prereqs:" "$(!)"
        : "All prereqs:" "$(?)"
```

Run nmake

```
$ nmake
```

Output

```
+ : All state prereqs: (FILE)
+ : All file prereqs: hello.c /usr/include/stdio.h
+ : All prereqs: hello.c /usr/include/stdio.h (FILE)
```

$(@) Automatic Variable

$(@) - action for the current target

Description

This variable is expanded to the action for the current target.

The :JOINT: definition in Makerules.mk is used to promote the use of high-level assertion operators instead of the lower-level .JOINT Special Atom on the assertions. The definition is listed below:

```
":JOINT:" : .MAKE .OPERATOR
        eval
        $$(<) : .JOINT $$(>)
                                $(@:V)
        end
```

When the :JOINT: assertion operator is used in the makefile as below:

```
g.c g.h :JOINT: g.shc
        $(CP) $(>) $(>:B:S=.c)
        $(CP) $(>) $(>:B:S=.h)
```

This assertion is translated to the following by nmake:

```
g.c g.h : .JOINT g.sch
        $(CP) $(>) $(>:B:S=.c)
        $(CP) $(>) $(>:B:S=.h)
```

since $(@) is used in the :JOINT: definition.

$(^) Automatic Variable

$(^) - current target down the viewpath

Description

The expansion of this variable depends on the state of the current target.

— If the current target binds to a file down the viewpath, $(^) expands to the full path of that file (at the lower view).

— If not, $(^) expands to *null*

The $(^) automatic variable is used in the definition of the .ARCHIVE.o rule in the Makerules.mk to copy an archive from down the viewpath, if it exists, for maintenance sake (such as adding to or removing members from it as the case may be).

$(~) Automatic Variable

$(~) - explicit prerequisites of the current target

Description

This variable is expanded to the list of all explicit prerequisites of the current target.

Example

In the following makefile, quotation marks are needed around $(~) to prevent shell interpretation of its value.

Files in directory

```
1.c x.c y.c z.c
```

Makefile

```
FILES = y.c z.c

targ : 1.c (FILES) (CC) (CCFLAGS)
        : "$(~)"
```

Run nmake

```
$ nmake
```

Output

```
+ : 1.c (FILES) (CC) (CCFLAGS)
```

$(-*option*) Automatic Variable

$(-*option*) - 1 or null

Description

The expansion of this variable depends on whether *option* has been set:

- If *option* has been set and is not zero, this variable is expanded to 1.
- If *option* has not been set, this variable is expanded to null.

Example

In the following example, $(-keepgoing) expands to 1.

Contents of Makefile

```
target :
        : $(-keepgoing)
```

Run nmake

```
$ nmake -k
```

Output

```
+ : 1
```

ancestor Base Rule Variable

ancestor - specify number of levels to search

Description

This variable can be used to specify how many directory levels up from the current directory to look for include and library directories. The standard directory names include and lib are searched for at each level up from the current directory.

Instead of specifying:

```
.SOURCE.h : ../include ../../include
.SOURCE.a : ../lib ../../lib
```

the following can be used:

```
ancestor=2
```

ancestor_list Base Rule Variable

ancestor_list - list directories to search

Description

This variable must be used with the base rule variable ancestor. It is used to parameterize the lib and include directories used with ancestor. The default value includes the default search directory names, i.e.:

ancestor_list = $(ancestor_source) .SOURCE.a lib .SOURCE.h include

The default value of ancestor_source is null. ancestor_source can be used to add to the list of directories to be searched for files.

ancestor_list can be used to change the default search directory names. For example, the following defintion would cause the LIB directory rather than the lib directory to be searched for .a files and the hdrs directory rather than the .include directory to be searched for .h files:

ancestor_list = .SOURCE.a LIB .SOURCE.h hdrs

To add to the list of directories to be searched for files, use the ancestor_source variable instead.

ancestor_source Base Rule Variable

ancestor_source - add to list of directories to search

Description

This variable must be used with the base rule variable ancestor. It is used to parameterize lib and include directories used with ancestor. ancestor_source is used to add to the list of default directories that are to be searched for header (include for .h files) and library files (lib for .a files). The default value for ancestor_source is null.

For example, to have the LIB directory searched for .a files in addition to the default lib directory and the hdrs directory searched for .h files in addition to the default include directory, define ancestor_source in the following manner:

```
ancestor_source = .SOURCE.a LIB .SOURCE.h hdrs
```

Files are searched in the order of directories listed in ancestor_list. In the above example, if a library file called values.a appears in both the LIB and lib directories, the one that appears in the LIB directory is included by nmake.

arclean Base Rule Variable

arclean - clean up after archive target is made

Description

This variable is used to control the removal of the generated (e.g., .o) files of an archive target after the target is made. The value of arclean is a list of edit operations. Archive members matching the list of edit operations are removed after the archive target is made. For example, if one is making library libx.a from a list of C files and arclean=N!=x.o, all the .o files except x.o are removed after libx.a is made.

The default value of arclean is null.

ccase_audit, ccase_message

ccase_audit - ClearCase[1] clearaudit support
ccase_message - warning message

Description

When using Alcatel-Lucent nmake in a Rational ClearCase environment, ClearCase Configuration Records and Derived Objects can be produced by running shell actions through clearaudit. To use clearaudit set ccase_audit=1. A warning message is printed to verify clearaudit is being used. The message can be changed or eliminated by setting variable ccase_message. To use clearaudit a ClearCase view must be set and the working directory must be in a dynamic view.

The default value of ccase_audit is null.

The default value of ccase_message is "clearaudit is turned on".

1 Rational and ClearCase are registered trademarks of IBM

cctype Base Rule Variable

cctype - specify which probe configuration file to use

Description

$(CC) is normally used when determining which probe configuration file will be loaded by nmake or cpp. $(cctype) can be used to specify that another C compiler's probe configuration file is to be loaded instead. See Appendix A in the *Alcatel-Lucent nmake Product Builder User’s Guide* for more information on probe.

cleanignore Base Rule Variable

cleanignore - list files not to be deleted by clean

Description

This variable can be set to a list of files that are not to be deleted by the clean common action. Shell patterns for files are allowed. The default value of cleanignore is null.

clobber Base Rule Variable

clobber - remove all generated files

Description

If $(clobber) is non-null, the default install action does not move the old install target to *target*.old.

compare Base Rule Variable

compare - compare against target during install

Description

If $(compare) is 0, the default install action does not do a cmp -s before installing the target file. The default value is 1.

disable_probe_message Base Rule Variable

disable_probe_message - user defined warning message when using the disableautomaticprobe option

Description

The disable_probe_message variable is used in conjunction with the disableautomaticprobe option. When disableautomaticprobe is enabled and a probe needs to be triggered, the value of disable_probe_message is emitted as a warning or error message.

disable_probe_message is set as "*ERRORLEVEL custom_message*". If *ERRORLEVEL* is 2 or less then *custom_message* is issued as a warning and processing is continued. If *ERRORLEVEL* is greater than 2 then *custom_message* is issued as an error and nmake exits with an exit code of *ERRORLEVEL*-2.

The message can be suppressed by setting disable_probe_message to null.

The default value of disable_probe_message is "3 probe file non-existent or out-of-date - contact nmake administrator".

force_shared Base Rule Variable

force_shared - control treatment of shared libraries

Description

If force_shared = 1, out-of-date executables with shared libraries are relinked. If force_shared = 0 , shared library time stamps are ignored. This is the default.

implicit_include Base Rule Variable

implicit_include - set up dependencies for C++ implicit include files

Description

If implicit_include = 1, nmake will attempt to emulate commonly used C++ implicit template definition search rules and automatically set up the required implicit header file dependencies. However, complete emulation of compiler implicit search rules cannot be guaranteed. The only foolproof approach is suppression of the compiler implicit include feature and use of explicit #includes.

By default, dependencies for C++ implicit includes will *not* be set up.

implicit_template_definition_warning

implicit_template_definition_warning - warning for C++ implicit include

Description

Define warning to be printed when the C++ implicit include feature is in effect, and the compiler provides some way to suppress this feature. By default,

implicit_template_definition_warning = C++ implicit template definition enabled: nmake automatic dependency generation may not work (disable implicit template definition if necessary)

The warning may be suppressed by setting implicit_template_definition_warning to null.

This feature detects a potential problem involving C++ implicit template definition search procedures. Some C++ compilers (especially older compilers) have an option that gives the compiler permission to go looking for “implicit include” files in order to find template definitions. Since nmake automatic file dependency generation is based upon detection of explicit #include statements during a scan of the program source, implicitly included files may not be properly set up as implicitly determined prerequisites. In this case, updating these implicitly included files may not trigger the appropriate recompilations. The recommended solution is to disable implicit template definition searching and use explicit #include statements for all template definition header files. This approach is consistent with the recommended template file organization in recent major C++ compilers, known as “definitions included.”

instrument Base Rule Variable

instrument - specify software package name

Description

This variable currently supports the following software packages: insight, quantify, purecov, purify[1], sentinel[2], insure and codewizard. When instrument is set to one of these packages (e.g., instrument=sentinel), nmake builds targets with that package.

By default, nmake searches /usr/add-on, /usr/addon, and /usr/local for these packages. If they are installed elsewhere, the full path must be specified in the instrument setting (e.g., instrument=/tools/sentinel).

1 Purify is a trademark of Pure Software Inc.

2 Sentinel is a trademark of Virtual Technologies, Inc.

libopts_warning Base Rule Variable

libopts_warning - warning for compiler flags that may affect probe results

Description

Define warning to be printed when the compiler supports flags for incorporating compiler supplied libraries and such a flag is included in CCFLAGS. By default,

libopts_warning = 1 the following CCFLAGS may affect probe values and should be moved to CC

The warning may be suppressed by setting libopts_warning to null.

This feature detects compiler support for flags used to incorporate compiler supplied optional libraries, such as the Sun C++ -library and -compat flags. The flags may change where the compiler searches for libraries and/or header files. In order for nmake to be aware of the compiler's behavior these flags should be included in the CC variable after the compiler instead of CCFLAGS so the compiler is probed in the proper mode to generate the correct probe values.

link Base Rule Variable

link - force install to link rather than copy target

Description

For install targets where the basename matches the pattern in $(link), the install action will attempt to link instead of copy. The default is link = 0. Setting link = 1 will cause all install files to be linked.

In this example, the following makefile is used to build several programs.

```
hello :: hello.c
jello :: jello.c
```

When nmake is run with the following options,

```
$ nmake install compare=0 clobber=1 link=h*
```

nmake attempts to link hello and copy jello to the installation directory. The output from this command-line is shown below. Note that a compare was not done before the link/copy (compare=0) and that existing files in the installation directory were not saved as hello.old/jello.old (clobber=1).

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/lib/probe/\
C/pp/B49DF4E0.bincc -I- -c hello.c
+ cc -O -o hello hello.o
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/lib/probe/\
C/pp/B49DF4E0.bincc -I- -c jello.c
+ cc -O -o jello jello.o
+ mkdir -p /home/u1/bob/bin
+ 2> /dev/null
+ ln -s ../NMAKE3.0/TEST/hello /home/u1/bob/bin/hello
+ ignore cp jello /home/u1/bob/bin/jello
```

localprobe Base Rule Variable

localprobe - specify alternate probe hierarchy root

Description

When non-null, localprobe allows for two alternate hierarchy schemes:

1. A node relative to the current viewpath (VPATH)

2. A node within a colon-separated list of directories called the "probe path" (PROBEPATH)

If set to vpath, the first form is in effect, if set to a different non-null setting, the second form is implied. In appearance, the alternate probe hierarchies are similar to using C language with make and cpp, using the convention <root>/lib/probe/C/make|pp for the directory structure, and also the calling of an associated probe shell off of this root. But in this scheme, <root> can either be a node in the VPATH, or a node in the PROBEPATH.

When using local probe files, normal file creation/access permissions are in effect, so the real uid (and gid) of the underlying user will determine the owner and group of any created probe information file. The creation mode of local probe files is 0464. The creation of local probe information files may also require the creation of the probe directory hierarchy itself. If nmake is required to create directories in the probe directory hierarchy, a creation mode of 0775 will be assumed.

nativepp Base Rule Variable

nativepp - control preprocessor use

Description

The setting of the nativepp base rule variable determines whether the nmake preprocessor or the native C preprocessor will be used. The nmake preprocessor enables viewpath support via the -I- option. Some native C preprocessors do not support this option and therefore impact the setting of nativepp. The following table outlines the result of setting nativepp:

nativepp	**ACTION**
0	Use nmake cpp
1	Use the native cpp with a warning message if -I- is not supported
-1	Use the native cpp with NO warning message if -I- is not supported
null (default)	If compiler supports -I- option then use the native cpp else use the nmake cpp

output Base Rule Variable

output - override output file base name

Description

This variable can be used to override the default common action output file base name.

physical Base Rule Variable

physical - limit recursion in 3DFS file system

Description

This variable is related to the 3DFS file system. When physical is set, it limits the recursion to directories that physically exist.

prefixinclude Base Rule Variable

prefixinclude - look for nested include files using directory prefix

Description

This variable is used to support C #include "file.h" compatibility by looking for nested include files using the directory prefix used to include its parent. The default is prefixinclude = 1 (feature is enabled) except when using a native preprocessor that does not support -I-. With prefixinclude = 1 the following scenario is valid:

Contents of main.c

```
#include "dir/file1.h"
```

Contents of file1.h

```
#include "file2.h"
```

nmake first looks for dir/file2.h using the default search rules. If the file is not found, nmake searches for file2.h using the default search rules.

With prefixinclude = 0, nmake searches only for file2.h using the default search rules.

When the nmake preprocessor is used, it is configured to also follow the search rules specified by prefixinclude.

preserve Base Rule Variable

preserve - save previous install targets using inode naming convention

Description

When the install common action is used and preserve = 1, a copy of each installed target is saved as *file#inode* on subsequent install actions, where *file* is the name of the target and *inode* is the inode number of *file*. When preserve = *file_list*, each file in *file_list* is saved as *file#inode*; all others are saved as the default *file*.old on subsequent install actions. *file_list* is the list of files to be preserved or the list of relative paths to the installed files; if no path is specified, the standard install areas are checked. If multiple files are specified on the command-line, they must be enclosed in double quotes. File match patterns are accepted, e.g., preserve = t*.

Since inodes are unique, all preserved versions are maintained; *file*.old is overwritten each time *file* is installed.

The default value of preserve is null.

quoteinclude Base Rule Variable

quoteinclude - issue warning or error for #include "..." directives

Description

This feature is useful for projects using compilation tools whose native preprocessor does not support the -I- option, and that for some reason cannot preprocess using the nmake preprocessor. Using quoteinclude, a project can effectively enforce the exclusive use of #include <...> directives, which do not require -I- for viewpath support. This allows projects to make full use of viewpathing, while using compilation tool chains which do not support -I-. The usage is as follows.

quoteinclude = 1|2|3 [prefix]

If the VPATH contains more than 1 node and the first token of $(quoteinclude) is an integer greater than zero, then instances of #include "..." directives existing within the viewpath are detected and an error with a severity level of the value of the first token is issued. The severity level matches the level specified in the nmake error statement.

If the optional prefix parameter is specified then warnings are confined to quoted headers within the viewpath that inherit a directory prefix. The inherited prefix may originate from the file's parent or an ancestor file that does not directly include the file. This is useful with compilers that provide means to override automatic search of quoted include files in the current directory but do not support prefix include processing, in cases where the nmake cpp is not used. This is the -J option on some compilers. If specified, the prefix parameter must be the second token in the variable. By default, $(quoteinclude) is null and no errors are issued.

Example

In this example hello.c uses #include "abc/a1.h" and a1.h uses #include "a2.h". When using quoteinclude=3, processing is halted due to the error. When the prefix option is specified only a2.h is identified since it inherits the abc/ directory prefix but a1.h does not inherit a directory prefix.

Show environment

```
$ echo $VPATH
/home/richb/dir1:/home/richb/dir2

$ pwd
/home/richb/dir1/src

$ ls -l
```

```
total 4
-rw-r--r--   1 richb   richb        13 Feb 15 14:28 Makefile
drwxr-xr-x 2 richb   richb      4096 Feb 15 14:28 abc
-rw-r--r--   1 richb   richb        49 Feb 15 14:26 hello.c

$ ls -l abc
total 0
-rw-r--r--   1 richb   richb        16 Feb 15 14:24 a1.h
-rw-r--r--   1 richb   richb        11 Feb 15 14:28 a2.h
```

Contents of Makefile

```
hello :: hello.c
```

Run nmake

```
$ nmake quoteinclude=1
```

Output

```
make: warning: hello.c uses quoted include for abc/a1.h
make: warning: abc/a1.h uses quoted include for a2.h
+ cc -O -I. -Iabc -I- -c hello.c
+ cc -O -o hello hello.o
```

Run nmake

```
$ nmake quoteinclude=2
```

Output

```
make: *** error hello.c uses quoted include for abc/a1.h
make: *** error abc/a1.h uses quoted include for a2.h
+ cc -O -I. -Iabc -I- -c hello.c
+ cc -O -o hello hello.o
```

Run nmake

```
$ nmake quoteinclude=3
```

Output

```
make: *** error hello.c uses quoted include for abc/a1.h
```

Run nmake

```
$ nmake quoteinclude="1 prefix"
```

Output

```
make: warning: abc/a1.h uses quoted include for a2.h
+ cc -O -I. -Iabc -I- -c hello.c
+ cc -O -o hello hello.o
```

recurse Base Rule Variable

recurse - specify maximum number of concurrent recursive makes

Description

This variable specifies the maximum number of recursive makes that can be done concurrently when using the :MAKE: assertion operator. The default is 1. The NPROC environment variable or the jobs command-line option must be set to a value greater than or equal to $(recurse) for multiple recursive makes to execute.

In the following example, the :MAKE: assertion operator is used to run nmake on the makefiles in the liba, libb, and exec subdirectories.

```
:MAKE: liba libb - exec
```

Example

For this example, the liba, libb, and exec subdirectories contain the following makefile:

```
target :
        sleep 10
        pwd
```

When nmake is run with the following options,

```
$ nmake-o jobs=3 recurse=3
```

nmake is started on liba/Makefile and libb/Makefile. When they are complete, nmake is started on exec/Makefile. The output is shown below:

liba build starts

```
liba:
```

libb build starts

```
libb:
+ sleep 10
+ sleep 10
+ pwd
```

liba build completes

```
/u1/bob/nmake/liba
+ pwd
```

libb build completes

```
/u1/bob/nmake/libb
```

exec build starts

```
exec:
+ sleep 10
+ pwd
```

exec build completes

```
/u1/bob/nmake/exec
```

recurse_begin_message, recurse_end_message

recurse_begin_message - specify a text message for recursing into a directory or file.
recurse_end_message - specify a text message for returning from a directory or file.

Description

There are three distinct recurse message types: default, gnu, and custom. These types are not compatible; changing the type during a recursive make is not supported and will trigger a warning message.

recurse_begin_message = default

The **default** type provides traditional nmake style recurse messages. This prints the target directory or file before recursion and nothing after recursion. The default messages are output only for baserules provided recursion facilities such as :MAKE:. This is the default setting for recurse_begin_message.

recurse_begin_message = gnu

The **gnu** type provides GNU make compatible recurse begin and end messages. These messages emulate GNU make recurse messages and contain the current recursion level and full target directory. This format is especially useful when using nmake with development tools that already understand the GNU recurse message formats. These development tools typically use recurse messages to track the current directory in recursive builds, allowing the tool to perform automatic mapping of compilation errors to the source file and line containing the error. Examples of such tools are the Eclipse CDT C/C++ IDE and GNU Emacs compilation mode. GNU type recurse messages are output in sub-makes at the very beginning (before engine bootstrap) and very end of an nmake run, allowing accurate tracking of all errors encountered during a run including early errors output by nmake itself. GNU type messages are output for all sub-makes, including those invoked by user defined rules. Message output is fully automatic, nothing special is required in user defined recurse rules to obtain GNU style messages.

recurse_begin_message = <user-defined-message>
recurse_end_message = <user-defined-message>

The custom recurse messages allow user specification of recurse begin and end message format. The value of $(recurse_begin_message) is printed when nmake recurses into a subdirectory or to a makefile. The value of $(recurse_end_message) is printed when nmake returns from a subdirectory or makefile. The messages are printed for any recursive nmake invocation including custom recurse rules. Variable expansion is performed at .MAKEINIT time so nmake variables must be defined at that point. The current numeric recursion level can be printed using the

$(MAKELEVEL) variable. As with GNU type messages, output is fully automatic, nothing special is required in user defined recurse rules to obtain custom style messages.

Example

The following example will print messages showing the directory when recursing along with the recursion level.

Contents of global.mk

```
recurse_begin_message = LEVEL $(MAKELEVEL) - Entering $(PWD)
recurse_end_message = LEVEL $(MAKELEVEL) - Leaving $(PWD)
```

Contents of top-level Makefile

```
:MAKE: dir_one dir_two
```

Contents of dir_one/ and dir_two/ makefiles

```
targ:
        : $(<) $(PWD)
```

Run nmake

```
$ nmake -g $PWD/global.mk
```

Output

```
LEVEL 1 - Entering /home/dma/dir_one
+ : targ /home/dma/dir_one
LEVEL 1 - Leaving /home/dma/dir_one
LEVEL 1 - Entering /home/dma/dir_two
+ : targ /home/dma/dir_two
LEVEL 1 - Leaving /home/dma/dir_two
```

The next example prints the directory and a time stamp.

Contents of global.mk

```
recurse_begin_message = Entering $(PWD) - $("":T=R:F=%(%b-%d-%Y %H:%M:%S)T)
recurse_end_message = Leaving $(PWD) - $("":T=R:F=%(%b-%d-%Y %H:%M:%S)T)
```

Run nmake

```
$ nmake -g $PWD/global.mk
```

Output

```
Entering /home/dma/dir_one - Jul-11-2007 14:10:56
+ : targ /home/dma/dir_one
Leaving /home/dma/dir_one - Jul-11-2007 14:10:56
Entering /home/dma/dir_two - Jul-11-2007 14:10:57
+ : targ /home/dma/dir_two
Leaving /home/dma/dir_two - Jul-11-2007 14:10:57
```

The final example prints GNU make style recurse messages.

Contents of global.mk

```
recurse_begin_message = gnu
```

Run nmake

```
$ nmake -g $PWD/global.mk
```

Output

```
make[1]: Entering directory ‘/home/dma/dir_one’
+ : targ /home/dma/dir_one
make[1]: Leaving directory ‘/home/dma/dir_one’
make[1]: Entering directory ‘/home/richb/build/dir_two’
+ : targ /home/dma/dir_two
make[1]: Leaving directory ‘/home/dma/dir_two’
```

save Base Rule Variable

save - list files to be saved by common actions

Description

This variable can be set to a list of files that should be saved by the save-oriented common actions (cpio, pax, shar, tar) in addition to the list of source files from the source dependency rules.

select Base Rule Variable

select - use edit operators to select files to be saved

Description

This variable can be set to a list of edit operations that are applied to the list of source files before the save-oriented common actions execute (e.g., select=N!=*.c prevents any .c files from being saved).

Example

In this example, the makefile creates three programs.

```
hello :: hello.c
jello :: jello.c
yello :: yello.c
```

When nmake is run,

```
$ nmake pax
```

the following command is run:

```
+ pax -w -f hello.pax  hello.c jello.c yello.c Makefile
```

When nmake is run with the following options,

```
$ nmake pax 'select=N=*.c:N!=j*'
```

the following command is run:

```
+ pax -w -f hello.pax  hello.c yello.c
```

sge_qrsh Base Rule Variable

sge_qrsh - enable the use of Sun Grid Engine or other job distribution utility.

Description

Allows job action blocks to be piped to a job distribution utility such as Sun Grid qrsh. Each triggered action block is piped to a separate invocation of qrsh or other distribution utility. No makefile changes are needed, the feature works for arbitrary existing makefiles. To enable the feature set sge_qrsh=1. All regular shell jobs are spun off (currently this is done without regard to the .LOCAL attribute.) The feature is generic and other compatible job distribution commands may be substituted for qrsh which is used by default.

The following variables are used for fine control of this feature.

sge_qrsh	Enable use of Sun Grid or other distribution utility. default: sqe_qrsh = 0
QRSH	The job distribution utility command. default: QRSH = qrsh
QRSHFLAGS	Command line flags for the distribution utility. default: QRSHFLAGS += -cwd -noshell -V
SGESHELL	The remote shell to invoke. default: SGESHELL := $(SGESHELL|COSHELL|SHELL|"ksh")

sharedliblinks Base Rule Variable

sharedliblinks - controls symbolic linking of shared libraries

Description

This variable controls the creation of symbolic links during the installation of the shared library, created with the :LIBRARY: operator. By default, this is set to "1" which will cause various links to be made to facilitate installing the shared library. When set to "0", the generation of these links will be turned off and the target shared library will simply be installed.

Example

This is a simple example that demonstates the difference between the default behavior of installing the shared library versus the behavior with sharedliblinks turning off the generation of the symbolic links.

Contents of Makefile

```
.SOURCE : src
INSTALLROOT = .
CCFLAGS = $(CC.PIC)

:ALL:
db 2.0 :LIBRARY: a.c b.c
```

Run nmake

```
$ nmake install
```

Output

```
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/a.c
make: warning: a.o file system time lags local time by at least 13.00s
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/b.c
+ ar r libdb.a a.o b.o
ar: creating libdb.a
+ rm -f a.o b.o
+ nm -p libdb.a
+ sed -e /[ ][TDBC][ ][          ]*[_A-Za-z]/!d -e s/.*[ ][TDBC][ ][ ]*// -e /_STUB_/d -e s/^/-u /
+ ld -G -o libdb.so.2.0 -u a -u b libdb.a
+ mkdir -p lib
+ 2> /dev/null
+ cp libdb.a lib/libdb.a
+ /usr/bin/rm -f lib/libdb.so
+ /usr/bin/ln -s libdb.so.2.0 lib/libdb.so
+ /usr/bin/cp libdb.so.2.0 lib/libdb.to.2.0
+ /usr/bin/rm -f lib/libdb.so.2.0
+ /usr/bin/ln -s libdb.to.2.0 lib/libdb.so.2.0
+ /usr/bin/ln lib/libdb.to.2.0 lib/libdb.no.2.0
+ /usr/bin/rm -f lib/libdb.so.2.0
+ /usr/bin/ln -s libdb.no.2.0 lib/libdb.so.2.0
+ /usr/bin/rm lib/libdb.to.2.0
```

Now let's run nmake with sharedliblinks=0 (after cleaning up).

Run nmake

```
$ nmake sharedliblinks=0 install
```

Output

```
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/a.c
make: warning: a.o file system time lags local time by at least 13.00s
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/b.c
+ ar r libdb.a a.o b.o
ar: creating libdb.a
+ rm -f a.o b.o
+ nm -p libdb.a
+ sed -e /[ ][TDBC][ ][          ]*[_A-Za-z]/!d -e s/.*[ ][TDBC][ ][ ]*// -e /_STUB_/d -e s/^/-u /
+ ld -G -o libdb.so.2.0 -u a -u b libdb.a
+ mkdir -p lib
+ 2> /dev/null
+ cp libdb.a lib/libdb.a
+ cp libdb.so.2.0 lib/libdb.so.2.0
```

sharedlibvers Base Rule Variable

sharedlibvers - controls whether shared library is created with version number

Description

One of the arguments to the :LIBRARY: operator allows one to create shared libraries with version numbers, *major.minor* version string. By default, the shared library is created with the 1.0 version number, otherwise it is created with the *major.minor* version number specified to :LIBRARY:. The choice of whether the shared library is created with a version number is now handled by sharedlibvers. When set to "1", the default, shared libraries with version numbers will be created. However, when set to "0", shared libraries without version numbers will be created, even if the *major.minor* number is specified to :LIBRARY:. So, with sharedlibvers set to 0, a plain lib*name*.$(CC.SUFFIX.SHARED) will be created. (CC.SUFFIX.SHARED is .so and .sl in the Solaris and HPUX environments, respectively).

Example

This example will use the sharedlibvers variable to create a shared library without version number, even if the major.minor is specified in the :LIBRARY:.

Contents of Makefile

```
.SOURCE : src
INSTALLROOT = .
CCFLAGS = $(CC.PIC)

:ALL:
db 2.0 :LIBRARY: a.c b.c
```

Run nmake

```
$ nmake sharedlibvers=0
```

Output

```
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/a.c
make: warning: a.o file system time lags local time by at least 13.00s
+ ppcc -i /tools/nmake/sparc5/lu3.3/lib/cpp cc -KPIC -I-D/tools/nmake/sparc5/lu3.3/lib/probe/
C/pp/BFE0A4FEobincc -I- -c src/b.c
+ ar r libdb.a a.o b.o
ar: creating libdb.a
+ rm -f a.o b.o
+ nm -p libdb.a
+ sed -e /[ ][TDBC][ ][         ]*[_A-Za-z]/!d -e s/.*[ ][TDBC][ ][ ]*// -e /_STUB_/d -e s/^/-u /
+ ld -G -o libdb.so -u a -u b libdb.a
+ mkdir -p lib
+ 2> /dev/null
+ cp libdb.a lib/libdb.a
+ cp libdb.so lib/libdb.so
```

symbolic_link Base Rule Variable

symbolic_link - forces :LINK: to attempt symbolic link instead of hard-link

Description

This variable when set to a non-zero value will force the :LINK: operator to attempt a symbolic link of its *lhs* files to the *rhs* files. If set to 0 or null (its default value), an hard link will be attempted.

NOTE: If your system does NOT support symbolic linking, a warning message is issued, and an hard link is attempted.

Example

The following example assumes that the system supports symbolic linking.

Contents of Makefile:

```
:ALL:
y.a :LINK: x.a
x.a :: x.c
```

Run nmake:

```
$ nmake symbolic_link=1
```

Output:

```
+ ppcc -i /tools/nmake/lu3.3/lib/cpp cc -O -I-D/tools/nmake/lu3.3/lib/probe/C/pp/890
893F4obincc -I- -c x.c
+ ar r x.a x.o
+ rm -f x.o
+ ln -s x.a y.a
```

tmp Base Rule Variable

tmp - temp file hash name, maximum 10 characters

Description

This variable is set to the value of $COTEMP. (See the nmake manual page or *Engine Environment Variables*, later in this section.).

viewverify Base Rule Variable

viewverify - issue warning or error if viewpath not set

Description

If $(viewverify) is non-null and there are no views, an error is issued with a severity level of the value of $(viewverify). The severity level matches the level specified in the nmake error statement. In the following example, there are no views (i.e., VPATH is not set, MAKEPATH is not set, there are no 3-D viewpaths set, and nmake is not invoked on the first VPATH node).

Makefile

```
hello.o :
```

Shell actions

```
$ export MAKE_OPTIONS=viewverify=3
$ echo $VPATH
```

Run nmake

```
$ nmake
```

Output

```
make: viewpath not set
```

NOTE:
This variable can not be set in the makefile because the VPATH test is performed before the makefile is read.

Base Rule Variables that can be Used in Action Blocks

A complete list of variables that are defined in the default base rules and that can be used in action blocks follows. The values of variables prefixed with CC are defined in the probe configuration file; see *Appendix A* in the *Alcatel-Lucent nmake Product Builder User's Guide* for details.

Variable	**Default Value**
AR	ar
ARFLAGS	r
AS	as
ASFLAGS	null
AWK	awk
BINDIR	$(INSTALLROOT)/bin
BINED	ed
CC	cc
cc	cc
CCFLAGS	-O
CCLD	$(CC)
CC.ARCHIVE	$(*.ARCHIVE:O=1)
CC.PROBE	$(cctype:@P=P=C)
CHGRP	chgrp
CHMOD	chmod
CHOWN	chown
CIA	$(CC.ALTPP.ENV) $(CC.DIALECT:N=C++:?CIA ? cia?)
CIADBFLAGS	null
CIAFLAGS	null
CMP	cmp
COATTRIBUTES	null
CP	cp
CPIO	cpio
CPIOFLAGS	null
CPP	$(MAKEPP)
CPPFLAGS	$(CCFLAGS:N=-[DIU]*)
CTAGS	ctags (if found in $PATH)

Variable	Default Value
EGREP	egrep
ETCDIR	$(INSTALLROOT)/etc
FEATURE	feature
FEATUREFLAGS	set cc $(CC) $(CCFLAGS) \ $(LDFLAGS) : $(-mam??set static $(CC.STATIC) : ?)
F77	f77
F77FLAGS	null
FUNDIR	$(INSTALLROOT)/fun
GREP	grep
HOSTCC	cc
INCLUDEDIR	$(INSTALLROOT)/include
IGNORE	ignore
INSTALLROOT	$(HOME)
LD	$(CC.LD)
LDFLAGS	null
LDLIBRARIES	null
LEX	lex
LEXFLAGS	null
LIBDIR	$(INSTALLROOT)/lib
LINT1	$(LINTLIB)/lint1
LINT2	$(LINTLIB)/lint2
LINTLIB	The directory of the first occurrence of lint1 found in the following order: /usr/local/lib/lint /local/lib/lint /usr/lib/lint /usr/local/lib/cmplrs/cc /local/lib/cmplrs/cc /usr/lib/cmplrs/cc /usr/local/ccs/lib /local/ccs/lib /usr/ccs/lib If it is not found in any of the above directories, the value is set to /usr/lib.
LINTLIBRARIES	libc$(CC.ARCHIVE)
LINTFLAGS	-bh (if machine is not running UNIX System V)
LN	ln

Variable	**Default Value**
LPR	lpr
LPRFLAGS	null
LPROF	lprof (if found in $PATH)
LS	ls
MAKEPP	$(MAKELIB)/../cpp
MKDIR	mkdir
MANDIR	$(INSTALLROOT)/man/man
M4	m4
M4FLAGS	null
MCS	mcs (if found in $PATH)
MCSFLAGS	-d (if mcs is found in $PATH)
MV	mv
NM	$(CC.NM)
NMEDIT	$(CC.NMEDIT) -e "/_STUB_/d"
NMFLAGS	$(CC.NMFLAGS)
PAX	pax
PPCC	$(MAKELIB:D:D)/bin/ppcc
PR	pr
PROTO	proto
PROTOCOPYRIGHT	$("$(HOME).copyright":T=F:T=I:@/$("\n")/ /G :@??Alcatel-Lucent Bell Labs?O)
PROTOFLAGS	-s -c'$(PROTOCOPYRIGHT)'
QRSH	qrsh
QRSHFLAGS	-cwd -noshell -V
RNLIB	$(IGNORE) ranlib
RM	rm
RMFLAGS	-f
SED	sed
SGESHELL	$(SGESHELL|COSHELL|SHELL|"ksh")
SHAR	shar
SHAREDIR	$(INSTALLROOT)/share
SILENT	silent
STRIP	strip
TAR	tar
TARFLAGS	v

Variable	**Default Value**
TMPDIR	usr/tmp
VGRIND	vgrind (if found in $PATH)
YACC	yacc
YACCFLAGS	-d

Directory Variables

Common directories are named by variables. The directory hierarchy is modeled after /usr in System V.

BINDIR = $(INSTALLROOT)/bin	The installation directory for generated commands.
ETCDIR = $(INSTALLROOT)/etc	The installation directory for generated miscellaneous system files.
FUNDIR = $(INSTALLROOT)/fun	The installation directory for ksh *functions.*
INCLUDEDIR = $(INSTALLROOT)/include	The installation directory for generated header files.
INSTALLROOT = $(HOME)	The root installation directory used by the :INSTALL: assertion operator and the install common action.
LIBDIR = $(INSTALLROOT)/lib	The installation directory for generated libraries and data files.
MANDIR = $(INSTALLROOT)/man/man	The installation directory prefix for man(1) pages. *.nnn man page files are installed in the $(MANDIR)nnn directory (e.g., nmake(1) is installed in the $(INSTALLROOT)/man/man1 directory).
SHAREDIR = $(INSTALLROOT)/share	The installation directory accessed by all hosts of the local network.
TMPDIR = /usr/tmp	The temporary file directory.

Engine Environment Variables

The file names and directories used by the nmake engine are parameterized using engine variables. These variables are assigned default values by the nmake engine. Some of the values are determined at installation time, while others are determined by each environment at invocation time . Default values are in **bold type**. Variables that are underlined designate values that are determined at invocation time.

COTEMP=**new value for each command**	The COTEMP environment variable is generated and set to a different value for each shell command. COTEMP is a 10-character hash name. Names of the form *prefix*.$COTEMP.*suffix* must be 14 characters or fewer to satisfy the System V file name length restriction. If COSHELL is set to ksh or sh, the value of COTEMP is *nmake-pidjob-pid*. If COSHELL is set to coshell, the value of COTEMP is *job-host-inet-addr-base-32job-pid-base-32*. Since Internet addresses must be unique among all hosts accessible to coshell, the COTEMP value is unique for all jobs on all hosts in the local network.
MAKE=**the current nmake program**	The path of the current nmake program suitable for use as a shell command or in an execvp(3) call. This variable is especially useful when calling nmake recursively in a makefile.
MAKEARGS=**Makeargs:makeargs**	A colon-separated list of the candidate argument file names.
MAKEFILE=**the name of the first makefile**	The first makefile found in the default list in $(MAKEFILES).
MAKEFILES=**Makefile:makefile**	A colon-separated list of candidate implicit makefile names.
MAKELEVEL=**integer recursion level**	Current makefile recursion level.
MAKELIB=$nmake_install_root**/lib/make**	The directory for nmake related files, e.g., base rules.

MAKEPP=$(MAKELIB)**/../cpp**	The name of the nmake preprocessor. Preprocessing the makefile with $(MAKEPP) is not recommended. nmake programming constructs should be used instead of makefile preprocessing.
MAKERULES=**makerules**	The name of the default base rules. The base rules file must be a compiled makefile; it is named by changing the suffix in $(MAKERULES) to .mo and binding the result using the directories in .SOURCE.mk. **makerules** can be an absolute path name.
MAKERULESPATH=$(LOCALRULESPATH): $(MAKELOCALPATH): $(INSTALLROOT|HOME)**/lib/make**: $(PATH:/:/ /G:D:B=lib/make:@/ /:/G): $(MAKELIB): **/usr/local/lib/make:/usr/lib/make**	Used to initialize .SOURCE.mk. All makefiles, including compiled makefiles and files included by makefiles, are bound using the directories in .SOURCE.mk. $(LOCALRULESPATH) and $(MAKELOCALPATH) are null by default. These variables can be used to point to the directory containing the project rules.
MAKEVERSION=**current nmake version string**	The nmake engine version stamp.
OLDMAKE=**/bin/make**	Executed using execvp(3) when the first input makefile is not a valid nmake makefile.
PWD=**the absolute path name of the current directory**	The current directory is the directory of the makefile.
VOFFSET=**current directory offset**	Set to the path that goes from the viewpath node root to the current directory.
VROOT=**relative path to vpath root**	Set to the relative path that goes from the viewpath in the current directory (.) to the viewpath node root.

Uninitialized Environment Variables

Uninitialized variables have null default values and are used only when defined.

COSHELL	The name of the shell or shell process server used to handle shell actions. execvp(3) is used to execute the shell. If COSHELL is not defined, then ksh, sh, and /bin/sh are tried in that order.
EXTRASTATE	This may be used to specify the name of a makefile whose state information will be used by the primary makefile. For example, this is used in the :cc: operator to refer to the name of the makefile whose statefile information is needed to complement the statefile information of the primary statefile, null.ms. In this case :cc: sets it to $(MAKEFILE).
MAKECONVERT	This variable is used for makefile conversion at compile time. For example, given a conversion program nmkgen to convert from build makefiles to nmake makefiles and a build makefile called make.in, this variable must be set in .profile as follows: `export MAKECONVERT='make.in "nmkgen $(>)"'` nmake produces .mo and .ms files from make.in; make.in remains a build makefile.

MAKEEXPORT	A colon-separated list of variable names, whose values are set in the nmake environment (by the user) and exported (by the engine) in the environment of the shell co-process. Hence, they can be referenced as shell variables in the shell action block.
	 `$ cat makefile`
	 `A = a`
	 `B = b`
	 `C = c`
	 `hello: .VIRTUAL`
	 `        silent echo $(A) $(B) $(C)`
	 `        silent echo $A $B $C`
	 `$ nmake MAKEEXPORT=A:B`
	 `+ echo a b c`
	 `+ echo a b`

MAKEIMPORT	A colon-separated list of environment variable names. the values of those names override any makefile variable assignments.

MAKEPATH	A colon-separated list of directory names from which nmake could be run on the current makefile. These directories are used by the initialization script to initialize the .VIEW Special Atom. The first view is always the current directory (.) by default. For example, if MAKEPATH = n1 : n2, nmake automatically changes .VIEW to . : n1 : n2 .
MAKE_OPTIONS	This variable sets the makerules variables. For example, MAKE_OPTIONS=ancestor=2. Multiple variables can be set. For example, MAKE_OPTIONS="ancestor=2 cctype=/bin/cc".
NPROC	Defines the maximum number of COSHELL processes that can be executed simultaneously. Setting NPROC is equivalent to using the jobs command-line option (-j). However, the jobs command-line option overrides the NPROC environment variable.
PROBEPATH	A colon-separated list of alternate probe hierarchy root paths. When used in conjunction with the localprobe base rules variable, this variable can be used to determine the search order for local probe information.
UMASKCHANGE	May be 0/NO/no to supress changing the umask to match the current directory permissions, or 1/YES/yes/null to change the umask. For compatibility the umask change is enabled by default. Any other value generates a warning and keeps the umask change enabled.
VERSION_ENVIRONMENT	A colon-separated list of environment variables for probe to consider when generating the filename hash for the probe configuration file.
VPATH	A colon-separated list of viewpath node names. The node names are converted to MAKEPATH directories that are used by the initialization script to initialize the .VIEW Special Atom. VPATH is ignored if nmake is not executing within the first viewpath node. Since the current directory is not set as the first node, it is necessary to be at the first node of the VPATH to use the viewpathing; otherwise, viewpathing is turned off.

State Variables Defined in the Default Base Rules

The values shown are the default values. Any of the values can be changed by reassigning values before the *target definition* line of the makefile. When expanded, the values given to variables in the makefile override the default.

Variable	Default Value
AR	ar
ARFLAGS	r
AS	as
ASFLAGS	null
CC	cc
CCFLAGS	-O
CCLD	$(CC)
CIA	$(CC.ALTPP.ENV) $(CC.DIALECT:N=C++:?CIA ? \ cia?)
CIADBFLAGS	null
CIAFLAGS	null
COATTRIBUTES	null
CPP	$(MAKEPP)
FEATURE	feature
FEATUREFLAGS	set cc $(CC) $(CCFLAGS) $(LDFLAGS) : \ $(-mam??set static $(CC.STATIC) : ?)
F77	f77
F77FLAGS	null
LD	$(CC.LD)
LDFLAGS	null
LDLIBRARIES*	null
LEX	lex
LEXFLAGS	null
M4	m4
M4FLAGS	null
YACC	yacc
YACCFLAGS	-d

*The value of LDLIBRARIES is the ordered list of libraries to be linked to all executable targets built with ::; it is automatically passed to the loader at link time.

5 Edit Operators

Edit operators allow variable values to be tested and modified during expansion. This section contains manual pages for all the edit operators.

A - Attribute Selection

A - Attribute Selection

Synopsis

$(S:[@]A[[!] = *expression1* **[** | *expression2* **] ...])**
$(S:[@]A<*pattern***)**
$(S:A>*pattern***)**

Description

This edit operator selects tokens (in the value of the expanded variable S) that have the user attribute or prerequisite *expression1*. If the edit operator is followed by an exclamation point (A!), it selects tokens that do not have the user attribute or prerequisite *expression1*. The optional pipe (|) specifies the inclusive OR from the list of expressions. If the edit operator is preceded by @, the value of the variable is treated as a single token. Where **A** is used alone, in the form **$(S:A)**, this expands to the list of user-defined attributes of the tokens of variable S.

In the form:

$(S:A<*pattern***)**

this edit operator determines which pattern association rules have been applied to an atom. *pattern* is the basename of the pattern association rule and the variable S contains the names of the atoms.

In the form $(S:A>*pattern*) this edit operator selects target tokens with the specified prerequisite *pattern*.

Examples

This example declares .MYAT as an attribute and assigns the .MYAT attribute to files with a .g suffix. When this makefile is executed, the following expansions take place, since there is no association rule applied to a.x.:

Variable	Expanded Value
$(FILES:A)	.MYAT
$(FILES:A=.MYAT)	a.g
$(FILES:A!=.MYAT)	a.x
$("a.g":A<.ATTRIBUTE.)	.ATTRIBUTE.%.g
$("a.x":A<.ATTRIBUTE.)	nothing

Contents of Makefile

```
FILES = a.g a.x
.MYAT: .ATTRIBUTE
.ATTRIBUTE.%.g: .MYAT

target :
        : $(FILES:A)
        : $(FILES:A=.MYAT)
        : $(FILES:A!=.MYAT)
        : $("a.g":A<.ATTRIBUTE.)
        : $("a.x":A<.ATTRIBUTE.)
```

Run nmake

```
$ nmake
```

Output

```
+ : .MYAT
+ : a.g
+ : a.x
+ : .ATTRIBUTE.%.g
+ :
```

The following example shows that this operator allows both attributes (.MYAT) and prerequisites (a.c, b.c, c.c, and d.c).

Contents of Makefile

```
.MYAT : .ATTRIBUTE
all : target1 target2 x
target1 : .MYAT
target1 : a.c b.c
target2 : c.c d.c
TARGS = target1 target2

x :
        : $(TARGS:A=a.c)
        : $(TARGS:A=a.c|c.c)
        : $(TARGS:A=.MYAT)
        : $(TARGS:A=.MYAT|c.c)
```

Run nmake

```
$ nmake
```

Output

```
+ : target1
+ : target1 target2
+ : target1
+ : target1 target2
```

The following example demonstrates the use of this edit operator to select target tokens that have specific prerequisite patterns.

Contents of Makefile

```
LIST = x y z
x : a.c b.c
y : a.c c.x
z : c.x

tst :
        : $(LIST:A>*.c)
```

Run nmake

```
$ nmake
```

Output

```
: x y
```

C - Changer

C - Changer

Synopsis

$(S:[@]C *<delim> old <delim> new <delim>* **[g])**

Description

This edit operator can be used to change character strings in tokens (in the value of the expanded variable *S*.) In each token, it substitutes the string *new* for the first occurrence of the string *old*. The addition of the character g (global) after the last delimiter character causes the *new* string to be substituted for every occurrence of the *old* string in each token.

If the edit operator is preceded by @, the value of the variable is treated as a single token.

<delim> is any delimiter character. The slash (/) is so frequently used as the delimiter character that :C/ is abbreviated as :/.

The patterns specified by *old* and *new* follow the regular expression pattern match from the ed(1) s(ubstitute) command. (See the *UNIX System V User's Reference Manual* for more information concerning the ed command.)

Example

In the following example, the delimiter is the pipe (|) character.

Contents of Makefile

```
FILES = aa.g aa.c
tst:
        : $(FILES:C|a|xx)
        : $(FILES:@C|a|xx)
        : $(FILES:@C|a|xx|g)
```

Run nmake

```
$ nmake
```

Output

```
+ : xxa.g xxa.c
+ : xxa.g aa.c
+ : xxxx.g xxxx.c
```

D:B:S - Full Path Name

D:B:S - Full Path Name

Synopsis

$(S:[@][D[=*new_dir*]][:]:[B[=*new_base*]] [:]:[S [=*new_suffix*]])

Description

These edit operators treat each token of the variable S as a path that is partitioned into three components: directory, basename, and suffix. The component types (D, B, and S) specified in the edit operation are selected and replaced by the new component name, if specified.

The directory component (specified by D) includes all characters up to, but not including, the last slash (/).

The basename component (specified by B) includes all characters between the last slash (/) and the last dot or set of dots (.). For both the following path names

```
/home/u1/ral/x.y.o
/home/u1/ral/x.y..o
```

x.y is the basename.

The suffix component (specified by S) includes the last set of dots and all characters after the last set of dots; in the above examples, the suffix components are .o and ..o, respectively. If a dot is the first character after the last slash and is the only dot, it is included in the basename component rather than the suffix component.

These edit operators, although separated by :, are grouped together for evaluation by nmake. Each is applied as a single edit operation; they do not form token pipelines. Token pipelines can be forced by adding extra colons (:) between these edit operations, for example, $(S:D::B).

When *new_dir* is specified, the directory component of each token is changed to *new_dir*. The same is true for *new_base* and *new_suffix*. These values can be specified as a single value resulting from a variable expansion.

A variable with a null value expands to a directory component of dot and null basename and suffix components. When a token value does not contain a specified component such as directory and a new component name is given, the expanded value includes the new component name.

The D:B:S edit operators do not cause the expanded values to be bound.

Example

For each of the examples below, the following variable, having three tokens as its value, is used:

FILES = a.c dir1/x.c dir2/.profile

The first token in the value of FILES is a.c, the second is dir1/x.c, and the third is dir2/.profile.

Edit Command	Result
$(FILES:D)	. dir1 dir2
$(FILES:B)	a x .profile
$(FILES:S)	.c .c

The third token has no suffix and produces a null value.

The following example shows the use of the D:B:S edit operators to change various components. The following variable is used in this example:

FILES = x.c dir1/y.c dir2/z.c

With FILES defined in this way, the edit command $(FILES:D=dir3:B:S) produces the following result: dir3/x.c dir3/y.c dir3/z.c, while the edit command $(FILES:D:B:S=.o) produces x.o dir1/y.o dir2/z.o.

E - Evaluation

E - Evaluation

Synopsis

```
$(S:E)
```

Description

This edit operator performs a non-tokenizing edit operation, that is, it treats the entire string S as a single value that contains an expression to be evaluated. The expression may be given as either a logical expression, a string expression, or an integer expression. The expanded value of the E edit operator is the resulting evaluation of the expression. Logical expressions evaluate to 0 on false, and 1 on true. The available logical operators include:

==	equivalence comparison
<	logical less than
<=	logical less than or equal to
>	logical greater than
>=	logical greater than or equal to
!=	logical not equal

Integer expressions can be given as logical comparisons or as integer evaluations. The logical operators listed above may be used in integer expressions as well as string comparisons. In addition, standard integer evaluation is available and includes the complete list of arithmetic operators.

Example

The E edit operator can be used to evaluate an integer expression needed by another edit operator. Frequently, it is used with the O edit operator to satisfy an integer expression.

Suppose a makefile contains an integer variable counter and a variable LIST, such that, if counter is even, the first token of LIST is needed, and if counter is odd, the second token is needed. The following statement produces the desired result:

```
$(LIST:O=$("(counter & 0x01)+1":E))
```

counter is an integer variable, and nmake expands its value whenever the name is encountered. Therefore, it is not necessary to specify the expansion using the syntax $(counter). The entire integer expression must be given within delimiting double quotes. Also, parentheses are used for proper mathematical evaluation.

F - Format

F - Format

Synopsis

$(S:[@]F =% [-] [*n*] [.*m*] *c*)

Description

This edit operator converts each token according to the rules governing the Standard C library printf(3S) function.

If the edit operator is preceded by @, the value of the variable is treated as a single token.

The optional - is used to left-justify the expanded token's formatted value. The length of the output field for each token is given by the optional characters *n.m*, where *n* represents the minimal field width for printing, and *m* is the maximum number of characters to be printed. Each token is printed in a field at least as wide as the minimum field width, and wider, if necessary. When the expanded token has fewer characters than the minimum field width, the token is padded on the left-hand side by default. When left-hand justification is specified, the padding occurs on the right-hand side.

The format type specifier c signifies the desired output type, which may cause the expanded token value to be converted. Valid format types are:

d	The token value is printed in decimal format. Expanded token values that are strings are given the value 0.
o	The token value is printed in octal format. Expanded token values that are strings are given the value 0.
s	The token value is printed as a string.
u	The token value is printed using unsigned decimal notation.

x	The token value is printed in hexadecimal. Expanded token values that are strings are given the value 0.
S	The token has a string conversion applied according to the specified format: **%(identifier)S** The token is converted to a valid nmake identifier name; invalid characters are converted to an underscore (_). **%(invert)S** The case of each character of the token is inverted. **%(lower)S** The token is converted to lower case. **%(upper)S** The token is converted to upper case. **%(variable)S** The token is converted to a valid nmake variable name; invalid characters are converted to dot (.).
T	The token argument is some number of seconds since Epoch (00:00:00 January 1, 1970), which is converted to the format: *Month_name day hour:minutes:seconds year*. See also F=%(*format*)T - Format Time Output.

See the printf(3S) manual page for more information about printf-style output formatting.

NOTE:
The L, U and V format types are obsolete. See the preferred %(*format*)S syntax above.

Example

Contents of Makefile

```
VARS = var1 VAR2
STRING = S$T*U
tst:
        : $(VARS:F=%(invert)S)
        : $(VARS:F=%(lower)S)
        : $(VARS:F=%(upper)S)
        : $(STRING:F=%(identifier)S)
        : $(STRING:F=%(variable)S)
```

Run nmake

```
$ nmake
```

Output

```
+ : VAR1 var2
+ : var1 var2
+ : VAR1 VAR2
+ : S_T_U
+ : S.T.U
```

Contents of Makefile

```
VAR = abc 123
tst: .MAKE
        print $(VAR:F=%-10s)
        print $(VAR:F=%-10d)
        print $(VAR:F=%-10o)
        print $(VAR:F=%-10x)
```

Run nmake

```
$ nmake
```

Output

```
abc       123
0         123
0         173
0         7b
```

F=%(*format*)T - Format Time Output

F=%(*format*)T - Format Time Output

Synopsis

$(S:F=%(*format*)T)

Description

This edit operator allows the user to specify the format that will be used to display time where S is relative time (the number of seconds since Epoch [00:00:00 GMT on January 1, 1970]) . The following format fields are supported.

%%	% character
%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%c	ctime(3) style date without the trailing newline.
%C	date(1) style date.
%d	Day of month number.
%D	Date as mm/dd/yy.
%e	Blank padded day of month number.
%E	Unpadded day of month number.
%h	Abbreviated month name.
%H	24-hour clock hour.
%i	International date(1) date that includes the time zone type name.
%I	12-hour clock hour.
%j	1-offset Julian date.
%J	0-offset Julian date.
%l	ls(1) -l date that lists recent dates with hh:mm and distant dates with yyyy.
%m	Month number.
%M	Minutes.
%n	newline character.
%p	Meridian (e.g., AM or PM).
%r	12-hour time as hh:mm:ss meridian.
%R	24-hour time as hh:mm.

%S	Seconds.
%t	tab character.
%T	24-hour time as hh:mm:ss.
%U	Week number with Sunday as the first day.
%w	Weekday number.
%W	Week number with Monday as the first day.
%x	Local date style, using tm_info.format[39], that includes the month, day and year.
%X	Local time style, using tm_info.format[38], that includes the hours and minutes.
%y	2-digit year.
%Y	4-digit year.
%z	Time zone type name.
%Z	Time zone name.
%#	Number of seconds since the epoch.

Example

In the following example the first line displays the current date and time and the second shows Epoch. See also the example for T=R.

Contents of Makefile

```
t :
        : $("":T=R:F=%(%a %m-%d-%y %H:%M:%S %Z)T)
        : $("0":F=%(%a %m-%d-%y %H:%M:%S %Z)T)
```

Run nmake

```
$ nmake
```

Output (depending on local time zone):

```
: Tue 06-17-08 11:21:57 EDT
: Wed 12-31-69 19:00:00 EST
```

G - Generate

G - Generate

Synopsis

$(S:G [!] = *metarule_pattern***)**

Description

This edit operator returns file names that can be generated from tokens of the variable S and that match *metarule_pattern*. If the optional ! is used, it returns tokens in variable *S* that cannot generate files matching the *metarule_pattern*. The *metarule_pattern* has the form <*prefix*><*base*><*suffix*>, where one or more of the components may be a literal value or null. The special % character matches all characters of a component except dot (.). When more than one component of *metarule_pattern* is specified, the literal dot must separate them. For example, if only the base and suffix are present, %.% is needed for a match; if all three components are present, %.%.% is needed for a match.

The pattern specified with the G edit operator can match only one metarule pattern. Therefore, when targets of different metarules are needed from the list, multiple G operations must be specified. For example:

$(FILES:G=%.o) $(FILES:G=%.nl)

The above edit operation produces a list that can be generated from $(FILES) and that matches the patterns %.o or %.nl. The edit operation does not cause the generated name list to be bound. However, the results of the G edit operator can be used to assign values to variable names. The list produced by the operation above can be assigned to a variable by using the following assignment:

SRC:= $(FILES:G=%.o) $(FILES:G=%.nl)

Example

In the following example, a metarule that generates multiple targets is asserted. This metarule can transform g.sch to g.c, c_g.c, and g.h. Thus $(FILES:G=%.h) evaluates to g.h. Since main.c can generate main.o using the metarule %.o : %.c specified in the default base rules, $(FILES:G=%.o) evaluates to g.o c_g.o main.o. Since main.c cannot generate any files matching %.h using the metarules specified either in the default base rules or in the makefile, $(FILES:G!=%.h) evaluates to main.c.

Contents of Makefile

```
FILES = g.sch main.c
%.c c_%.c %.h : %.sch
        cat $(>) > $(>:B:S=.c)
        echo "#include '$(>:B).h'" > \
            c_$(>:B:S=.c)
        cat $(>) >> c_$(>:B:S=.c)
        echo "#define $(>:B)_x 1" > $(>:B:S=.h)

tst:
        : $(FILES:G=%.h)
        : $(FILES:G=%.o)
        : $(FILES:G!=%.h)
```

Output

```
+ : g.h
+ : g.o c_g.o main.o
+ : main.c
```

As an another example, suppose the L edit operator is used to read each token of S as a directory and expand the list of all files in the directory. The list can contain object files and data files. The edit operation to find the names of files that can be generated from the list is:

```
$(PWD:L:G=%.o)
```

H - Heap Sort

H - Heap Sort

Synopsis

$(S:H [[<|>|<=|>=][=[U]]])

Description

This edit operator sorts the expanded token list of the variable S using low-to-high (ascending) order, by default. Thus:

```
$("process.c input.c input.o":H)
```

evaluates to input.c input.o process.c by default.

The < sort key causes sorting in ascending, the > sort key in descending order. The <= sort key causes numeric sorting in ascending, the >= sort key numeric sorting in descending order.

Use of the U key ensures a sorted list in which each member is unique.

Examples

Suppose the variable MEMBERS contains the unsorted list of members to be placed in an archive. The member list, however, must be sorted prior to issuing the actual archive command. The edit operation to do this is:

```
MEMBERS := strtok.o strlen.o strcat.o strcpy.o strmrk.o
SORTED_LIST := $(MEMBERS:H)
```

The tokens of variable MEMBERS are sorted from low to high (ascending), and the sorted list is assigned to the variable SORTED_LIST.

This example demonstrates the sorting order of tokens.

Contents of Makefile

```
LIST = 1 bananas 10 apples 2 bananas oranges

target:
        : $(LIST)
        : $(LIST:H)
        : $(LIST:H<)
        : $(LIST:H<=)
        : $(LIST:H>)
        : $(LIST:H>=)
        : $(LIST:H<=U)
```

Run nmake

```
$ nmake
```

Output

```
+ : 1 bananas 10 apples 2 bananas oranges
+ : 1 10 2 apples bananas bananas oranges
+ : 1 10 2 apples bananas bananas oranges
+ : apples bananas bananas oranges 1 2 10
+ : oranges bananas bananas apples 2 10 1
+ : 10 2 1 apples bananas bananas oranges
+ : apples bananas oranges 1 2 10
```

I - Intersection

I - Intersection

Synopsis

$(S:I=*token_list*)

Description

This edit operator expands the value of *token_list* and selects the names in the variable S that also appear in the expanded value of *token_list*. Each selected name appears only once in the return value. The directory path names are canonicalized before comparison.

token_list is a variable name specified in the expansion context or a literal value specified in the immediate value expansion context. For example, *token_list* may be $(PATHS) or $(*automatic_variable*). *token_list* cannot be specified as a pattern.

Examples

This example returns the tokens common between VAR1 and VAR2.

Contents of Makefile

```
VAR1 = a b c d e
VAR2 = d e f g h
targ :
        : $(VAR1:I=$(VAR2))
```

Run nmake

```
$ nmake
```

Output

```
+ : d e
```

This example removes the duplicate tokens.

Contents of Makefile

```
VAR3 = a.c b.c a.c b.c c.c
targ :
        : $(VAR3:I=$(VAR3))
```

Run nmake

```
$ nmake
```

Output

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

This example illustrates how the I edit operator works with the D edit operator.

Contents of Makefile

```
VAR1 = dir1/a.c dir2/b.c dir3/c.c
VAR2 = a.c b.c dir3/c.c
targ :
        : $(VAR1:I=$(VAR2))
        : $(VAR1:D:I=$(VAR2:D))
```

Run nmake

```
$ nmake
```

Output

```
+ : dir3/c.c
+ : dir3
```

K=pfx - Long Line Split

K=*pfx* - Long Line Split

Synopsis

$(S:K[=*pfx*])

Description

Like xargs(1), this edit operator splits long lines into a number of token groups, optionally prepending *pfx* to each group.

Example

The following example illustrates how the K operator can be used to break a string that exceeds the input limit to the ar(1) command into lengths that can be processed. The tokens 1 through 799 are file names; the ellipses (...) indicate intervening tokens.

Contents of Makefile

```
FILES = 1 2 3 ... 275
FILEs += 276 277 278 ... 500
FILES += 501 502 503 ... 700
FILES += 701 702 703 ... 799

target :
        $(FILES:K=ar cr libx)
```

Run nmake

```
$ nmake
```

Output

```
+ ar cr libx 1 2 3 ... 100
+ ar cr libx 101 102 103 ... 201
+ ar cr libx 202 203 204 ... 302
+ ar cr libx 303 304 305 ... 403
+ ar cr libx 404 405 406 ... 504
+ ar cr libx 505 506 507 ... 605
+ ar cr libx 606 607 608 ... 706
+ ar cr libx 707 708 709 ... 799
```

The K operator splits the FILES tokens into strings of a length that the archive command can process.

L - List

L - List

Synopsis

$(S:L [[< | > =] [*pattern*]])

Description

Tokens of the variable S are treated as directories: the contents of the directory matching the shell file match expression *pattern* are expanded. When the variable contains multiple tokens, only the first occurrence of the file name is expanded. The action of this operator is not recursive, nor does it viewpath, so only those files actually appearing in the named directories are listed. This operator does not bind the expanded list.

By default, the generated list is unsorted. The optional < sort key is used to sort the list from low to high (ascending). The > sort key is used to sort from high to low (descending). A sort key should be used only with the optional = key. If a sort key is used without =, only the first or last file in the list is reported.

This edit operator is especially useful for dynamic generation of the list of all files found in directories that are prerequisites of the .SOURCE atom. The special target .SOURCE has as its prerequisites one or more directories to search when looking for source files. The complete list of all files in those directories could be generated by $(*.SOURCE:L), or for a sorted list, $(*.SOURCE:L<=). And by using this edit operator in conjunction with the G edit operator, it is possible to generate the subset of all the files generated by $(*.SOURCE:L<=) that are based on the metarule pattern %.o by using the expression: $(*.SOURCE:L<=:G=%.o).

The List edit operator uses the ksh file match expressions to evaluate those tokens matching the specified pattern. All file match expressions supported by ksh are available for use with the L edit operator. For example, to list the directory contents excluding object files (*.o), the pattern would be given as $(PWD:L<=!(*.o)).

Example

A list of all files in the current working directory can be generated by $(PWD:L); if a sorted list is needed, it may be generated by $(PWD:L<=).

The following example generates a list of .c files from the source directories.

```
SEARCH_PATH = $(*.SOURCE)
FILES = $(SEARCH_PATH:L<=*.c)
```

SEARCH_PATH is set to all the source directories that nmake is told to search. FILES is set to all .c files in those directories. <= specifies that the list of file names is to be sorted in ascending order.

M - Regular Expression Match

M - Regular Expression Match

Synopsis

$(S:[@]M [!] = *regx_pattern***)**

Description

This edit operator selects tokens from the variable S that match the regular expression pattern given as *regx_pattern*. Each token is compared individually to the pattern, unless the @ character precedes the edit operator, in which case case the entire string value of the variable S is treated as a single value to compare to the pattern. The != operator is used to select those tokens that do not match the regular expression pattern. The regular expression patterns follow the syntax of egrep(1).

NOTE:
The N and M edit operators both provide token-match capabilities. The operator to choose depends on the values to be matched.

Example

Note the use of the = and != operators in the following example.

Contents of Makefile

```
NAMES = abcde 12345 fbi.c blu.e
target :
        : 1 - $(NAMES:M=2)
        : 2 - $(NAMES:M!=2)
        : 3 - $(NAMES:M=.c)
        : 4 - $(NAMES:M=\.c)
        : 5 - $(NAMES:M=\.c|\.e)
        : 6 - $(NAMES:M=b.*e)
```

Run nmake

```
$ nmake
```

Output

```
+ : 1 - 12345
+ : 2 - abcde fbi.c blu.e
+ : 3 - abcde fbi.c
+ : 4 - fbi.c
+ : 5 - fbi.c blu.e
+ : 6 - abcde blu.e
```

N - Shell File Match

N - Shell File Match

Synopsis

$(S:[@]N [!] = *shell_pattern***)**

Description

This edit operator selects tokens from the variable S that match the shell file match pattern given as *shell_pattern*. Each token is compared in individually to the pattern unless the @ character precedes the edit operator, in which case the entire string value of the variable S is treated as a single value to compare to the pattern. The != expression is used to select tokens that do not match the shell file match pattern. Shell file match patterns follow the syntax of ksh.

NOTE:
The N and M edit operators both provide token-match capabilities. The operator to choose depends on the values to be matched.

Example

Note the use of the = and != operators in the following example.

Contents of Makefile

```
NAMES = abcd 12345 fbi.c blu.e
target :
        : 1 - $(NAMES:N=*2*)
        : 2 - $(NAMES:N!=*2*)
        : 3 - $(NAMES:N=.c)
        : 4 - $(NAMES:N=*.c)
        : 5 - $(NAMES:N=*.c|*.e)
        : 6 - $(NAMES:N=*[3f]*)
```

Run nmake

```
$ nmake
```

Output

```
+ : 1 - 12345
+ : 2 - abcde fbi.c blu.e
+ : 3 -
+ : 4 - fbi.c
+ : 5 - fbi.c blu.e
+ : 6 - 12345 fbi.c
```

O - Ordinal

O - Ordinal

Synopsis

$(S:O[!] | [*relop n*])

Description

This edit operator numbers each token (1, 2, 3, etc.) by its left-to-right position. If *relop n* is specified, the operator selects tokens whose positions match that specified by *relop* (relational operator) *n* (number). The relational operators are:

<	less than the integer value *n*
<=	less than or equal to the integer value *n*
=	equal to the integer value *n*
>	greater than the integer value *n*
>=	greater than or equal to the integer value *n*
!=	not equal to the integer value *n*.

If O is specified alone, the output is the number of tokens; O! gives the string length of each token.

Example

This example illustrates the use of the = and > operators.

Contents of Makefile

```
VAR=a bb ccc
target :
        : $(VAR:O=1)
        : $(VAR:O>1)
        : $(VAR:O>1:O!)
        : $(VAR:O)
```

Run nmake

```
$ nmake
```

Output

```
+ : a
+ : bb ccc
+ : 2 3
+ : 3
```

P - Path Name

P - Path Name

Synopsis

$(S:P=*op***)**

Description

This edit operator treats each token as a file path and applies the path name edit operator given as *op*. Descriptions of the path name edit operators appear on the following manual pages.

P=A - Absolute Path Name

P=A - Absolute Path Name

Synopsis

$(S:P=A)

Description

This edit operator returns the absolute path for specified files.

Example

Assume that both file f.c and the makefile reside in directory /user/u1/arh/src.

Contents of Makefile

```
FILES= f.c
target:
        : $(FILES:P=A)
```

Run nmake

```
$ nmake
```

Output

```
/user/u1/arh/src/f.c
```

P=B - Physically Bound Token

P=B - Physically Bound Token

Synopsis

```
$(S:P=B)
```

Description

This edit operator returns physically bound tokens.

Example

The following example returns a.c because a.c is bound and b.c is not.

Contents of Makefile

```
FILES = a.c b.c
t : a.c
        : $(FILES:P=B)
```

Run nmake

```
$nmake
```

Output

```
a.c
```

P=C - Path Name Canonicalizing

P=C - Path Name Canonicalizing

Synopsis

$(S:P=C)

Description

This edit operator returns canonicalized path names. Dots (.) and redundant slashes (/) are removed, unless a dot is the only remaining character.

Each double dot (..) cancels the path component to its left, unless that component is also a double dot. Double dots are moved to the front of the name.

Slash-double dot (/..) forms are preserved in deference to some remote file system implementations.

Example

```
HDR = ../t/../hdr/./hd_func.h
$(HDR:P=C)
```

This evaluates to: ../hdr/hd_func.h

P=D - Bound Directory Path

P=D - Bound Directory Path

Synopsis

$(S:P=D)

Description

This edit operator treats the tokens of the variable S as bound atoms and returns the directory name where the token was bound. Tokens that are unbound evaluate to null.

Example

Suppose there is a need to evaluate the directories where the prerequisite files of a target were found. If the target name is proc, the evaluation is given as:

$(!proc:P=D)

The automatic variable $(!proc) evaluates to the list of implicit and explicit prerequisites of the proc target.

P=H[*suffix*] - Hash Name

P=H[*suffix*] - Hash Name

Synopsis

$(S:P=H[=*suffix*])

Description

This edit operator generates a hash file name for the given token. The hash file name is 9 characters long unless *suffix* is used, in which case it is at most 14 characters long.

Example

This example illustrates the use of the P=H operator with and without *suffix*.

Contents of Makefile

```
LIST = a.c b.c a1.c
tst: $(LIST)
        : :P=H[=suffix] - $(*:P=H=.hashxxx)
        : :P=H - $(*:P=H)
```

Run nmake

```
$ nmake
```

Output

```
+ : :P=H[=suffix] - MA9EE2711.hash MAE109702.hash MA6C393EF.hash
+ : :P=H - MA9EE2711 MAE109702 MA6C393EF
```

P=I - Identical Name

P=I - Identical Name

Synopsis

$(S:P[!]=I=*name*)

Description

This edit operator selects those tokens of the variable S that have the identical device and inode number as *name*. Only those tokens of the variable S that are bound to existing files will have their device and inode numbers compared to those of *name*. The != operator selects all tokens bound to existing files that do not match the device and inode number of *name*.

The value of *name* must bind to a single existing file; otherwise null is expanded, and the edit operation continues.

Example

In the following example, assume x.c is linked to x1.c.

$(list) expands to x.c a.c x1.c.

$(list:P=I=x.c) expands to x.c x1.c.

$(list:P!=I=x.c) expands to a.c.

Contents of Makefile

```
list = x.c a.c x1.c
targ : $(list)
        : $(list:P=I=x.c)
        : $(list:P!=I=x.c)
```

Run nmake

```
$ nmake
```

Output

```
+ : x.c x1.c
+ : a.c
```

P *relop* L - Paths Bound in Level

P *relop* L - Paths Bound In Level

Synopsis

$([!]S:P<*relop*>L [=*level*] | [*|!])

Description

This edit operator selects tokens that are atoms bound in the *relop level* view. The view is given as one or more sets of product-level directory structures to use when looking for files. The current view is the current working directory, with L=0; this is the default. Each level represents a complete directory hierarchy to search.

For those tokens found down the viewpath, the full path to the files is returned. *level* can be any integer value, specified either as a literal integer or as the result of an integer evaluation.

The relational operators (*relop*s) are:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
!=	not equal to

L* returns all views of a bound file.

L! returns the first occurrence of a bound file from all the views.

Example

The following edit operation determines the list of prerequisite files for target procr that were found in any level other than the current view.

$(!procr:P>L=0)

P=P - Probe Path

P=P - Probe Path

Synopsis

$(S:P = P = *lang[,tool|,[tool],path]***)**

Description

This edit operator treats each token of the variable S as a processor for the language specified as *lang* and returns the path to the *probe* configuration file for that language processor.

If unspecified, *tool* defaults to *make*. Use *pp* for the cpp probe file. *path* indicates the root of an alternate probe hierarchy.

Example

Suppose that the full path to a C compiler is /bin/cc. If variable S has the value /bin/cc, the edit operation:

$(S:P=P=C)

evaluates to the full path of a file under the nmake installation directory. This file contains the information generated by the *probe* command. The contents of the file could be read into the current nmake invocation using the T=I edit operator, for example '$(S:P=P=C:T=I-)'. The quotes prevent shell interpretation. See the example for T=I. See Appendix A of the *Alcatel-Lucent nmake Product Builder User's Guide* for information on probe.

The following will give the path to the pp probe file for the current compiler.

$(CC:P=P=C,pp)

P=R - Relative Path

P=R - Relative Path

Synopsis

$(S:P = R = *name***)**

Description

This edit operator generates the relative path from each token of the variable S to the directory name given by *name*. The expanded value of *name* must be a single path. This edit operator is useful when generating the relative position between two directories, such as the current directory and the top of the product directory structure.

Example

Suppose the variable $(NODE) is set to the full path of the project root directory, and the current working directory is some subdirectory *n* levels deep. The edit operation

$(PWD:P=R=$(NODE))

evaluates to the relative path from the current directory to the top directory. The expanded name is canonicalized where possible and may contain a series of ../ components to represent the relative offset.

P=S - Bound Path

P=S - Bound Path

Synopsis

```
$(S:P=S)
```

Description

This edit operator expands the path to each token, viewed as an atom that is bound to a file. Those tokens not yet bound to files expand to null. Atoms that are bound to files in subdirectories are expanded as the relative location to the file. Atoms that are bound to files found down the viewpath are expanded to the full path to the file.

Example

The following example shows the relative path for the source files in the current view and the full path for other files. Assume new.c is the only file in the current view. The VPATH = /v1:/v2.

Contents of Makefile

```
FILES = main.c process.c new.c

target: $(FILES)
        : $(FILES:P=S)
```

Run nmake

```
$ nmake
```

Output

```
+ : /v2/main.c /v2/process.c new.c
```

P=U - Unbound Path

P=U - Unbound Name

Synopsis

```
$(S:P=U)
```

Description

This edit operator returns the unbound name for each token of the variable S. Tokens that are not bound atoms are returned unaltered.

Example

This example illustrates the difference between unbound names returned by the P=U operator and bound names returned by the T=F operator. Assume a directory structure that contains dir1 and dir2; dir1 contains a.c and dir2 contains b.c. The file c.c does not appear in the directories in .SOURCE or in the current directory.

Contents of Makefile

```
.SOURCE : dir1 dir2
FILES = a.c b.c c.c
target :
        : $(FILES:P=U)
        : $(FILES:T=F)
```

Run nmake

```
$ nmake
```

Output

```
+ : a.c b.c c.c
+ : dir1/a.c dir2/b.c
```

The P=U operator returns c.c because it is not a bound atom and thus is returned unaltered. See the discussion of the T=F operator, which returns bound names.

P=V - View Directory Path Name

P=V - View Directory Path Name

Synopsis

```
$(S:P=V)
```

Description

This edit operator returns the view directory path name of the current directory for each token bound to a file. Those tokens not yet bound to files expand to null**.**

Example

In this example, VPATH = /v1:/v2, a is in /v1, and b is in /v2.

Contents of Makefile

```
FILES = a b
target : $(FILES)
        : $(*:P=V)
```

Run nmake

```
$ nmake
```

Output

```
+ :. /v2
```

P=VL - View Local Path Name

P=VL - View Local Path Name

Synopsis

```
$(S:P=VL)
```

Description

This edit operator returns the view local path name for each token that is an atom bound to a file. Paths in the viewpath are canonicalized and converted to their corresponding local path relative to the current directory. The resulting local path need not exist in the top node. Paths outside the viewpath are returned unmodified. Unbound paths return null.

Example

This example shows the effect of applying :P=VL to a bound file path. In this example, VPATH = /v1:/v2; the current directory is /v1/src/abc. The following source files are in the viewpath nodes:

```
/v1/src/file1
/v2/include/file2
```

Contents of Makefile

```
.SOURCE : $(VROOT)/src $(VROOT)/include
:ALL: t1 t2
t1 : file1 .demo
t2 : file2 .demo
.demo : .USE
        : target $(<)
        : bound = $(*)
        : P=VL = $(*:P=VL)
```

Run nmake

```
$ nmake
```

Output

```
+ : target t1
+ : bound = ../../src/file1
+ : P=VL = ../file1
+ : target t2
+ : bound = /v2/include/file2
+ : P=VL = ../../include/file2
```

P=X - Unbound Existing File

P=X - Unbound Existing File

Synopsis

$(S:P=X)

Description

This edit operator returns each token that is a path to an existing file. The expanded list is not bound. Each token of the variable S is treated as a canonicalized path name. If there is an existing file with that name, the token value is returned; otherwise, null is returned.

Example

SRC = main.c process.c output.c

The evaluation

$(SRC:P=X)

expands to only those files found in the current directory.

To search the list of directories given by .SOURCE, the evaluation must be specified as

$(*.SOURCE:X=$(SRC):P=X)

The X Cross Product edit operator is used to generate the complete list of possible paths for each source file given by $(SRC).

Q - Quote

Q - Quote

Synopsis

$(S:[@]Q)

Description

This edit operator places quotation marks around each token so that a literal interpretation will be used by the shell. See the UNIX system sh(1) command in the *UNIX System V User's Reference Manual* for more information.

If the edit operator is preceded by the @ character, the value of the variable is treated as a single token.

Example

In this example, nmake is invoked three times, each time with a different makefile. The first time nmake is invoked, the action in the makefile results in a syntax error. To prevent this error from occurring, the makefile is modified. In the modified makefile, the value of LIST is quoted because a literal interpretation of the left parenthesis in (DEBUG) is necessary. The third version of the makefile demonstrates the use of the Q edit operator.

Contents of Makefile

```
LIST = a.c b.c (DEBUG)
tst:
        : echo $(LIST)
```

Run nmake

```
$ nmake
```

Output

```
ksh[33]: syntax error at line 2 :\ '(' unexpected
```

Contents of Makefile

```
LIST = a.c b.c (DEBUG)
tst:
        : echo '$(LIST)'
```

Run nmake again

```
$ nmake
```

Output

```
+ : echo a.c b.c (DEBUG)
```

Contents of Makefile

```
LIST = a.c b.c (DEBUG)
tst:
        : echo $(LIST:Q)
```

Run nmake a third time

```
$ nmake
```

Output

```
+ : echo a.c b.c (DEBUG)
```

R - Read Makefile

R - Read Makefile

Synopsis

$(S:R)

Description

This edit operator evaluates the value of the variable S as a single value and reads the string as if it were a makefile. Any variable definitions, assertions, programming constructs, etc., take effect immediately upon completion of the expansion. All assertions, variable names, and values from the existing makefile are available in the evaluation of S.

Example

In this example, a prerequisite of .MAIN cannot be determined until run time. Suppose the makefile contained the following lines:

```
DYNAMIC=$(TARGET):$(PREREQ) .IMPLICIT
eval
$(DYNAMIC:R)
end
```

Changing the values for the variables TARGET and PREREQ alters the makefile dynamically. For example, when TARGET is defined in the environment as main.o and PREREQ is defined as main.c, this makefile generates the assertion main.o:main.c .IMPLICIT.

T - Type

T - Type

Synopsis

$(S:T = [W | X] *type* **)**

Description

The tokens are bound to atoms (unless stated otherwise) and are operated on according to the type operator given as *type*. If T=*type* is true (non-null), the selected tokens are returned. Otherwise, null is returned.

If *type* is preceded by W, nmake does not wait for the completion of binding. If *type* is preceded by X, no binding is performed.

Remember that all *type* edit operators bind any tokens that are not yet bound; the files named by the T expansion must exist prior to the evaluation; otherwise, null is returned.

The *type* operators are described in the following manual pages.

T=A - Archive Update Action

T=A - Archive Update Action

Synopsis

$(S:T=A)

Description

This edit operator returns the archive update action for each token that is an existing file with the .ARCHIVE attribute. The archive update action is the command block that is to be used to bring the archive up-to-date after it has been modified or copied (i.e., ranlib).

This edit operator is seldom used, but has been included for completeness, specifically for users who write rules.

T=D - State Variable cpp Generator

T=D - State Variable cpp Generator

Synopsis

```
$(S:T=D)
```

Description

This edit operator returns the cc-style definition of each token that can be bound to a *state variable*. (See cc(1) in the *UNIX System V User's Reference Manual* for more information.)

The definition of CCFLAGS in the default base rules includes $(&:T=D). The automatic variable $(&) evaluates to the list of all explicit and implicit state variable prerequisites of the current target. The T=D edit operator expands the state variables to the -D*name*=*value* format definitions being passed to the C compiler.

Example

This example illustrates the use of the T=D edit operator.

Contents of Makefile

```
DEBUG == 5
MACHINE == 3b2
MSV = (DEBUG) (MACHINE)
CDEFS := $(MSV:T=D)
tst :
        : $(CDEFS)
```

Output

The above edit operation evaluates the variable CDEFS to have the value:

```
-DDEBUG=5 -DMACHINE=3b2
```

T=E - State Variable Name Value Pairs

T=E - State Variable Name Value Pairs

Synopsis

$(S:T=E)

Description

This edit operator expands the *name=value* pairs for tokens of the variable S that bind to state variables. Tokens not binding to state variables are expanded to null. The expanded *name=value* represents the name of the state variable along with its current value.

Exmaple

Suppose a makefile contains the following:

```
DEBUG == 5
MACHINE == 3b2
name.nl : (DEBUG) (MACHINE)
        echo $(?:T=E) > $(<)
```

Two state variables, DEBUG and MACHINE, are defined as having values 5 and 3b2, respectively. The default target to build is name.nl, which is dependent on the two state variables. The desired action for this target is to save the state variable names and values in the target file. The automatic variable $(?) is used to list all implicit and explicit prerequisites for the target. The T=E edit operator selects only those tokens that can be bound to state variables, or in this case, DEBUG and MACHINE. The result of this edit operator is to generate the *name=value* pairs; in other words, $(?:T=E) evaluates to DEBUG=5 MACHINE=3b2.

T=F - Bound File Name

T=F - Bound File Name

Synopsis

```
$(S:T[!]=F)
```

Description

This edit operator selects the tokens of the variable S that can bind, or are already bound to existing files. nmake attempts to search the directories of the .SOURCE or .SOURCE.*pattern* Special Atom to find the file named by the token value. If it is found, the token is bound and the token value is expanded. For tokens found down the viewpath, full paths to the files are returned; otherwise, relative paths to the files are returned. If a file cannot be found using the nmake file search algorithm, null is returned. This edit operator searches the viewpath for the named file. The form != can be used to expand the list of tokens that cannot be bound to files using the nmake file search algorithm.

Example

In this example, a.c, b.c, and c.c exist in the current directory.

Contents of Makefile

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

target :
        : $(FILES:T=F)
```

Run nmake

```
$ nmake
```

Output

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

T=G - Generated File Name

T=G - Generated File Name

Synopsis

$(S:T[!]=G)

Description

This edit operator selects tokens from variable S that can be generated from other prerequisites based on metarule patterns. The bound names of the tokens are returned. The form != operator can be used to expand the list of tokens that cannot be generated based on metarule patterns.

Example

A makefile containing the specification:

$(!targ:T=G)

evaluates to all existing prerequisites of targ that were generated using metarules. The following specification will provide this information for all the targets in the makefile (assuming the prerequisites of .MAIN include all the targets in the makefile):

$(!$(*.MAIN):T=G)

T=I - Token Include

T=I - Token Include

Synopsis

$(S:T=I[-])

Description

Each token of the variable S is the name of an existing file to read dynamically. The optional - prevents expansion of variable names appearing in the file. The results of this evaluation can be assigned to another variable or can be used to alter dynamically the existing makefile.

If the expansion of variable names from the file is not desired and if the results of this operation are to be assigned to a variable, take care to avoid unwanted expansion during the assignment process.

The token file name is sought according to the nmake file search algorithm. Thus, the file can be found down the viewpath. When the token file name cannot be found, nmake issues a warning that it cannot read the file.

Using this edit operator to assign values to variables dynamically or to alter existing makefile specifications can yield undesired results, since the file is actually included only one time. Whenever nmake generates *makefile*.mo, the token file is read and evaluated accordingly. However, if the file is modified after *makefile*.mo has been made up-to-date, the modification of the token file does not cause *makefile*.mo to be regenerated. Using this edit operator in the action block of a target that has the .FORCE attribute causes the action to occur.

The T=I edit operator can be very useful, but the user should take care to ensure that the desired evaluation is completed by nmake under all conditions.

Example

A makefile contains the following lines:

```
source_list:name.list .MAKE .FORCE .ALWAYS
    eval
            $(<)=$(*:T=I-)
    end
```

name.list is a file containing a list of source file names. The .MAKE attribute causes nmake to evaluate the action block instead of handing it off to the shell. The .FORCE and .ALWAYS attributes force the action to occur even if nmake thinks the target is up-to-date. In the action block, the eval...end programming statements are used to force nmake to evaluate the statement properly.

In the statement:

```
$(<)=$(*:T=I-)
```

$(<) is an automatic variable evaluating to the name of the current target. $(*:T=I-
) causes nmake to read the prerequisite atoms bound to files without expanding variable names within the file.

T=M - Parentage Chain

T=M - Parentage Chain

Synopsis

```
$(S:T=M)
```

Description

This edit operator returns all the parents of the variable S.

Example

In this example, $(<) represents the current target, x.

Contents of Makefile

```
a : b
b : x
x :
        : $(<:T=M)
```

Run nmake

```
$ nmake
```

Output

```
a b x
```

T=N - Null Variable Identification

T=N - Null Variable Identification

Synopsis

$(S:T=N)

Description

This edit operator identifies null variables. If an unbound value of *variable* is null, 1 is returned; otherwise, if the variable has a value, null is returned.

Example

List = $(List:T=N) evaluates to 1.

T=O - Token Written

T=O - Token Written

Synopsis

$(S:T=O[*mode*][=*text*])

Description

Each input token is the name of the file to be written according to the specified *mode.* If no *mode* is specified, *text* overrides the contents of each file. By default, the newline separator is attached to the end of the *text.* This edit operator is the complement to T=I[–]. *mode* can be:

+	Appends *text.*
-	Does not put the newline separator at the end of *text*.

Example

In the following example, the contents of infile is read in using the T=I edit operator, modified by changing input to output, and then saved in outfile.

After nmake is invoked, outfile is created in the current working directory.

Contents of infile

```
 echo "This is an input file"
```

Contents of Makefile

```
IFILES = infile
OFILE = outfile
t:
        : $(OFILE:T=O=$(IFILES:T=I:C/input/output/)
```

Run nmake

```
$ nmake
```

Contents of outfile

```
 echo "This is an output file"
```

T=P - Bound Files Without Links

T=P - Bound Files Without Links

Synopsis

```
$(S:T=P)
```

Description

This edit operator selects the tokens of the variable S that are bound, or could bind, to existing files that are not symbolic links. The bound atom name is returned. If symbolic links are not implemented, T=P is equivalent to T=F.

Example

In the following example, file.c is a regular file and link.c is a symbolic link to some other file.

Contents of Makefile

```
FILES= link.c file.c
target:
        : $(FILES:T=P)
```

Run nmake

```
$ nmake
```

Output

```
+ : file.c
```

T=Q - Query for Existing Rule Atoms

T=Q - Query for Existing Rule Atoms

Synopsis

```
$(S:T[!]=Q)
```

Description

This edit operator selects tokens from the variable S that are existing rule atoms. The form != can be used to select tokens that are not rule atoms.

Example

a.c, t1, and t2 are atoms of the assertions in this Makefile. b.c is not an atom in any of the rules in this Makefile.

Contents of Makefile

```
FILES = a.c b.c t1 t2

t1 :
        : $(FILES)
        : $(FILES:T=Q)
        : $(FILES:T!=Q)

t2 : a.c
```

Run nmake

```
$ nmake
```

Output

```
+ : a.c b.c t1 t2
+ : a.c t1 t2
+ : b.c
```

T=QV - Query for Defined Variables

T=QV - Query for Defined Variables

Synopsis

$(S:T=QV)

Description

This edit operator returns a defined variable.

Example

Contents of Makefile

```
LIST= A B C
A==1
B= b.c
T :
        : $(LIST:T=QV)
```

Run nmake

```
$nmake
```

Output

```
: A B
```

T=R - Relative Time

T=R - Relative Time

Synopsis

$(S:T=R)
$("":T=R)

Description

This edit operator treats each token of the variable S as an atom and returns the relative time (the number of seconds since Epoch [00:00:00 GMT on January 1, 1970]) of each atom. This operation causes the tokens to be bound; any tokens that cannot be bound evaluate to null.

Typically, the T=R edit operator is used in conjunction with the F edit operator to generate a readable time stamp for the atoms.

The second form of the synopsis, $("":T=R), operates on the current date.

Example

In the following example, both targets x and a are made because :ALL: is asserted. x.c includes x.h, which includes y.h.

In the definition of :ALL: contained in the default base rules (Makerules.mk), the prerequisite list of .ALL, i.e., $(*.ALL), is defined; this list is x a in this example. The target .DONE has the attributes .MAKE, .ALWAYS, and .FORCE, which cause nmake to force the action to be triggered and to evaluate the action instead of giving it to the shell. The action of the .DONE target is to list the prerequisites of .ALL along with their epoch time and to list all the targets made in this makefile and their file prerequisites along with their time stamps. $(!$(*.ALL)) is the file prerequisites list of $(*.ALL).

Contents of Makefile

```
.SOURCE: ../inc ../include
:ALL:
x :: x.c
a :: a.c
.DONE: .MAKE .ALWAYS .FORCE
        for i $(*.ALL)
            print $(i)'s epoch time is $("x":T=R)
        end
                for i $(*.ALL) $(!$(*.ALL))
                        print $(i)'s time stamp is $(i:T=R:F=%T)
        end
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I../inc \
-I../include -I- -c x.c
+ cc -O -o x x.o
+ cc -O -Qpath /nmake3.0/lib -I-D/nmake3.0/\
lib/probe/C/pp/361FBDCFrucbcc -I- -c a.c
+ cc -O -o a a.o
x’s epoch time is 885432534
a’s epoch time is 885432534
x’s time stamp is Jan 21 20:28:54 1998
a’s time stamp is Jan 21 20:28:54 1998
x.o’s time stamp is Jan 21 20:28:54 1998
x.c’s time stamp is Jan 21 20:28:54 1998
../inc/x.h’s time stamp is Jan 21 20:28:54 1998
../include/y.h’s time stamp is Jan 21 20:28:54 1998
a.o’s time stamp is Jan 21 20:28:54 1998
a.c’s time stamp is Jan 21 20:28:54 1998
```

The following example, which illustrates the second form in the synopsis, was run on June 17, 2008.

Contents of Makefile

```
t :
        : $("":T=R:F=%(%a %m/%d/%y %H:%M:%S %Z)T)
```

Run nmake

```
$ nmake
```

Output

```
: Tue 06-17-08 11:03:15 EDT
```

T=S - Name Conversion

T=S - Name Conversion

Synopsis

$(S:T=S[F][*conv*[*data*]])

Description

This edit operator treats each token of the variable S as an atom and converts each atom to a new atom type based on the specified *conv* and *data*. null is returned when the conversion specifies an undefined atom. F forces the atom to be made. The conversion operators (*conv*) that can be specified are listed on the following pages.

T=SA - State Atom name Conversion

T=SA - State Atom Name Conversion

Synopsis

$(S:T=SA*data***)**

Description

This edit operator converts the specified internal name for a state rule to the original name.

Example

See T=SR.

This edit operator is seldom used but has been included for completeness, specifically for users who write rules.

T=SF - Force new Atom Creation

T=SF - Force New Atom Creation

Synopsis

$(S:T=SF*data*)

Description

This edit operator is used to create a new atom and is used in conjunction with other conversion operators.

Example

See T=SV.

T=SM - Select Metarule Name

T=SM - Select Metarule Name

Synopsis

$(S:T=SM*data*)

Description

This edit operator returns the metarule name that generates files matching the metarule pattern *data* from the files matching the metarule patterns named by each token. This edit operator is useful in determining which metarule would be applied to build a target from a set of prerequisites.

Example

The following example shows the use of the T=SM edit operator.

Contents of Makefile

```
.SOURCE: src
FILES = main.c x.s
t :
        : $(FILES:B=%:S:T=SM%.o:Q)
```

Run nmake

```
$ nmake
```

Output

```
+ : %.c>%.o %.s>%.o
```

The output shows the metarules nmake would apply to generate corresponding object files (i.e., main.o x.o). The Path edit operators B and S are applied to each token of the variable, substituting the % for the basename component and keeping the suffix component. The T=SM%.o instructs nmake to evaluate which metarule defines how to transform the tokens into .o targets. The Q causes the literal interpretation of the > character in the metarule format.

T=SP - Alternate Prerequisite State Atom

T=SP - Alternate Prerequisite State Atom

Synopsis

$(S:T=SP*data*)

Description

For tokens that are both implicit and explicit atoms, this edit operator returns the internal name for the state rule containing the alternate binding information.

Example

In this example, x.c includes x.h, which is generated in this makefile. Thus, x.h is both an implicit and explicit atom in this makefile. After x is generated, nmake internally stores two sets of binding information for x.h. $("x.h":T=SR) returns the state atom name containing the primary binding information (i.e., ()x.h); $("x.h":T=SP) returns the name of the rule containing the alternate binding information (i.e., (+)x.h). This information can also be seen in the query output. The assertion .DONE: .QUERY is to put nmake into the query mode. Entering x.h at the make> prompt will display all the binding information for x.h. To exit query mode, enter q followed by a carriage return.

This edit operator is seldom used, but has been included for completeness, specifically for users who write rules.

Contents of Makefile

```
x :: x.c
x.h:
        > $(<)
tst:
        : '$("x.h":T=SP)'
.DONE: .QUERY
```

T=SR - Primary State Atom Name

T=SR - Primary State Atom Name

Synopsis

$(S:T=SR*data*)

Description

This edit operator returns the primary state atom name that contains the primary bounding information.

Example

The following edit operation returns the state atom name for the state rule associated with pax.o (i.e., ()pax.o). The Q is to put quotes around the output to prevent shell interpretation.

$("pax.o":T=SR:Q)

The following edit operation converts the output of the above operation back to the original name (i.e., pax.o).

("$("pax.o":T=SR)":T=SA)

This edit operator is seldom used, but has been included for completeness, specifically for users who write rules.

T=SV - State Variable Name

T=SV - State Variable Name

Synopsis

$(S:T=SV*data*)

Description

Given a variable name, this edit operator returns the corresponding state variable name.

Example

The following example returns the state variable name for the debug variable (i.e., (debug)).

$("debug":T=SFV)

The F edit operator forces creation of the debug state variable if it does not exist.

Since the output has special characters (the two parentheses), it must be quoted to avoid shell interpretation.

This edit operator is seldom used, but has been included for completeness, specifically for users who write rules.

T=T - Time Comparison

T=T - Time Comparison

Synopsis

$(S:T*relop***T***atom***)**

Description

This edit operator is used to compare the time of *atom* with the times associated with the atoms in the token list. The tokens whose times are *relop atom* are returned. If *atom* is not specified, it defaults to the current target.

The relational operators (*relop*s) are:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
!=	not equal to

Example

The edit operation:

$(*:T>T)

lists the prerequisites whose times are newer than the time for the current target.

T=U - State Atom

T=U - State Atom

Synopsis

$(S:T=U)

Description

This edit operator returns the atom or variable name for each state atom token.

Example

The state atom name of the atom a.c is ()a.c; it is stored in the *makefile*.ms file. Supposing that a.c is not in the current directory, $("()a.c":T=U) returns the relative path of a.c if viewpathing is not active, the full path of a.c if it is.

T=V - Variable Value Identification

T=V - Variable Value Identification

Synopsis

$(S:T=V)

Description

This edit operator returns 1 if the unbound value of the variable is non-null; otherwise it returns null.

Example

List =$(List:T=V) evaluates to the null string.

T=W*type* - Don’t Wait for Bind

T=W*type* - Don't Wait For Bind

Synopsis

$(S:T=W*type*)

Description

If a *type* operator is forcing a file to be bound and the W operator is applied, do not wait for the bind to complete. The W must appear after the = operator but before *type*.

T=X*type* - Skip Bind

T=X*type* - Skip Bind

Synopsis

$(S:T=X*type*)

Description

This edit operator prevents the *type* operator from forcing binding to occur. The X must appear after the = operator but before *type*.

T=Z - Time

T=Z - Time

Synopsis

$(S:T=Z[CERS])

Description

This edit operator returns the input token's cancellation time, event time, relative time, or state time. The event time is the time that nmake finds the token. The relative time is the current rule time. The state time is the last modification time of the token, which is recorded in the state file.

Example

In the following example, there is no cancellation time, so the output from the T=ZC operator indicates the Epoch time.

Contents of Makefile

```
FILE = a.c
t : $(FILE)
        : $(*:T=ZC:F=%(%a %m-%d-%y %H:%M:%S)T)
        : $(*:T=ZE:F=%(%a %m-%d-%y %H:%M:%S)T)
        : $(*:T=ZR:F=%(%a %m-%d-%y %H:%M:%S)T)
        : $(*:T=ZS:F=%(%a %m-%d-%y %H:%M:%S)T)
```

Run nmake

```
$nmake
```

Output

```
+ : Wed 03-11-98 19:00:00
+ : Fri 04-24-98 10:29:33
+ : Fri 04-24-98 10:29:33
+ : Thu 04-23-98 14:43:25
```

V - Value

V - Value

Synopsis

$(S:V)

Description

This edit operator uses the value of the variable S as a single value and returns that value without expansion. This operator must appear before all other edit operators.

Example

VAR = xxxx
RESULT = $(VAR)
$(RESULT:V)

Normally, the expansion of the variable VAR would be xxxx. However, in this example, the V edit operator inhibits expansion and the string $(VAR) is returned.

VA - Auxiliary Value

VA - Auxiliary Value

Synopsis

$(S:VA)

Description

This edit operator returns the auxiliary value of the variable S. Every variable has a primary value (assigned by the =, :=, or += assignment operator) and may have an auxiliary value (assigned by the &= assignment operator). The auxiliary value is not saved in the statefile.

$(S:VA) returns variable values without expansion.

Example

```
X = dummy
X &= dummy2
```

$(X:VA) expands to dummy2. (See example under VP for more information.)

VP - Primary Value

VP - Primary Value

Synopsis

$(S:VP)

Description

This edit operator returns the primary value of the variable S. The primary value of a variable is assigned by the =, :=, or += assignment operator. The auxiliary value is assigned by the &= operator.

$(S:VP) returns variable values without expansion.

Example

```
X = primary1
X &= aux1
Y = $(X)
print $X is $(X)
print $X:VP is $(X:VP), $X:VA is $(X:VA)
X += primary2
print $X is $(X)
print $X:VP is $(X:VP), $X:VA is $(X:VA)
X &= aux2
print $X is $(X)
print $X:VP is $(X:VP), $X:VA is $(X:VA)
print $Y is $(Y)
print $Y:VP is $(Y:VP), $Y:VA is $(Y:VA)
targ :
```

Output

```
$X is primary1 aux1
$X:VP is primary1, $X:VA is aux1
$X is primary1 primary2 aux1
$X:VP is primary1 primary2, $X:VA is aux1
$X is primary1 primary2 aux2
$X:VP is primary1 primary2, $X:VA is aux2
$Y is primary1 primary2 aux2
$Y:VP is $(X), $Y:VA is
```

The += assignment operator appends to the primary value. The &= assignment operator overwrites the auxiliary value every time it is used; the auxiliary value is not stored in the statefile.

X - Cross Product

X - Cross Product

Synopsis

$(S:X=*cross*)

Description

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

The first dot (.) on the left-hand side of the operand produces a dot in the cross product. All right-hand side absolute path operands are collected in order after all other products have been computed, regardless of the operands on the left-hand side.

Example

This example shows how the editing operator X matches each token in DIRS with each token in FILES.

Contents of Makefile

```
DIRS = dir1 dir2 dir3
FILES = a.c b.c c.c
target :
        : $(DIRS:X=$(FILES))
```

Run nmake

```
$ nmake
```

Output

```
dir1/a.c dir2/a.c dir3/a.c dir1/b.c dir2/b.c dir3/b.c \
dir1/c.c dir2/c.c dir3/c.c
```

Y - Yield

Y - Yield

Synopsis

$(S:Y *<del> non-null <del> null <del>***)**

Description

This edit operator evaluates the values of each token in the variable S. If that value is non-null, *non-null* is expanded and returned. Otherwise, *null* is expanded and returned.

<del> can be any delimiter character.

The *non-null* and/or the *null* action can be null (empty), but the proper number of delimiters must be present. The value of S can be from a previous edit operation chained to the :Y edit operator. The value of *non-null* and the value of *null* can be given as any valid variable expansion. Therefore, Y edit operations can nest, but the user should take care to ensure proper evaluation under all conditions.

A special form of this edit operator is **$(S:Y** *<del> <del> null <del>***O)** which is equivalent to **$(S:Y** *<del> $(S)<del> null <del>***).** This form is particularly useful when the value of S is obtained from a non-trivial chaining of previous edit operations chained to the :Y edit operator.

Y? can be abbreviated as ?, where ? is the delimiter.

Example 1

$(MACHINE:M=m3b2[01]:?m3b2.o? default.o?)

If the MACHINE variable matches the regular expression m3b2[01], the value is non-null, and the string m3b2.o is returned. If the pattern does not match, the value is null, so the string default.o is returned.

Example 2

Without the special form of the :Y edit operator with the **O** option, one would have to evaluate the following expressions like this (Note, the repeat evaluation of the *non-null* part);

$(VAR:P=A:N=*.c:D:N=*/src/*:Y?$(VAR:P=A:N=*.c:D:N=*/src/*)?not-src-dir?)

However, with the **O** option of the :Y operator, the above would simply be,

```
$(VAR:P=A:N=*.c:D:N=*/src/*:Y??not-src-dir?O)
```

Special Atoms

6

This section contains manual pages for all the Special Atoms. Special Atoms can be used to control the disposition of default base rules and targets in a makefile. All Special Atoms defined by nmake have the form *.ID*, where *ID* is any string of capital letters, dots, or numbers. Special Atoms are categorized by type, depending upon the function they perform. The nine types are:

- Action Rule Atoms
- Assertion Attribute Atoms
- Dynamic Attribute Atoms
- Dynamic List Atoms
- Immediate Rule Atoms
- Pattern Association Atoms
- Readonly Attribute Atoms
- Readonly List Atoms
- Sequence Atoms

Some Special Atoms may be of more than one type; for example, there is a .MAKE Dynamic Attribute atom and a .MAKE Immediate Rule Atom. In these cases, the manual pages include the type name in the name of the manual page.

The following sections briefly list and describe the atoms of each type.

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. Action rule atoms are normally placed on the right-hand side of an assertion.

Listed below are the action rule atoms and a brief description of them.

.ACTIONWRAP	If defined, the .ACTIONWRAP rule action is expanded in place of each shell action prior to command execution.
.QUERY	.QUERY places nmake in an interactive debugging loop. Input entered at the make> prompt can be any valid makefile input. However, parsing read-ahead requires that a blank line follow an interactive assertion before it takes effect. The interactive loop is exited by pressing the CTRL and D keys simultaneously from the keyboard. The assertion .INTERRUPT : .QUERY causes the debug loop to be entered on interrupts.

Assertion Attribute Atoms

Assertion attribute atoms provide fine control over dependency operator assertions. They are used only to control assertions and are not associated with other atoms. Assertion attribute atoms are normally placed on the right-hand side of an assertion.

Listed below are the assertion attribute atoms along with a brief description of each.

.CLEAR	Clears the attributes, prerequisites, and action for the targets in the assertion.
.INSERT	Causes the prerequisites to be inserted before rather than appended to the target prerequisite list.
.NULL	Assigns the null action to the target atoms. Used when an action must be present, usually to force source files with prerequisites to be accepted.
.SPECIAL	Target atoms in the assertion are not appended to the prerequisites of .MAIN; multiple action diagnostics are inhibited.

Dynamic Attribute Atoms

Dynamic attribute atoms are assigned to target atoms by listing attribute names as prerequisites in assertions. This capability allows atoms to be classified by giving attributes to certain atoms. For example, all C source files could be given a unique attribute so that they would be recognized by nmake for a particular task. Dynamic attribute atoms are normally placed on the right-hand side of an assertion.

An attribute is added as an assertion by typing one of the following:

target **:** *.ATTRIBUTE*
target **:** *+ATTRIBUTE*

It is negated by typing:

target **:** *-ATTRIBUTE*

Dynamic attribute atoms can be tested using the A edit operator.

Listed below are the dynamic attribute atoms along with a brief description of each.

.ACCEPT	Only file prerequisites can make .ACCEPT atoms out-of-date.
.AFTER	.AFTER prerequisites are made after the action for the target atom has completed.
.ALWAYS	The noexec option inhibits the execution of shell actions. However, shell actions for .ALWAYS targets are executed even with noexec on. If an action in an assertion contains the sequence $(MAKE), the targets are automatically assigned the .ALWAYS attribute. $(MAKE:) can be used to defeat this feature.
.ARCHIVE	Target atoms with the .ARCHIVE attribute are treated as archives (see ar(1)). Binding a .ARCHIVE target atom also binds the archive members to the target. .ARCHIVE can be used as a pseudosuffix for .APPEND and .INSERT, which allows all targets with this attribute to have common prerequisites.
.ATTRIBUTE	Marks the target as a named attribute. Named attributes can be tested using the A edit operator. A maximum of 32 named attributes can be defined. The prerequisites of .ATTRIBUTE constitute the list of all named attributes.
.BATCH	Marks each job that is currently compiling a batch of files.
.BATCHED	Marks targets that are currently batched for compilation and not yet up-to-date.
.BEFORE	.BEFORE prerequisites are made just before the action for the target atom is executed.

.COMMAND	Marks target atoms that bind to executable command files. .COMMAND can be used as a pseudosuffix for .APPEND and .INSERT, which allows all targets with this attribute to have common prerequisites.
.DONTCARE	If a .DONTCARE target cannot be made, nmake continues as if it existed; otherwise an error is issued and nmake either discontinues work on the target parent and siblings if keepgoing is on, or it terminates processing and exits.
.FORCE	Atoms with this attribute are always out-of-date the first time they are made during a single nmake execution.
.FOREGROUND	The target action is blocked until all other actions have completed. Normally nmake makes future prerequisites while concurrent actions are being executed; however, a .FOREGROUND target causes nmake to wait until the corresponding action completes.
.FUNCTIONAL	A functional atom is associated with a variable of the same name. Each time the variable is expanded, the corresponding atom is made before the variable value is determined. The atom action can compute and assign a value to the variable.
.IGNORE	Prevents any parent targets from becoming out-of-date with the target, even if the target has just been built. This capability allows initialization sequences to be specified for individual atoms.
.IMMEDIATE	An atom with this attribute is made immediately after each assertion on the atom.
.IMPLICIT	This target attribute causes the implicit metarules to be applied even if an action or prerequisites have been specified for the target. Otherwise, the implicit metarules are applied only to targets with no explicit actions and prerequisites. This attribute turns the .TERMINAL attribute off at assertion time.
.JOINT	The action causes all targets on the left-hand side to be jointly built with respect to the prerequisites on the right-hand side.
.LOCAL	Assignment of .LOCAL as an attribute assures that coshell will be able to distribute the processing of the $(MAKE) command.
.MAKE	Causes the target action to be read by nmake rather than executed by the shell. Such actions are always read, even with noexec on.

.MULTIPLE	Normally each assertion removes duplicate prerequisites from the end of the target atom prerequisites list. .MULTIPLE atoms are allowed to appear more than once in a prerequisite list.
.OPERATOR	This attribute marks the target as an operator to be applied when reading makefiles. Assertion operator names must match either :: or :*identifier*:.
.PARAMETER	State variables with the .PARAMETER attribute are not expanded by the T=D and T=E edit operators.
.READ	The standard output of the action is read and interpreted as nmake statements.
.REPEAT	By default, an atom is made at most once per nmake invocation. .REPEAT marks atoms that are to be made repeatedly. .FORCE is required to force the action to be triggered each time the atom is made.
.SCAN	Marks an atom as defining scanning rules for a product-specific language.
.SCAN.*x*	Marks an atom (when bound to a file) to be scanned for implicit prerequisites using the *.x* scan strategy. .ATTRIBUTE.*pattern* : .SCAN.*x* is asserted for each of the strategies listed below, which are defined by default.
.SCAN.c	Scan for C #include file prerequisites and candidate state variable references. Include statements requiring macro substitutions are not recognized. " " include files are bound using the directories of .SOURCE.c first and then the directories of .SOURCE and .SOURCE.h. This order is the opposite of the default directory binding order. < > include files are bound using only the directories of .SOURCE.h. " " include files are assigned the .LCL.INCLUDE attribute; < > include files are assigned the .STD.INCLUDE attribute. .LCL.INCLUDE takes precedence if a file appears as both " " and < >. A warning is issued if a file appears as both " " and < >. To support popular C coding style, if a file is included as #include "*prefix*/file1" (where *prefix* represents at least one level of a directory structure), and if file1 includes another file as #include "file2", nmake first looks in *prefix* for file2. If it is not found there, nmake uses .SOURCE.h, .SOURCE.c, and .SOURCE, respectively to locate file2.
.SCAN.f	Generic SQL include file prerequisites or FORTRAN include and INCLUDE file prerequisites.
.SCAN.F	A combination of the .SCAN.sql, .SCAN.f, and elements of the .SCAN.c strategy.

.SCAN.idl	Scan for IDL #include file prerequisites and candidate state variable references. Include statements requiring macro substitutions are not recognized. " " include files are bound using the directories of .SOURCE.idl first and then .SOURCE. < > include files are bound using only the directories of .SOURCE.idl. " " include files are assigned the .LCL.INCLUDE.idl attribute; < > include files are assigned the .STD.INCLUDE.idl attribute. To support popular coding style, if a file is included as #include "*prefix*/file1" (where *prefix* represents at least one level of a directory structure), and if file1 includes another file as #include "file2", nmake first looks in *prefix* for file2. If it is not found there, nmake uses .SOURCE.idl and .SOURCE respectively to locate file2.
.SCAN.r	RATFOR include and INCLUDE file prerequisites. Include statements requiring macro substitutions are not recognized.
.SCAN.m4	m4(1) include( ), sinclude( ), and INCLUDE( ) file prerequisites. The INCLUDE( ) form also handles S (statistical analysis package) source file prerequisites. Include statements resulting from macro substitutions are not recognized.
.SCAN.nroff	nroff(1) and troff(1) .so file prerequisites. .so file is only recognized when it begins a line. .so statements requiring macro substitutions are not recognized.
.SCAN.sh	Scans for candidate state variable references in shell scripts, i.e., those state variables initialized with the == operator.
.SCAN.sql	SQL include file prerequisites of the form EXEC SQL include *filename* or #include *filename* where *filename* is "*file*", *file*;, or *file*.
.SCAN.IGNORE	Inhibits scanning. Used to override .ATTRIBUTE.*pattern* scan strategies.
.SCAN.NULL	Inhibits scanning. Used to override .ATTRIBUTE.*pattern* scan strategies.
.SCAN.STATE	Marks candidate state variables for scanning.
.SEMAPHORE	Limits the number of executing shell actions for target atoms having the same .SEMAPHORE prerequisite. Each .SEMAPHORE within an assertion increments the semaphore count by 1. The maximum semaphore count is 7.

.TERMINAL	This attribute allows only .TERMINAL metarules to be applied to the target atom when determining metarule prerequisites. Normally, metarules are applied to targets with no explicit actions and prerequisites, so using .TERMINAL makes the behavior more restrictive. This attribute turns the .IMPLICIT attribute off at assertion time. For metarule assertions, .TERMINAL marks unconstrained metarules that apply only to .TERMINAL targets or nongenerated source files. For directories, this attribute causes nmake to mark each directory as having no subdirectories. This attribute is used to optimize processing when there are many directories that do not have subdirectories and many files with directory prefixes.
.USE	Marks the target as a .USE atom. Any target having a .USE atom as a prerequisite will be built using the .USE atom action and attributes. The left most .USE prerequisite takes precedence.
.VIRTUAL	A .VIRTUAL atom never binds to a file. .VIRTUAL atom times are saved in the statefile.

Dynamic List Atoms

Dynamic list atom prerequisites control nmake actions.

Depending on how they are used, dynamic list atoms may appear on either the left- or the right-hand side of an assertion. Listed below are the dynamic list atoms along with a brief description of each.

.ARGS	The prerequisites of .ARGS are the command-line target arguments. If, after making the .INIT sequence atom, .ARGS has no prerequisites, the prerequisites of .MAIN are copied to .ARGS. As each prerequisite of .ARGS is made, it is removed from the .ARGS prerequisite list.
target.BATCH	Lists the queued targets for batch job *target*. See the .BATCH and .BATCHED dynamic attributes.
.EXPORT	The prerequisites are treated as variable names to be included in $(=) automatic variable expansions.

.MAIN	If, after the .INIT target has been made, .ARGS has no prerequisites then the prerequisites of .MAIN are appended onto .ARGS. If not explicitly asserted in the input makefiles then the first prerequisite of .MAIN is set to be the first target in the input makefile that is not marked as .FUNCTIONAL, .OPERATOR, or .REPEAT, and is neither a state variable nor metarule.
.SOURCE	The prerequisites of .SOURCE are directories to be scanned when searching for files. The (left-to-right) directory order is important. The first directory containing the file is used. The current directory (.) is always searched first.
–	This attribute is used to control synchronization when making a list of prerequisites. When – appears in a prerequisite list, nmake waits until the preceding prerequisite's actions are complete before continuing.

Immediate Rule Atoms

An immediate rule atom invokes an nmake action when it appears as a target in assertions at assertion time. The prerequisites and actions are cleared after the assertion. When they are cleared, they are removed internally, as if the atom never had prerequisites or actions associated with it. Immediate rule atoms are normally placed on the left-hand side of an assertion.

Listed below are the immediate rule atoms along with a brief description of each.

.ACCEPT	The prerequisite atoms are accepted as being up to date.
.ALARM	.ALARM sets an alarm signal (in seconds) during nmake execution.
.BIND	The prerequisites are bound.
.IMMEDIATE	The prerequisites and action are made each time this atom is asserted.
.MAKE	The prerequisites and action are made each time this atom is asserted.
.QUERY	Full atom and state information is listed to stderr for each prerequisite.
.REBIND	Each prerequisite is touched as if it had already been made.
.RETAIN	The prerequisites are variable names whose values are to be retained in the statefile.

.STATE	The prerequisites are variable names that are marked as candidate implicit state variable prerequisites.
.SYNC	This atom is used for synchronizing the statefile.
.UNBIND	Each prerequisite is unbound as if it had not been bound. Directories are removed from the internal directory cache.

Pattern Association Atoms

Pattern association atoms are used to associate patterns with atoms. *pattern* can be any one of the following:

- %.*x* where *x* can be any suffix. (% is a wild-card character (called the stem) and stands for all characters.)
- Any of the provided Attributes. Attributes are properties of an atom.
- User-defined Named Attributes.

Pattern association atoms are normally placed on the left-hand side of an assertion. Listed below are the pattern association atoms along with a brief description of each.

.APPEND.*pattern*	The prerequisites of .APPEND.*pattern* are appended to the prerequisite list of each target atom with suffix .*pattern* immediately before the target prerequisites are made. Note that .*pattern* must be %.ARCHIVE for .ARCHIVE targets and %.COMMAND for .COMMAND targets.
.ATTRIBUTE.*pattern*	When an atom with suffix .*pattern* is bound, it inherits the attributes of .ATTRIBUTE.*pattern*.
.BIND.*pattern*	Specifies binding rules to be used when an atom cannot be bound using the normal rules. The action must set .BIND to the name of a file to which the atom should be bound.
.INSERT.*pattern*	The prerequisites of .INSERT.*pattern* are inserted into the prerequisite list of each target atom with suffix .*pattern* immediately before the target prerequisites are made. Note that .*pattern* must be .ARCHIVE for .ARCHIVE targets and .COMMAND for .COMMAND targets.

.NOCROSSPROCUCT.*pattern*	Suppress the cross product expansion of *pattern* from the prerequisite list of .SOURCE.*pattern2.* Specified as an attribute on the RHS of .SOURCE.*pattern2.*
.REQUIRE.*pattern*	Modifies the binding algorithm to allow a bound atom to map to a list of bound atoms.
.SOURCE.*pattern*	The prerequisites of .SOURCE.*pattern* are directories to be scanned when searching for files with suffix *.pattern*. If the file is not found, the directories specified by the prerequisites of .SOURCE are checked. The (left-to-right) directory order is important. The first directory containing the file is used. The directory . is always searched first. The implicit dependency scan strategy (see .SCAN) may augment or override the default .SOURCE search for individual atoms.

Readonly Attribute Atoms

Readonly attribute atoms are maintained by nmake; they cannot be explicitly assigned. They are used to mark internal atom states for use by nmake. The most common use of readonly attribute atoms is in conjunction with the A edit operator.

Listed below are the readonly attribute atoms along with a brief description of each.

.ACTIVE	Marks atoms that are currently being made, i.e., currently active
.BOUND	Marks bound prerequisites.
.BUILT	Marks targets that have been built for storage in the statefile.
.ENTRIES	Marks scanned directories or archives that have entries (members).
.EXISTS	Marks targets that have been successfully made.
.FAILED	Marks targets that have been unsuccessfully made.
.FILE	Marks prerequisites bound to existing files.
.MAKING	Marks each target whose (shell) action is executing.
.MEMBER	Marks atoms that are members of a bound archive atom.
.NOTYET	Marks targets that have not been made.
.REGULAR	Marks prerequisites that are bound to regular files.
.SCANNED	Marks archive and directory atoms that have been scanned for members.

.STATE	Marks atoms that have either the .STATEVAR or the .STATERULE attribute.
.STATERULE	Marks internal state rule atoms that are used to store state information.
.STATEVAR	Marks state variable atoms.
.TARGET	Marks atoms that appeared as the target of an assertion.
.TRIGGERED	Marks atoms whose actions have been triggered during the current nmake execution.

Readonly List Atoms

Readonly list atoms contain prerequisites maintained by nmake. These atoms cannot be used as targets in assertions.

Listed below are the readonly list atoms along with a brief description of each.

.GLOBALFILES	The prerequisites are the list of global makefiles specified by the global option.
.INTERNAL	This atom is used internally and appears here for completeness.
.MAKEFILES	The prerequisites are the list of makefiles specified by the file option.
.METARULE	The ordered list of left-hand side metarule patterns for all asserted metarules excluding the % match-all metarules. This list determines the metarule application order.
.OPTIONS	The prerequisites are the options declared by the option option.
.TMPLIST	This atom is used internally and appears here for completeness.
.VIEW	The prerequisites are view directory paths. By default, the current directory (.) is the first view directory. A view directory is a directory from which nmake can be executed. .VIEW is initialized from the MAKEPATH and VPATH environment variables.

Sequence Atoms

Sequence atoms are made at specific times during nmake execution. Sequence atom shell actions are always executed in the foreground. To avoid attribute or base and global rule clashes, use intermediate rules (with actions) in assertions, and append these to the sequence atom prerequisite list. Sequence atoms are normally placed on the left-hand side of an assertion, so that a given atom's prerequisites and the associated actions will be made at a specific time.

Listed below are the sequence atoms along with a brief description of each.

.COMPDONE	The action of .COMPDONE is executed immediately after the makefile is compiled.
.COMPINIT	This atom is made when the input makefiles are (re)compiled, just before the make object file is written.
.DONE	This is the last user-level atom made before nmake terminates execution; the last atom is .MAKEDONE.
.ERROR	This atom is executed when an error is encountered and the compile option is off. If the .ERROR make action block returns 0 (default), control returns to the point after the error. If the action block returns –1, the error processing continues.
.INIT	Made after the .MAKEINIT sequence atom.
.INTERRUPT	This atom is made when an interrupt signal is caught. The state of nmake may become corrupted by actions triggered while .INTERRUPT is active. If the interrupt occurs while .QUERY is being made and set nointerrupt is executed by .INTERRUPT after .INTERRUPT is made, nmake continues from the point where the interrupt occurred; otherwise, nmake exits with a nonzero status.
.INTERRUPT <*signal*>	This atom is used to handle a specified <*signal*>, where <*signal*> is one of the symbolic names, as in signal(5). Usage assumes uppercase names without the SIG prefix.
.MAKEDONE	This atom is made after .DONE and just before the statefile is updated.
.MAKEINIT	This atom is made just after the statefile has been loaded. The base rules typically use this atom to initialize the nmake engine. ⚠ **CAUTION:** *Do not redefine the* .MAKEINIT *action or attributes. However, it is safe to insert or append prerequisites to* .MAKEINIT.

.ACCEPT (Dynamic Attribute) Special Atom

.ACCEPT - accept atom as up-to-date

Type

Dynamic Attribute

Description

Only changes to file prerequisites (not statefile inconsistencies) can make .ACCEPT atoms out-of-date.

Example

In the makefile below, the .ACCEPT atom is included in the list of actions for the prereq1 file. When the nmake command is first run, all targets are made, including prereq1.

After the touch command is run to change the time-stamp on the prereq1 and prereq2 files, the nmake command is run again. At that time, the .ACCEPT atom prevents the remaking of the prereq1 target, because only the time-stamp has changed.

Contents of Makefile

```
all : target1 target2                   /* define the target all */

target1 : prereq1                       /* define target1 */
        : making $(<)
        > $(<)

prereq1 : prereq3 .ACCEPT               /* define prereq1 */
        : making $(<)
        > $(<)

target2 : prereq2                       /* define target2 */
        : making $(<)
        > $(<)

prereq2 :                               /* define prereq2 */
        : making $(<)
        > $(<)

prereq3 :                               /* define prereq3 */
        : making $(<)
        > $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ : making prereq3
+ 1> prereq3
+ : making prereq1
+ 1> prereq1
+ : making target1
+ 1> target1
+ : making prereq2
+ 1> prereq2
+ : making target2
+ 1> target2
```

Touch prereq1 and prereq2

```
$ touch prereq1 prereq2
```

Run nmake again

```
$ nmake
```

Output

```
+ : making target1
+ 1> target1
+ : making prereq2
+ 1> prereq2
+ : making target2
+ 1> target2
```

Touch prereq3

```
$ touch prereq3
```

Run nmake again

```
$ nmake
```

Output

```
+ : making prereq3
+ 1> prereq3
+ : making prereq1
+ 1> prereq1
+ : making target1
+ 1> target1
```

.ACCEPT (Immediate Rule) Special Atom

.ACCEPT - accept prerequisite atoms as up-to-date

Type

Immediate Rule

Description

The prerequisite atoms are accepted as being up to date. After .ACCEPT, the atom is not rebuilt even if its filesystem timestamp is older than the timestamp of its prerequisites. However, subsequent file prerequisites may make the atoms out of date.

Example

In the makefile below, myrule is a .MAKE rule which is made before the targets, targ1 and targ2. myrule uses .ACCEPT with targ1 to accept targ1 as up to date. When the makefile is built only targ2 is generated since targ1 is considered up to date. Even touching targ1 and its prerequisite do not trigger it. More sophisticated makefiles might use .ACCEPT in a similar fashion but only under certain conditions.

Contents of Makefile

```
:ALL: myrule - targ1 targ2

myrule : .MAKE .VIRTUAL .FORCE
        .ACCEPT : targ1

targ1 : prereq1
        touch $(<)

targ2 : prereq2
```

Run nmake

```
$ nmake
```

Output

```
+ touch targ2
```

The next example illustrates the use with a user defined attribute. The attribute is a prerequisite to target targ1. Using .ACCEPT with the attribute allows the attribute to be added and removed from the targ1 prerequisite list without triggering an update. This may be useful for certain attributes that are dynamically added or removed during a build.

Contents of Makefile

```
.Z : .ATTRIBUTE
.ACCEPT : .Z

targ1 : .Z
        touch $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ touch targ1
```

Contents of Makefile after removing the attribute

```
.Z : .ATTRIBUTE
.ACCEPT : .Z

targ1 :
        touch $(<)
```

Run nmake

```
$ nmake
```

Output

```
[ empty ]
```

Contents of Makefile after adding the attribute back

```
.Z : .ATTRIBUTE
.ACCEPT : .Z

targ1 : .Z
        touch $(<)
```

Run nmake

```
$ nmake
```

Output

```
[ empty ]
```

.ACTIONWRAP Special Atom

.ACTIONWRAP - rule action expanded in place of each shell action.

Type

Action Rule Attribute

Description

If defined, the .ACTIONWRAP rule action is expanded in place of each shell action prior to command execution. The expansion takes place in the original rule context so that automatic variables refer to the original rule values. In particular, $(@) expands to the action for the original rule.

This allows easy dynamic modification of all rules in an nmake execution. Note that the operation of the actionwrap rule can test the target attributes and vary the action modification accordingly.

Example

The following example causes each rule to be dynamically modified so that a message containing the target name and current time is printed before and after the execution of the original rule. In addition, the starting message includes the word ``command'' if the rule target has the .COMMAND attribute. The .ACTIONWRAP is conditionally defined based upon the value of variable modify_rules.

Contents of Makefile

```
if modify_rules
        .ACTIONWRAP :
                : "starting $(<:A=.COMMAND:?command ??)target $(<) at `silent date`"
                $(@)
                : "target $(<) completed at `silent date`"
end
a :: a.c
```

Run nmake to illustrate unmodified rule output

```
$ nmake
```

Output

```
+ ppcc -i /home/porsche/gms/wrk/nmake/312/solaris/lib/cpp cc -O -I-D/home/porsche/gms/wrk/
nmake/312/solaris/lib/probe/C/pp/BFE0A4FEobincc -I- -c a.c
+ ppcc -i /home/porsche/gms/wrk/nmake/312/solaris/lib/cpp cc -O -o a a.o
```

Run nmake to illustrate modified rule output

```
$ nmake modify_rules=1
```

Output

```
+ : starting target a.o at Wed Mar 24 10:58:45 EST 1999
+ ppcc -i /home/porsche/gms/wrk/nmake/312/solaris/lib/cpp cc -O -I-D/home/porsche/gms/wrk/
nmake/312/solaris/lib/probe/C/pp/BFE0A4FEobincc -I- -c a.c
+ : target a.o completed at Wed Mar 24 10:58:45 EST 1999
+ : starting command target a at Wed Mar 24 10:58:45 EST 1999
+ ppcc -i /home/porsche/gms/wrk/nmake/312/solaris/lib/cpp cc -O -o a a.o
+ : target a completed at Wed Mar 24 10:58:46 EST 1999
```

.ACTIVE Special Atom

.ACTIVE - Marks atoms that are currently being made

Type

Readonly Attribute

Description

This Special Atom marks each target that is being made, i.e., currently active. The attribute is assigned to the target for the duration it takes to make it and its prerequisites. This atom is similar to the .MAKING Special Atom, with the exception that the .MAKING attribute is assigned to the target for the duration of the shell action turnaround time (i.e., from the time the shell action has been sent to the shell until the shell action returns to nmake).

Example

The following example shows the assignment of the .ACTIVE attribute for targets made from shell or make action blocks and its contrasting behavior to the .MAKING atom. Note that the targets made from a make action block does not have the .MAKING attribute.

Contents of Makefile

```
X = x y z
U = u v w
a : $(X) - $(U)
$(X) :
        : $(<) : $(X:A=.MAKING:@/.*/& has .MAKING/)
        : $(<) : $(X:A=.ACTIVE:@/.*/& has .ACTIVE/)
        : $(<) : $(<<:A=.MAKING:@/.*/parent & has .MAKING/)
        : $(<) : $(<<:A=.ACTIVE:@/.*/parent & has .ACTIVE/)
        silent echo -------------------

$(U) : .MAKE
        print $(<) : $(U:A=.MAKING:@/.*/& has .MAKING/)
        print $(<) : $(U:A=.ACTIVE:@/.*/& has .ACTIVE/)
        print $(<) : $(<<:A=.MAKING:@/.*/parent & has .MAKING/)
        print $(<) : $(<<:A=.ACTIVE:@/.*/parent & has .ACTIVE/)
```

Run nmake

```
$ nmake a
```

Output

```
+ : x : x has .MAKING
+ : x : x has .ACTIVE
+ : x :
+ : x : parent a has .ACTIVE
-------------------
+ : y : y has .MAKING
+ : y : y has .ACTIVE
+ : y :
+ : y : parent a has .ACTIVE
-------------------
+ : z : z has .MAKING
+ : z : z has .ACTIVE
+ : z :
+ : z : parent a has .ACTIVE
-------------------
u :
u : u has .ACTIVE
u :
u : parent a has .ACTIVE
v :
v : v has .ACTIVE
v :
v : parent a has .ACTIVE
w :
w : w has .ACTIVE
w :
w : parent a has .ACTIVE
```

.AFTER Special Atom

.AFTER - make atom after parent target atom's action

Type

Dynamic Attribute

Description

Prerequisite atoms with the .AFTER Special Atom are made after the action for the parent target atom has completed.

Example

The following example demonstrates the effect of using the .AFTER Special Atom. The first time nmake is run, the target afterthought is made first; the second time, after the .AFTER attribute has been associated with it, afterthought is made after the actions for the target file have been completed.

Contents of Makefile

```
afterthought :
        : the target $(<) was made

file : afterthought
        : making $(<)
```

Run nmake

```
$ nmake file
```

Output

```
+ : the target afterthought was made
+ : making file
```

Add the .AFTER atom as a prerequisite

```
afterthought : .AFTER
```

Run nmake again

```
$ nmake file
```

Output

```
+ : making file
+ : the target afterthought was made
```

.ALARM Special Atom

.ALARM - set alarm signal

Type

Immediate Rule

Description

.ALARM sets an alarm signal (in seconds) during nmake execution. For example:

```
.ALARM : 15
```

sets an alarm signal 15 seconds after nmake execution.

Interrupt handling can be specified for the alarm signal. The value of $(.ALARM) is the absolue time to the next alarm; a value of 0 indicates no alarm.

Example

Contents of Makefile

```
.ALARM : 5

.INTERRUPT.ALRM : .MAKE
        error 3 Alarm received, exiting...

DATE : .FORCE .REPEAT .FUNCTIONAL
        set +x
        date
        sleep 1

printdate : .MAKE
        for I   1 2 3 4 5 6 7 8 9 10
                print $(DATE)
        end
```

Run nmake

```
$ nmake
```

Output

```
Mon Jun 15 11:45:45 EDT 1998
Mon Jun 15 11:45:46 EDT 1998
Mon Jun 15 11:45:47 EDT 1998
Mon Jun 15 11:45:48 EDT 1998
make: Alarm received, exiting...
```

.ALWAYS Special Atom

.ALWAYS - always execute shell action for target atom

Type

Dynamic Attribute

Description

The -n command-line option inhibits the execution of shell actions. However, shell actions for .ALWAYS targets are executed even when the -n option is used.

Example

In the following example, when nmake is run with the -n command-line option, target1 is not made, but target2 is, because it has the .ALWAYS Special Atom. Neither an object file nor a statefile is generated.

When nmake is run on the same makefile without the -n command-line option, both targets are made and both an object file and a statefile are generated.

In the output listing, commands that are denoted by +> are actions echoed by the shell; commands that are denoted by +1> are actions that are executed by the shell.

Content of Makefile

```
all : target1 target2              /* define the target all */

target1:                           /* define the target target1 */
        > $(<)

target2: .ALWAYS                   /* define the target target2 */
        > $(<)
```

Run nmake with noexec on

```
$ nmake -n
```

Output

```
+ > target1
+1 > target2
```

Created files

```
target2
```

Remove target2

```
$ rm target2
```

Run nmake again without any options

```
$ nmake
```

Output

```
+1 > target1
+1 > target2
```

Created files

```
makefile.mo
makefile.ms
target1
target2
```

.APPEND.*pattern* Special Atom

.APPEND.*pattern* - append prerequisites to target atoms that match specified attributes

Type

Pattern Association

Description

The prerequisites of .APPEND.*pattern* are appended to the prerequisite list of each target atom with suffix *.pattern* immediately before the target prerequisites are made.

Note that *.pattern* must be %.ARCHIVE for .ARCHIVE targets and %.COMMAND for .COMMAND targets.

Example

The following example appends id.o to the .ARCHIVE target, a.a, causing id.o to be included in a.a automatically.

Contents of Makefile

```
.APPEND.%.ARCHIVE: id.o
a.a :: a.c
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -c a.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -c id.c
+ ar r a.a a.o id.o
+ ignore ranlib a.a
+ rm -f a.o id.o
```

.ARCHIVE Special Atom

.ARCHIVE - treat target atom as archive

Type

Dynamic Attribute

Description

This Special Atom causes the file suffix types to be treated as archives. It can also be used to cause the target to be treated as an archive. The .ARCHIVE.o rule is used to update the target.

Binding an ARCHIVE target atom also binds the archive members to the target.

See ar(4) in the *UNIX System V User’s Reference Manual* for more information about archive format.

Examples

In the following example, .ARCHIVE is used to specify that all files with a .x suffix be treated as archives. Because the target liba.x has the .x suffix, it is generated as an archive.

Contents of Makefile

```
.ATTRIBUTE.%.x : .ARCHIVE
liba.x :: sub1.c sub2.c
```

Run nmake

```
$ nmake
```

Output

```
+ 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
+ ar r liba.x sub1.o sub2.o
ar: creating liba.x
+ ignore ranlib liba.x
+ rm -f sub1.o sub2.o
```

In the following example, the .ARCHIVE Special Atom marks the target liba.y as an archive.

Contents of Makefile

```
liba.y :: sub3.c .ARCHIVE
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -c sub3.c
+ ar r liba.y sub3.o
ar: creating liba.y
+ ignore ranlib liba.y
+ rm -f sub3.o
```

.ARGS Special Atom

.ARGS - an atom whose prerequisites contain target arguments specified on the command-line

Type

Dynamic List

Description

The prerequisites of .ARGS are the command-line target arguments. If, after making the .INIT sequence atom, .ARGS has no prerequisites, the prerequisites listed in .MAIN are copied to .ARGS. These prerequisites are then made and removed from the .ARGS list. If .MAIN does not exist, no action is taken.

Example

The items from the .ARGS list (target arguments on the command-line) are removed as they are made. The following example shows how to retain the initial value of .ARGS. When .MYARGS is made, the value of $(.ARGS) is frozen into the prerequisites of .MYARGS.

Contents of Makefile

```
.INIT     : .MYARGS               /* define the targets to be made */
            : targets to be made are \"$(*.ARGS)\"
            : command-line targets were\"$(*.MYARGS)\"

.MYARGS: $$(*.ARGS)

target0   :          /* define the actions for the command-line targets */
            : 0 making target $(<)
target1   :
            : 1 making target $(<)
target2   :
            : 2 making target $(<)

.DONE :              /* show the contents of $(*.ARGS) and $(*.MYARGS) */
            : targets to be made were\"$(*.ARGS)\"
            : targets to be made were\"$(*.MYARGS)\"
```

Run nmake

```
$ nmake target0 target1
```

Output

```
+ : 0 making target target0
+ : 1 making target target1
+ : targets to be made are "target0 target1"
+ : command-line targets were "target0 target1"
+ : targets to be made were ""
+ : targets to be made were "target0 target1"
```

The last two lines of output show that the original value of $(*.ARGS) is contained in $(*.MYARGS) and the value of $(.ARGS) has been cleared.

.ATTRIBUTE Special Atom

.ATTRIBUTE - mark target atom as user-defined attribute

Type

Dynamic Attribute

Description

This Special Atom defines the target as a user attribute. User attributes should be of the form *.ID*, where *ID* is any string of lower-case or mixed-case letters, dots, or numbers. Attributes can be tested using the A edit operator.

Example

The following example shows how a user-defined attribute can be used to select members from a predefined list of files. When nmake is run, a.g and c.c are selected, because they are in both the files list and the myattr list.

Contents of Makefile

```
files = a.g b.g c.c d.p             /* give a list of values to the variable
                                                files */

.myattr : .ATTRIBUTE                /* define myattr as a user attribute */

a.g c.c d.c : .myattr               /* define the files that make up myattr */

target : $(files)                   /* define the target */
        : $(files:A=.myattr)
```

Run nmake

```
$ nmake target
```

Output

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

.ATTRIBUTE.*pattern* Special Atom

.ATTRIBUTE.*pattern* - indicate inheritance of attributes

Type

Pattern Association

Description

When an atom that matches .*pattern* is bound, it inherits the prerequisite attributes of .ATTRIBUTE.*pattern*.

Example

The following nmake statement means that when any file that matches the pattern %.pq (i.e., any file with the suffix .pq) is bound, it inherits the .SCAN.sql attribute. (% is a wild-card character (called the *stem*) that stands for all characters.)

Define Attribute

```
.ATTRIBUTE.%.pq : .SCAN.sql
```

.BATCH / .BATCHED Special Atoms

.BATCH - marks each job that is currently compiling a batch of files.
.BATCHED - marks targets that are currently batched for compilation and
not yet up-to-date.
target.BATCH - lists the queued targets for batch job *target*.

Type

Dynamic Attributes - .BATCH, .BATCHED
Dynamic List - *target*.BATCH

Description

.BATCH and .BATCHED are used in conjunction to allow safe concurrent builds for batched file generation, in particular the batched Java compilation feature. When a Java file is first encountered and the batch queue is not yet full, its class target is touched and added to a list of files for a batch compile. Later, when the queue fills up, the batch is compiled all at once. .BATCH and .BATCHED are used to prevent nmake from building dependent targets when the class target is first touched (before the batch is completed.)

.BATCHED marks targets that are on a queue that has not yet triggered, or has triggered and not yet completed building. When searching for jobs to run, nmake will not consider jobs that depend on a .BATCHED job. These jobs are blocked from running. After the queue fills up, the batch job is triggered. The target that caused the queue to fill is marked .BATCH, and the queue contents are stored as prerequisites of *target*.BATCH. .BATCH marks each job that is compiling a batch of files (each of which is marked .BATCHED). When a .BATCH job completes, nmake clears the .BATCH attribute for *target*, and for each prerequisite of *target*.BATCH it clears the .BATCHED attribute and accepts that target. This finally allows the blocked dependents jobs on the queue to run.

Example

In this example, target depa.class is dependent on a.class. In the first run we don't use .BATCH/.BATCHED and depa.class gets triggered immediately after a.class is first touched, before the queued compile, which is not what we want. The second run, with .BATCH/.BATCHED, does not trigger depa.class until after the queued compile is complete.

Contents of Makefile

```
all: depa.class a.class b.class c.class

a.class : a.java .ADDQUEUE          /* queue a.gen for later compilation */
        touch a.class               /* compilation is delayed until queue is full */

b.class : b.java .ADDQUEUE          /* queue b.gen for later compilation */
        touch b.class               /* compilation is delayed until queue is full */

c.class : c.java .FILLQUEUE                 /* finish queue */
        javac $(QUEUE:B:S=*.java)           /* perform the batched compile */

depa.class : depa.java a.class
        javac $(*:N=*.java)

.ADDQUEUE : .MAKE .BEFORE .FORCE .REPEAT .VIRTUAL
        QUEUE += $(<<)     /* queue for later compilation */
        if enablebatch
                print marking $(<<) .BATCHED
                $(<<) : .BATCHED
        end

.FILLQUEUE : .MAKE .BEFORE .FORCE .REPEAT .VIRTUAL
        QUEUE += $(<<)     /* queue is full */
        if enablebatch
                print marking $(<<) .BATCH
                $(<<) : .BATCH
                $(<<).BATCH : $(QUEUE)
        end
```

Run nmake without .BATCH/.BATCHED

```
$ nmake -j2
```

Output

```
+ touch a.class
+ touch b.class
+ javac depa.java
+ javac a.java b.java c.java
```

Run nmake with .BATCH/.BATCHED

```
$ nmake -j2 enablebatch=1
```

Output

```
marking a.class .BATCHED
marking b.class .BATCHED
+ touch a.class
+ touch b.class
marking c.class .BATCH
+ javac a.java b.java c.java
+ javac depa.java
```

.BEFORE Special Atom

.BEFORE - make atom just before parent target atom's action

Type

Dynamic Attribute

Description

Prerequisites with the .BEFORE Special Atom are made just before the action for the target atom is executed.

Example

In the following example, the order in which targets are made depends on the use of the .BEFORE Special Atom.

The first time nmake is run, the action for the prerequisite chkbefore is executed before the action for the target variable targ is triggered; then the target atom all action is performed.

After the .BEFORE Special Atom is added to the makefile, nmake is run again, and this time the variable targ action is triggered first, the chkbefore action is performed immediately before the target atom action, and the target atom all action is last.

Contents of Makefile

```
targ = a                    /* assign file name a to the variable $(targ) */

chkbefore :                 /* define chkbefore action */
        : 2 doing $(<)

all : chkbefore $(targ)             /* define target atom all action */
        : 1 doing $(<)

$(targ) :: a.c              /* define target $(targ) */
```

Run nmake

```
$ nmake all
```

Output

```
+ : 2 doing chkbefore
+ 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
+ : 1 doing all
```

Add .BEFORE Special Atom as a prerequisite of chkbefore

```
chkbefore : .BEFORE
: 2 doing $(<)
```

Touch a.c

```
$ touch a.c
```

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
+ : 2 doing chkbefore
+ : 1 doing all
```

.BIND Special Atom

.BIND - force binding of prerequisite atoms

Type

Immediate Rule

Description

This Special Atom causes all prerequisite atoms to be bound to the rules used to generate the prerequisites. It also causes variables to be bound to their values.

This Special Atom can be used to force binding to occur at a specific time.

Example

The following example shows how a variable can be explicitly bound to its value. The output shows that VAR1 has been bound but VAR2 has not. The .BOUND atom displays bound atoms.

Contents of Makefile

```
.BIND : VAR1                    /* assign the .BIND atom to VAR1 */

T = VAR1                        /* assign values to the variables */
U = VAR2

target :                        /* define target */
        : "$(T:A=.BOUND)"
        : "$(U:A=.BOUND)"
```

Run nmake

```
$ nmake
```

Output

```
+ : VAR1
+ :
```

.BIND.*pattern* Special Atom

.BIND.*pattern* - specify default binding rule for atoms based on *pattern*

Type

Pattern Association

Description

This atom is used to specify a default binding rule to be used when an atom cannot be bound using the normal rules. .BIND.*pattern* is typically marked with the .FUNCTION Special Atom (defined in Makerules.mk), and its action block returns the name of the file to which the atom should be bound.

Example

.BIND.-l% is used in the default Makerules.mk file to convert library specifications to their actual names, e.g., it maps -lx to libx.a.

.BOUND Special Atom

.BOUND - marks atom that is bound

Type

Readonly Attribute

Description

This Special Atom marks bound atoms. The following types of atoms can be bound.

- State variables can be bound to variables.
- A rule atom can be bound to a file (i.e., the path to the file is determined and the file's time-stamp is saved).
- Pattern association atoms are bound to the atoms matching the pattern (i.e., .ATTRIBUTE.%.X : .IGNORE).

Example

This atom is not commonly used in user makefiles. It is documented here only for completeness.

.BUILT Special Atom

.BUILT - marks atom that is built

Type

Readonly Attribute

Description

This 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.

Example

The state atoms ()a.o and ()b.o in the following example are tested to see if they have been built.

Contents of Makefile

```
VAR = ()a.o                    /* assign the state rule atoms to variables */
VAR1 = ()b.o

all : a target                 /* define the targets all, a, b, and 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 is a target created by this nmake 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.

.CLEAR Special Atom

.CLEAR - clear definitions from previous rules

Type

Assertion Attribute

Description

This Special Atom removes the current definitions of the attributes, prerequisite list, and action for the targets in the assertion. If a rule is not cleared, rewriting it in a makefile that is read at a later time (such as a developer's makefile) appends the new prerequisites/attributes to the existing list, instead of creating a new list. It will execute the atom read at a later time.

Example

The following example shows how to clear specified definitions.

The definition of build in the global makefile

```
build: x y z
```

Include a line similar to this in the global makefile to clear specified definitions

```
%.o : %.c .CLEAR
```

Clear the global definition for build in the local makefile

```
build : .CLEAR
```

.COMMAND Special Atom

.COMMAND - mark target atoms that bind to executable command files

Type

Dynamic Attribute

Description

This Special Atom marks target atoms that bind to executable command files. The .COMMAND Special Atom is automatically given to targets generated with the source dependency assertion operator (::).

Example

Normally, the common action clean removes all the generated intermediate files after nmake execution. The .COMMAND atom is used in the following example to mark main.o as an executable command file to prevent its removal.

Contents of Makefile

```
OBJECTS = main.o          /* assign file main.o to variable OBJECTS */

prog : $(OBJECTS) .COMMAND.o              /* define target prog */
$(OBJECTS) : .COMMAND
```

Run nmake

```
$ nmake
```

Output

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

Run nmake

```
$ nmake clean
```

Output

```
+ ignore rm -rf
```

NOTE:
The atom .COMMAND.o is defined in the default base rules (Makerule.mk). It is a .USE rule used to link the target.

.COMPDONE Special Atom

.COMPDONE - atom made immediately after makefile is compiled

Type

Sequence

Description

The action of .COMPDONE is executed immediately after the makefile is compiled. .COMPDONE is skipped if the compiled makefile (*.mo) has not been changed.

Example

The following example illustrates the use of .COMPDONE. The first time nmake is run, the action for .COMPDONE is executed, because the makefile has been compiled. The second time nmake is run, the action for .COMPDONE is not executed, because a recompilation of the makefile was not required.

Contents of Makefile

```
MESSAGE = making $(<) action

.COMPDONE :
        : $(MESSAGE)

target :
        : $(MESSAGE)
```

Run nmake

```
$ nmake
```

Output

```
+ : making .COMPDONE action
+ : making target action
```

Run nmake again

```
$ nmake
```

Output

```
+ : making target action
```

.COMPINIT Special Atom

.COMPINIT - atom made before makefile is compiled

Type

Sequence

Description

This atom is made if the makefile is out-of-date with the object file (i.e., if there is no .mo file), if the makefile has been touched, or if a frozen variable has changed. Making this atom allows special actions to be performed at makefile compile time.

Example

In the following example, nmake is run with the same makefile three times:

- The first time, .COMPINIT is made because no .mo file exists yet.
- The second time, .COMPINIT is not made because nothing has changed and the .mo file exists.
- The third time, .COMPINIT is made because the value of VAR2 that had been frozen in the .mo file has been changed. The -e option prints an explanation for makefile actions on stdout.

Contents of Makefile

```
VAR1 := $(VAR2)                 /* assign variable $(VAR2) to VAR1 */

.COMPINIT :                     /* include atom .COMPINIT as a target */
        : making .COMPINIT

target :                        /* define target target */
        : making target
```

Run nmake

```
$ nmake
```

Output

```
+ : making .COMPINIT
+ : making target
```

Run nmake

```
$ nmake
```

Output

```
+ : making target
```

Run nmake

```
$ nmake VAR2=100 -e
```

Output

```
make: warning: frozen command argument variable VAR2 changed
+ : making .COMPINIT
+ : making target
```

.DONE Special Atom

.DONE - atom made after all targets

Type

Sequence

Description

This atom is made after all 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.

This rule is usually placed toward the end of the makefile. The commands in the action block for this atom are always executed in the foreground shell.

Example

The following example demonstrates the effect of the .DONE atom on the order of processing.

Contents of Makefile

```
.DONE :                                 /* define the .DONE action */
        : Processing has finished.

nexttolast :                            /* define the target nexttolast */
        : Made before .DONE
```

Run nmake

```
$ nmake
```

Output

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

.DONTCARE Special Atom

.DONTCARE - continue if target atom cannot be made

Type

Dynamic Attribute

Description

This Special Atom causes nmake to continue even if the target atom cannot be made.

If the .DONTCARE Special Atom is not used and nmake cannot make a target, nmake will issue an error message and either terminate processing and exit, or, if the command-line option keepgoing is on, discontinue work on the parent target.

Example

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

In this example, nmake is run twice, once with and once 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 */
```

Remove the statefile and compiled makefile

```
rm Makefile.ms Makefile.mo
```

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
```

.ENTRIES Special Atom

.ENTRIES - marks atom that is either a directory or archive and contains members

Type

Readonly Attribute

Description

This Special Atom marks scanned directory or archive atoms that have entries (members). An archive atom is bound to an archive file.

Example

In the following example, nmake is run twice on the same makefile. After the first run, no files are listed as a result of the .ENTRIES Special Atom, because libgg.a does not exist and bb.a has no members. After the second run, libgg.a is listed, because it has members, but bb.a is not, because it still has no members.

Contents of Makefile

```
LIBS                              /* assign files libgg.a and bb.a
                                  to the variable */
LIBS = libgg.a bb.a

:ALL:                             /* define the targets */

bb.a :: .DONTCARE

libgg.a :: hh.c

.DONE :                           /* define the action for .DONE
        : $(LIBS:A=.ENTRIES)      to list archives with members */
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -c hh.c
+ ar r libgg.a hh.o
ar: creating libgg.a
+ ignore ranlib libgg.a
+ rm -f hh.o
+ :
```

Run nmake

```
$ nmake
```

Output

```
+ : libgg.a
```

.ERROR Special Atom

.ERROR - atom made upon encountering error conditions

Type

Sequence

Description

This Special Atom is executed when an error is encountered and the compile option, -c, is off. The return code of the make action block controls whether nmake continues processing. If the make action block returns 0 (the default), control returns to the point after the error; if the make action block returns -1, error processing continues and nmake exits.

Example

In the following example, when the target1 shell action is executed, cp detects an invalid number of arguments and exits with a code of 2. This triggers the .ERROR action, which writes a message, and continues building the remaining target.

Define the targets

```
targets : target1 target2               /* define the targets */

target1 :                   /* Bad target: missing arguments to cp */
        cp

target2 :
        : $(<)

.ERROR : .MAKE                          /* define the .ERROR action */
        error 0 An error has occurred...continuing
        return 0
```

Run nmake

```
$ nmake
```

Output

```
+ cp
Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
make: *** exit code 1 making target1
An error has occurred...continuing
+ : target2
```

.EXISTS Special Atom

.EXISTS - marks an atom or target that exists

Type

Readonly Attribute

Description

This Special Atom marks atoms or implicit targets that have been successfully made or that already exist from a previous invocation of nmake.

Example

In the following example, the targets have the following status:

- prog0 is listed as a target, but its prerequisite file, nowhere.c, does not exist
- prog1 is successfully made
- prog2 fails to be made because of a syntax error
- prog3 does not exist and is not a target
- prog4 exists but is not a target.

The list created with the .EXISTS Special Atom shows those targets that have been successfully made.

Contents of Makefile

```
PROGS = prog0 prog1 prog2 prog3 prog4/* assign files to PROGS */

:ALL:                                  /* define the targets */

prog0 :: nowhere.c .DONTCARE
prog1 :: prog1.c
prog2 :: prog2.c

.DONE :                                /* define the action for .DONE to
                                       list targets successfully made */
        : $(PROGS:A=.EXISTS)
```

Run nmake

```
$ nmake
```

Output

```
make: warning: don’t know how to make .ALL : prog0 : nowhere.o : nowhere.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c prog1.c
+ cc -O -o prog1 prog1.o
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c prog2.c
+ cc -O -o prog2 prog2.o
+ : prog1 prog2
```

.EXPORT Special Atom

.EXPORT - atom whose prerequisites are always treated as command-line variable assignments

Type

Dynamic List

Description

The prerequisites listed are treated as variable names to be included in $(=) automatic variable expansions.

Example

In the following example, the makefile in the current directory includes the .EXPORT atom so that VAR1 is always expanded. A second makefile in dir1 is executed from the original makefile. nmake is run twice, the second time with a variable assignment on the command-line.

Contents of Makefile in the current directory

```
.EXPORT : VAR1                  /* assign VAR1 to .EXPORT */

VAR1 = 100                      /* assign a value to VAR1 */

target :                        /* define the target */
        cd dir1
        $(MAKE) $(=)
```

Contents of Makefile in the dir1 directory

```
targ :                          /* define the target */
        : in dir1
        : $(=)
```

Run nmake

```
$ nmake
```

Output

```
+ cd dir1
+ nmake VAR1=100
+ : in dir1
+ : VAR1=100
```

Run nmake again

```
$ nmake VAR2=200
```

Output

```
+ cd dir1
+ nmake VAR2=200 VAR1=100
+ : in dir1
+ : VAR2=200 VAR1=100
```

.FAILED Special Atom

.FAILED - marks atoms whose action failed

Type

Readonly Attribute

Description

This Special Atom marks atoms for which the action execution failed. nmake does not attempt to make the atom again during the current execution.

Example

The following example lists target atoms for which execution failed. The targets have the following status:

- prog0 is listed as a target but never made
- prog1 is successfully made
- prog2 fails to be made because of a syntax error.

nmake is run with the -k flag to cause processing to continue even if an error occurs

Contents of Makefile

```
PROGS = prog0 prog1 prog2          /* assign the list of files to PROGS */

:ALL: prog2                        /* define the targets prog0, prog1, and prog2 */

prog0 :: .DONTCARE nowhere.c
prog1 :: prog1.c
prog2 : prog2.c
        cp

.DONE :                            /* define the action for .DONE to list
                                   targets that failed nmake execution */
        : $(PROGS:A=.FAILED)
```

Run nmake

```
$ nmake -k
```

Output

```
make: warning: don’t know how to make .ALL : prog0 : nowhere.o : \
nowhere.c
+ cp
Usage: cp [-ip] f1 f2; or: cp [-ipr] f1 ... fn d2
make: *** exit code 1 making target1
An error has occurred...continuing
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c prog1.c
+ cc -O -o prog1 prog1.o
+ : prog2
```

prog0 is not listed as failed because no attempt was made to make it.

.FILE Special Atom

.FILE - marks atom bound to file

Type

Readonly Attribute

Description

This Special Atom marks atoms bound to existing files.

Example

The following example shows that the nmake binding process occurs only when an assertion is made. In the example, hello.c is not recognized by nmake as an existing file since it is not part of an assertion. To nmake, hello.c is simply a string.

Contents of hello.c

```
main()
{
        printf("Hello\n");
}
```

Contents of program.c

```
main()
{
}
```

Contents of Makefile

```
FILES = program program.o program.c hello.c

program :: program.c

.DONE :
        : $(FILES:A=.FILE)
```

Run nmake

```
$ nmake
```

Output

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

.FORCE Special Atom

.FORCE - treat target atom as out-of-date

Type

Dynamic Attribute

Description

This Special Atom forces nmake to treat the target as out-of-date, even if it is up-to-date. Targets with this Special Atom are always made. If a target must be made repeatedly during a single nmake execution, the .REPEAT Special Atom should also be used.

When used with .SOURCE.*pattern*, .FORCE inhibits the automatic use of .SOURCE and .SOURCE.*suffix* for file searches. For example, Scanrules.mk defines the following:

```
.SOURCE.%.STD.INCLUDE : .FORCE $$(*.SOURCE.h)
```

With .FORCE specified nmake will not search the .SOURCE directories for standard include files.

Example

In the following example, targets t1 and t3.dep are always out-of-date because each has the .FORCE atom as a prerequisite. They are made whether or not their prerequisites have been updated. Target t3 is also out-of-date, because it includes a prerequisite target (t3.dep) that has .FORCE as a prerequisite as well. The .FORCE atom in target t1 has no effect on the prerequisite t1.dep.

Contents of Makefile

```
all : t1 t2 t3                /* assign the list of targets to all */

t1 : .FORCE t1.dep            /* define targets */
        touch $(<)
t2 :
        touch $(<)
t3 : t3.dep
        touch $(<)

t1.dep :
        touch $(<)
t3.dep : .FORCE
        touch $(<)
```

Run nmake

```
$ nmake all
```

Output

```
+ touch t1.dep
+ touch t1
+ touch t2
+ touch t3.dep
+ touch t3
```

Run nmake again

```
$ nmake all
```

Output

```
+ touch t1
+ touch t3.dep
+ touch t3
```

.FOREGROUND Special Atom

.FOREGROUND - execute target action in foreground shell

Type

Dynamic Attribute

Description

This Special Atom causes the target action to be executed in the foreground shell. Normally, nmake computes future prerequisites while the action is being executed, but .FOREGROUND causes nmake to wait until the action completes before doing this computation.

Example

In the following example, nmake is run twice. The first time, the makefile does not contain the .FOREGROUND Special Atom; the second time it does. In both instances, nmake is invoked with the command-line option -j3, which allows nmake to execute three jobs at the same time. The first target, all, has four prerequisites. When nmake is invoked, these four prerequisites are made independently. In the first invocation of nmake, because the .FOREGROUND atom is not used, t2, t3, and t4 are made before t1 (because of the -j3 option). In the second invocation, because the .FOREGROUND atom is used, nmake waits for t1 to be made before making t2, t3, and t4.

Makefile contents (without the .FOREGROUND atom)

```
all : t1 t2 t3 t4
t1 :: a.c
t2 : b.c
        cp $(>) $(<)
t3 : c.c
        cp $(>) $(<)
t4 : d.c
        cp $(>) $(<)
```

Run nmake

```
$ nmake -j3
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -c a.c
+ cp b.c t2
+ cp c.c t3
+ cp d.c t4
+ cc -O -o t1 a.o
```

Makefile contents (with the .FOREGROUND atom)

```
all : t1 t2 t3 t4
t1 :: a.c .FOREGROUND
t2 : b.c
        cp $(>) $(<)
t3 : c.c
        cp $(>) $(<)
t4 : d.c
        cp $(>) $(<)
```

Run nmake

```
$ nmake -j3
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -c a.c
+ cc -O -o t1 a.o
+ cp b.c t2
+ cp c.c t3
+ cp d.c t4
```

.FUNCTIONAL Special Atom

.FUNCTIONAL - treat target atom as variable

Type

Dynamic Attribute

Description

A target with the .FUNCTIONAL attribute can be used as a variable. Each time the variable is to be expanded, the corresponding target is made before the variable value is determined. The target action may be a complex sequence of commands designed to compute and assign a value to the variable.

Example

In the following example, the value of $(var) is expanded. This expansion causes the function of var to be executed. The files a.c and b.c exist in the subdirectory src.

The single return statement used here could have been a complex series of commands.

Contents of Makefile

```
var : .MAKE .FUNCTIONAL .FORCE .REPEAT .SPECIAL            /* define the
                                                        function for var */
        return $(%:T=F)

tst :                                               /* define the target tst */
        : $(var a.c b.c)
```

Run nmake

```
$ nmake tst
```

Output

```
+ : src/a.c src/b.c
```

.GLOBALFILES Special Atom

.GLOBALFILES - atom whose prerequisites are a list of global makefiles

Type

Readonly List

Description

The list contains the names of all the global makefiles specified with the global option (-g flag) on the command-line.

Example

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

Contents of Makefile

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

Run nmake

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

Output

```
+ : glob1 glob2 glob3
```

.IGNORE Special Atom

.IGNORE - prevents atom from causing parent targets to be out-of-date

Type

Dynamic Attribute

Description

This Special Atom prevents a target from becoming out-of-date with any of its prerequisites that have the .IGNORE atom. This capability allows initialization sequences to be specified for individual atoms.

Examples

In the following example, the init target is made whenever the makefile is made. But because it has the .IGNORE atom, remaking the init target does not cause the prog target to be out-of-date.

Contents of Makefile

```
prog: init a.c b.c                                /* define the targets */
        $(CC) -o $(<) $(*:N=*.c)
init: .IGNORE
        : Start building prog
```

Run nmake

```
$ nmake
```

Output

```
+ : Start building prog
+ cc -o prog a.c b.c
a.c:
b.c:
```

Run nmake again

```
$ nmake -f ignore.mk
```

Output

```
+ : Start building prog
```

In the following example, changing the time-stamp a.c causes prog to be rebuilt.

Touch a.c

```
touch a.c
```

Run nmake again

```
$ nmake -f ignore.mk
```

Output

```
+ : Start building prog
+ cc -o prog a.c b.c
a.c:
b.c:
```

.IMMEDIATE (Dynamic Attribute) Special Atom

.IMMEDIATE - make atom immediately

Type

Dynamic Attribute

Description

An atom with this Special Atom is always made immediately.

Example

In the following example, target3 has been given the .IMMEDIATE Special Atom. Therefore, it is made first and is the only target listed as a result of the .DONE action (see the .DONE manual page). target3 is the only target made the second time nmake is run.

Contents of Makefile

```
PROGS = target1 target2 target3 target4             /* assign the list of
                                                    targets to PROGS */

all : $(PROGS)                         /* define the targets */

target1 :
        touch $(<)
target2 :
        touch $(<)
target3 : .IMMEDIATE
        touch $(<)
target4 :
        touch $(<)

.DONE :                           /* define the action for .DONE to list
                                  targets with the .IMMEDIATE attribute */
        : $(PROGS:A=.IMMEDIATE)
```

Run nmake

```
$ nmake
```

Output

```
+ touch target3
+ touch target1
+ touch target2
+ touch target4
+ : target3
```

Run nmake again

```
$ nmake
```

Output

```
+ touch target3
+ : target3
```

.IMMEDIATE (Immediate Rule) Special Atom

.IMMEDIATE - make prerequisites and actions

Type

Immediate Rule

Description

This Special Atom causes the prerequisites and actions to be made each time it is asserted. It is equivalent to the .MAKE Special Atom.

Example

.IMMEDIATE is useful for forcing a target to be triggered under user control. In this following example, the current directory is checked (by use of the B file component edit operator) during startup to verify that the file component's basename is the source directory. If the current directory does not have source as its basename file component, then the .init.error target is triggered. The .init.error target prints out an error message and exits.

Contents of Makefile

```
startup : .MAKE .VIRTUAL .FORCE
        if "$(PWD:B)" != "source"
                .IMMEDIATE : .init.error
        else
                error 0 Starting build in directory named source
        end

.init.error : .FORCE .ALWAYS .MAKE .VIRTUAL
        error 3 error: must be in the source directory
```

Run nmake

```
$ nmake
```

Output

If the current directory is named source:

```
Starting build in directory named source
```

If the current directory is not named source:

```
make: error: must be in the source directory
```

.IMPLICIT Special Atom

.IMPLICIT - apply implicit metarules to additional scenarios

Type

Dynamic Attribute

Description

This Special Atom causes the implicit metarules to be applied to targets that have prerequisites, but no explicit actions. If this Special Atom is not used, the implicit metarules are applied only to targets with no explicit actions and no prerequisites. This Special Atom turns off the .TERMINAL Special Atom at assertion time.

Example

The following example shows the actions taken on five different types of targets to show how the use of .IMPLICIT affects the use of metarules. The five types of targets shown are:

Type	Properties	Action
xsh	Has a prerequisite but no explicit actions, and no .IMPLICIT atom	Not made
ysh	Has no prerequisites and no explicit actions	Implicit use of metarules
zsh	Has a prerequisite, the .IMPLICIT atom, and no explicit actions	Implicit use of metarules
ash	Has a prerequisite, the .IMPLICIT atom and also has explicit actions	Explicit use of actions
bsh	Has a prerequisite and explicit actions	Explicit use of actions

Contents of Makefile

```
all : xsh ysh zsh ash bsh                     /* define targets */

xsh : xsh.sh

ysh :

zsh : zsh.sh .IMPLICIT

ash : ash.sh .IMPLICIT
        ln $(>) $(<)

bsh : bsh.sh
        ln $(>) $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ cp ysh.sh ysh
+ chmod u+w,+x ysh
+ cp zsh.sh zsh
+ chmod u+w,+x zsh
+ ln ash.sh ash
+ ln bsh.sh bsh
```

.INIT Special Atom

.INIT - atom made before user targets

Type

Sequence

Description

This Special Atom is made automatically after the .MAKEINIT Special Atom and before any target in a user makefile.

Example

The following example shows a .INIT action that is always made first.

Contents of Makefile

```
.INIT :
        : making $(<)

target :
        : making $(<)

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

Run nmake

```
$ nmake
```

Output

```
+ : making .INIT
+ : making target
+ : making .DONE
```

.INSERT Special Atom

.INSERT - insert assertion prerequisites at beginning of prerequisite list

Type

Assertion Attribute

Description

This Special Atom causes assertion prerequisites to be inserted at the beginning of the target prerequisite list, rather than at the end.

Examples

Separate examples are given to show two ways in which this atom can be used.

The first example shows how to cause a new member of a prerequisite list to be inserted before the existing members.

Define the .SOURCE.c rule

```
.SOURCE.c : DIR2 DIR3
.SOURCE.c : .INSERT DIR1
```

The rule is evaluated as . DIR1 DIR2 DIR3.

The current directory is implicit and is always searched first.

The second example shows how to force nmake to make a target in a local makefile (Makefile) before making a target in a global makefile (global.mk). Using .INSERT as a prerequisite to .INIT changes the .INIT : globtarg line in the global makefile to: .INIT : mytarg globtarg.

Edit global.mk

Define .INIT

```
.INIT : globtarg
```

Define the target

```
globtarg :
        : making globtarg
```

Edit Makefile

Define .INIT with the .INSERT atom

```
.INIT : mytarg .INSERT
```

Define the target

```
mytarg :
        : making mytarg
```

Run nmake

```
$ nmake -g global.mk
```

Output

```
+ : making mytarg
+ : making globtarg
```

.INSERT.*pattern* Special Atom

.INSERT.*pattern* - insert onto prerequisite list of .*pattern* atoms

Type

Pattern Association

Description

The prerequisites of .INSERT.*pattern* are inserted into the prerequisite list of each target atom that matches .*pattern* immediately before the target prerequisites are made. Note that .*pattern* must be %.ARCHIVE for .ARCHIVE targets and %.COMMAND for .COMMAND targets.

Example

The following example shows how to insert .ID.o into the prerequisite list of .COMMAND targets, which are a and c, so that id.o can be added to all the link steps of .COMMAND targets.

Contents of Makefile

```
.INSERT.%.COMMAND : .ID.o
.ID.o : .MAKE .VIRTUAL .REPEAT .FORCE .IGNORE
        $(<<) : id.o
:ALL:
a :: a.c
c :: c.c
```

Run nmake

```
$ nmake
```

Output

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

.INTERNAL Special Atom

.INTERNAL - for internal use only

Type

Readonly List

Description

This Special Atom is used internally and appears here for completeness.

.INTERRUPT Special Atom

.INTERRUPT - atom made if interrupt signal is caught

Type

Sequence

Description

This atom is made automatically when an interrupt signal is caught. It allows the user to perform special processing when an interrupt signal is caught.

If the interrupt occurs while .QUERY is active and set nointerrupt is executed by .INTERRUPT, nmake continues from the point where the interrupt occurred after .INTERRUPT is made. Otherwise, nmake exits with a nonzero status.

The commands in the action are always executed in the foreground. nmake exits after the commands have been executed.

The engine state may become corrupted by actions triggered while .INTERRUPT is active.

Example

In the following example, the interrupt key was pressed after target c was made. As a result, the action defined for the .INTERRUPT atom was performed and nmake stopped processing.

Contents of Makefile

```
.INTERRUPT :                    /* define the action for .INTERRUPT */
        : processing .INTERRUPT

targets = a b c d e f           /* assign target files to variable
                                targets */

all : $(targets)                /* define targets */

$(targets) :
        touch $(<)
        sleep 2
```

Run nmake

```
$ nmake all
```

Output

```
+ touch a
+ sleep 2
+ touch b
+ sleep 2
+ touch c
+ sleep 2
```

Press the Interrupt Key

```
+ : processing .INTERRUPT
```

.INTERRUPT<*signal*> Special Atom

.INTERRUPT<*signal*> - atom made if interrupt signal *signal* is caught

Type

Sequence

Description

This Special Atom is used to handle a specified signal. The signals SIGILL, SIGIOT, SIGEMT, SIGBUS, and SIGSEGV cannot be caught. <*signal*> is a symbolic signal name as specified in signal (5), without the SIG prefix.

Example

The following example handles both an alarm signal that is set for 10 seconds after nmake execution and any interrupt signals. The return value of the signal handler determines whether processing continues. A return value of -1 forces process termination.

```
.INTERRUPT.INT: .MAKE
        error caught interrupt
        return 0

.INTERRUPT.ALRM: .MAKE
        error caught alarm

.ALARM: 10

sleep:
        sleep 30
```

Repeating alarms can be set by reinitializing .ALARM within the signal handler. In this example, if the interrupt key is hit twice during execution, the output will be:

```
+ sleep 30
caught interrupt
caught interrupt
caught alarm
```

.JOINT Special Atom

.JOINT - specify that an action generates multiple files

Type

Dynamic Attribute

Description

This Special Atom is used to specify that an action generates multiple files. It causes all targets on the left-hand side to be jointly built with respect to the prerequisites on the right-hand side.

The :JOINT: assertion operator in the default base rules provides this functionality at a higher level and should be used in place of the .JOINT Special Atom.

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

Examples

In the following example, x.c and x.h are jointly built from xx because of the .JOINT Special Atom. The xx file must exist.

Contents of Makefile

```
x.c x.h : .JOINT xx
        cp $(*) $(<:N=*.c)
        cp $(*) $(<:N=*.h)
```

Run nmake

```
$ nmake
```

Output

```
+ cp xx x.c
+ cp xx x.h
```

In the following example, x.c and x.h are jointly built from xx because of the :JOINT: operator.

Contents of Makefile

```
x.c x.h :JOINT: xx
        cp $(*) $(<:N=*.c)
        cp $(*) $(<:N=*.h)
```

Run nmake

```
$ nmake
```

Output

```
+ cp xx x.c
+ cp xx x.h
```

.LOCAL Special Atom

.LOCAL - specify target action's interaction with coshell

Type

Dynamic Attribute

Description

When coshell is active it establishes shell coprocesses to machines in the LAN and send user jobs to these shell coprocesses. However, there are occasions when it is required to execute a process on the local machine. One such case is the need to ensure that coshell, available on the local machine, will be able to distribute the processing of the $(MAKE) command. So the assignment of .LOCAL as an attribute of the target whose action block defines the $(MAKE) assures that coshell will be able to distribute the subprocesses to the LAN.

Example

In the following example, .LOCAL is assigned as an attribute of target t.

Contents of Makefile

```
t : .LOCAL
        $(MAKE) $(-) $(=) -f mk1
```

Results

When the makefile mk1 is executed from target's action block, the process occurs on the local machine. Since coshell is available on the local machine, it can distribute the subprocesses to the LAN.

.MAIN Special Atom

.MAIN - atom whose prerequisites specify default targets when none are provided

Type

Dynamic List

Description

The prerequisites of .MAIN are used as the main targets if no targets are explicitly listed on the command-line. If no prerequisites are included in the input makefile, the first target that is not marked as .FUNCTIONAL, .OPERATOR, or .REPEAT, and is neither a state variable nor metarule is set as the first target.

Example

In the following example, target1 and target3 are prerequisites of .MAIN. When nmake is invoked without any specified targets, target1 and target3 are built. When a target is specified on the command-line, only that target is built, as in the second run of nmake in this example. If the .MAIN line in this makefile did not exist, the target target1 would be built.

Contents of Makefile

```
.MAIN: target1 target3

target1:
        : making $(<)
target2:
        : making $(<)
target3:
        : making $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ : making target1
+ : making target3
```

Run nmake again

```
$ nmake target2
```

Output

```
+ : making target2
```

.MAKE (Dynamic Attribute) Special Atom

.MAKE - treat target atom's action as makefile statements

Type

Dynamic Attribute

Description

This Special Atom causes the target action to be treated as a makefile and parsed, even if the -n command-line option is specified. This capability is used extensively for coding operators in the default base rules.

Example

In the following example, .MAKE allows the use of the nmake if construct to test the makefile variable MYINIT for a value and set the variable x accordingly. The actions of the setx target are read and executed as a makefile instance.

Contents of Makefile

```
t : setx                          /* define target */
        : $(x)

setx : .MAKE                      /* define .MAKE action for target t
                                  prerequisite setx */
        if ! "$(MYINIT)"
                x = 1
        else
                x = 2
        end
```

Run nmake

```
$ nmake
```

Output (x = 1 because MYINIT is null by default)

```
+ : 1
```

Run nmake with a value for MYINIT

```
$ nmake MYINIT=100
```

Output (x=2 because MYINIT is not null)

```
+ : 2
```

.MAKE (Immediate Rule) Special Atom

.MAKE - make prerequisites and actions

Type

Immediate Rule

Description

This Special Atom causes the prerequisites and actions to be made each time this rule is asserted. It is equivalent to the .IMMEDIATE immediate rule.

Example

In the following example, the .MAKE atom causes the prerequisite target1 to be made as part of the action for target.

Contents of Makefile

```
all : target

target : .MAKE
        .MAKE : target1

target1 :
        : making target1
```

Run nmake

```
$ nmake
```

Output

```
+ : making target1
```

.MAKEDONE Special Atom

.MAKEDONE - atom made after .DONE

Type

Sequence

Description

This Special Atom is made after .DONE and just before the statefile is updated. It is made unconditionally, regardless of the completion code of previous targets. The commands in the action block for this atom are always executed in the foreground shell.

Example

Assume that, during the build process, files with a suffix of .fld are created. They can be removed as the very last action the makefile takes, as follows:

```
.MAKEDONE :
        $(RM) $(RMFLAGS) *.fld
```

.MAKEFILES Special Atom

.MAKEFILES - atom whose prerequisites are a list of makefile names

Type

Readonly List

Description

The list specifies the makefile names read by nmake when the -f option is specified; otherwise, default makefiles are used.

Example

In the following example, one makefile, makefile1, is specified on the command-line, and the value of the list .MAKEFILES is set to that one name.

Contents of Makefile

```
tst :                          /* define the action of tst to print
                               the value of .MAKEFILES */
        : $(*.MAKEFILES)
```

Run nmake with makefile specified

```
$ nmake -f makefile1
```

Output

```
+ : makefile1
```

.MAKEINIT Special Atom

.MAKEINIT - atom made upon initialization

Type

Sequence

Description

A target that is a prerequisite of this Special Atom is made immediately after the statefile has been loaded and before the SOURCE targets are examined.

This Special Atom should not be redefined because the base rules use it to initialize nmake; however, it is acceptable to insert or append prerequisites to it. Variable assignments within .MAKEINIT commands override any command-line variable assignments.

Example

The following example shows how a prerequisite can be added to the .MAKEINIT atom in a global makefile (glob.mk) to append specified directories to .SOURCE.a after they have been initialized by .MAKEINIT. This approach is useful when a project's configuration management person does not have permission to regenerate the makerules.mo file.

Prepare the global makefile glob.mk

```
.MAKEINIT : .mysource            /* define the .Makeinit prerequisite */

.mysource : .MAKE .AFTER         /* specify the library directory to be
                                 appended after .MAKEINIT has been made */
        .SOURCE.a : ./libggg
```

Prepare the local makefile

```
a :: a.c -lggg            /* define the target */
```

Run nmake

```
$ nmake -g glob.mk
```

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 libggg/libggg.a
```

.MAKING Special Atom

.MAKING - marks target being made

Type

Readonly Attribute

Description

This Special Atom marks each atom whose action is executing. The attribute is assigned to the target from the time the shell action has been sent to the shell until the shell action returns to nmake.

Example

The following example shows how to control the number of targets with the .MAKING Special Atom using the -j option. The shell action echoes the current target and the targets that have the .MAKING Special Atom for each of three runs of nmake.

When the -j flag is set to 1 (-j1), nmake works on only one target at a time. nmake waits for the completion of one target before making the next. Because the echo of the current target is done while the shell action is in the shell, no target is given the .MAKING Special Atom.

When the -j flag is set to 2 (-j2), nmake determines the next target while the current target is being made (i.e., when nmake is expanding the shell action to make target y, target x has the .MAKING Special Atom).

When the -j flag is set to 3 (-j3), nmake behaves in the same manner as when it is set to 2. It is also free to determine that two more targets are to be made next and to send the shell script for making those targets to the shell (i.e., when nmake determines that target z is the next target, targets x and y are being made and have the .MAKING Special Atom).

Contents of Makefile

```
X = x y z                    /* assign the list of files to the X variable */

a : $(X)                     /* define the targets */

$(X) :
        : $(<) : $(X:A=.MAKING)
```

Run nmake

```
$ nmake -j1
```

Output

```
+ : x : x
+ : y : y
+ : z : z
```

Run nmake

```
$ nmake -j2
```

Output

```
+ : x : x
+ : y : x y
+ : z : y z
```

Run nmake

```
$ nmake -j3
```

Output

```
+ : x : x
+ : y : x y
+ : z : x y z
```

.MEMBER Special Atom

.MEMBER - marks members of bound archive atoms

Type

Readonly Attribute

Description

This atom marks atoms that are members of a bound archive atom.

Example

The following example shows how to list those atoms that are members of a bound archive. It is assumed that x.a exists and all prerequisites are up-to-date when nmake is run.

Contents of Makefile

```
SRC = aa.c bb.c          /* aa.c and bb.c to the SRC variable */

:ALL:                    /* all :: targets are made if none are specified
                         on the command-line */

x.a :: aa.c              /* define the targets */
x :: bb.c

.DONE   :                /* define the action for .DONE to list the
                         members of bound archives, i.e., x.a */
        : $(SRC:B:S=.o:A=.MEMBER)
```

Run nmake

```
$ nmake
```

Output

```
+ : aa.o
```

.METARULE Special Atom

.METARULE - atom whose prerequisites are a list of metarules

Type

Readonly List

Description

The list contains the left-hand side default base rule patterns for all asserted default base rules except the % match-all default base rules; it determines the order in which default base rules are applied.

Example

The following example shows how to list the default base rules that are asserted when the makefile is executed. The %.o and %.mo rules are listed in the output because they are part of the default base rule assertions.

Contents of Makefile

```
.INIT :                         /* define the action for .INIT to
                                list the asserted default base rules */
        : $(*.METARULE)

a.x : a.y .IMPLICIT             /* define the targets */

%.x : %.y
        cp $(>) $(<)
```

Run nmake

```
$ nmake a.x
```

Output

```
+ : %.o %.c %.h %.mo cc-% FEATURE/% %.A %.x
+ cp a.y a.x
```

.MULTIPLE Special Atom

.MULTIPLE - allows duplicate prerequisites

Type

Dynamic Attribute

Description

This Special Atom allows duplicate prerequisites to remain in a prerequisite list. This capability can be used for library prerequisites that may require multiple scans.

Examples

In the following example -lm is given the .MULTIPLE Special Atom to allow it to appear more than once in the prerequisite list. Note that -lcurses is also listed as a prerequisite more than once but only appears once in the output list.

Contents of Makefile

```
-lm : .MULTIPLE
LIST =  -lm -lm -lcurses -lcurses
target: $(LIST)
        : $(*)
```

Run nmake

```
$ nmake target
```

Output

```
+: /lib/libm.a /lib/libm.a /lib/libcurses.a
```

In the following example the libm.a library is to be be listed twice at the link step. It works correctly because the source dependency operator generates the link rule with a .MULTIPLE Special Atom.

Contents of Makefile

```
a :: program.c -lm -lm
```

Run nmake

```
$ nmake
```

Output

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

.NOCROSSPRODUCT.*pattern* Special Atom

.NOCROSSPRODUCT.*pattern* - suppress cross product expansion

Type

Pattern Association

Description

When this Special Atom is applied as a prerequisite to .SOURCE.*pattern2*, the items matching *pattern* in the .SOURCE.*pattern2* list are not expanded for each node of the viewpath when SOURCE.*pattern2* is expanded. Only the first occurrence of the item in the viewpath is expanded. This is useful for some types of files that can be specified in search paths.

Examples

The following is included in the default Scanrules.mk. For every .jar and .zip file listed in .SOURCE.class, only the first of each file in the viewpath is returned when .SOURCE.class is expanded. Other items, such as directories, are expanded for each node of the viewpath as usual.

.SOURCE.class : .NOCROSSPRODUCT.%.jar .NOCROSSPRODUCT.%.zip

.NOTYET Special Atom

.NOTYET - atom that has not yet been made

Type

Readonly Attribute

Description

This Special Atom marks atoms that have not yet been made.

Example

The following example shows that the .NOTYET Special Atom is removed from a target when nmake starts making that target (i.e., while a target is being made, it no longer has the .NOTYET attribute).

As each target is made, its attribute list is updated. When target c is made, none of the targets has the .NOTYET attribute.

Contents of Makefile

```
ALLTARGS = a b c

all: $(ALLTARGS)

.PRINT.NOTYET : .USE
            : $(<) "->         "\$(ALLTARGS:A=.NOTYET)

$(ALLTARGS) : .PRINT.NOTYET
```

Run nmake

```
$ nmake
```

Output

```
+ : a ->   b c
+ : b ->   c
+ : c ->
```

.NULL Special Atom

.NULL - no action

Type

Assertion Attribute

Description

This Special Atom defines a null (empty) action for the targets in an assertion.

Example

The .ARCLEAN rule is defined in the default base rules. Its function is to remove intermediate object files after creating an archived library.

In the following example, nmake is run twice. The first time, the makefile does not contain the .NULL Special Atom, so the default .ARCLEAN action occurs and the intermediate files are removed. The second time nmake is run the makefile contains the .NULL Special Atom, which prevents the default .ARCLEAN action from occurring. As a result, the intermediate files are not removed.

Contents of Makefile

```
libxy.a :: x.c y.c
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c x.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c y.c
+ ar r libxy.a x.o y.o
ar: creating libxy.a
+ ignore ranlib libxy.a
+ rm -f x.o y.o
```

Contents of Makefile

```
.ARCLEAN : .NULL
libxy.a :: x.c y.c
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c x.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c y.c
+ ar r libxy.a x.o y.o
ar: creating libxy.a
+ ignore ranlib libxy.a
```

.OPERATOR Special Atom

.OPERATOR - indicates that target atom is an assertion operator

Type

Dynamic Attribute

Description

This Special Atom marks the target as an assertion operator to be applied when reading makefiles. Assertion operators provide fine control over makefile assertions.

Each operator is an atom whose action is executed whenever it appears in an assertion. Assertion operator names must match either :: or :*identifier*:.

Assertion operators are defined by assertions. The operator name must be enclosed in double quotes in the defining assertion.

":*operator***:" : .OPERATOR**

An assertion operator is activated by an assertion of the form:

target **:***operator***:** *prerequisites*

When an assertion of this type is read, only the variables used in the action definition of the operator get expanded.

Example

The following example defines :display: as an assertion operator. The operator echoes the left-hand side and right-hand side atoms of this operator. In the example, when :display: is read, $(<) expands to target a, and $(>) expands to $(LIST).

Contents of Makefile

```
LIST = b c d
":display:" : .OPERATOR
        : the left-hand side is $(<)
        : the right-hand side is $(>)

a :display: $(LIST)
```

Run nmake

```
$ nmake
```

Output

```
+ : the left-hand side is a
+ : the right-hand side is b c d
```

.OPTIONS Special Atom

.OPTIONS - atom whose prerequisites are a list of options declared by the option option

Type

Readonly List

Description

The list contains the options declared by the option option.

Example

The following example shows how new options may be declared and displayed using the .OPTIONS atom. Declared options can be used in a makefile or on the command-line in the same manner as the standard options.

Contents of Makefile

```
set option=Z,zopt,b             /* declare boolean option Z */

set zopt                        /* initialize Z by option name */

.INIT :                         /* define the action for .INIT to list the*/
        : 1 $(*.OPTIONS)        /* members of the .OPTIONS list and to */
        : 2 $(-)                /* include only the non-default values */

all : .MAKE                     /* define the targets. the if construct */
        if "$(-Y)"              /* describes the options X and z */
                error 0 Y is set to $(-Y)
        else
                error 0 Y is not set
        end
        if "$(-Z)"
                error 0 Z is set to $(-Z)
        else
                error 0 Z is not set
        end
```

Run nmake

```
$ nmake
```

Output

```
+ : 1 Z,zopt,b
+ : 2 -o option=Z,zopt,b, zopt
```

```
Y is not set
Z is set to 1
```

Run nmake again

```
$ nmake +Z
```

Output

```
+ : 1 Z,zopt,b,
+ : 2 -o option=Z,zopt,b nozopt
Y is not set
Z not set
```

Run nmake again

```
$ nmake option=Y,yopt,n
```

Output

```
+ : 1 Y,yopt,n Z,zopt,b,
+ : 2 -o option=Z,zopt,b zopt
Y is not set
Z is set to 1
```

Run nmake again

```
$ nmake -o option=Y,yopt,n -o yopt=10
```

Output

```
+ : 1 Y,yopt,n Z,zopt,b,
+ : 2 -o option=Y,yopt,n, yopt=10 option=Z,zopt,b zopt
Y is set 10
Z is set to 1
```

.PARAMETER Special Atom

.PARAMETER - prevent expansion of state variables in certain cases

Type

Dynamic Attribute

Description

This Special Atom prevents the expansion of state variables by the T=D and T=E edit operators.

Example

In the following example, the .PARAMETER Special Atom is used to prevent the variable VAR1 from being expanded.

Changing the value of VAR1 can still force a rebuild.

Contents of Makefile

```
VAR1 = 100                      /* assign values to variables */
VAR2 = 200

(VAR1) : .PARAMETER             /* assign the .PARAMETER atom to VAR1 */

target : (VAR1) (VAR2)          /* define the target */
        : "$(&)"
        : "$(&:T=D)"
        : "$(&:T=E)"
        touch $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ : (VAR1) (VAR2)
+ : -DVAR2=200
+ : VAR2=200
+ touch target
```

Run nmake

```
$ nmake VAR1=200
```

Output

```
+ : (VAR1) (VAR2)
+ : -DVAR2=200
+ : VAR2=200
+ touch target
```

.QUERY (Action Rule) Special Atom

.QUERY - enter interactive debug session

Type

Action Rule

Description

This Special Atom places nmake in an interactive debugging loop. Input entered at the make> prompt can be any valid makefile input. However, a blank line must follow an interactive assertion (i.e., one entered at the keyboard) before it takes effect. If further input is required to complete a construct, the >>>>> prompt is displayed.

If a list of atoms is entered without an assertion or assignment operator, the atoms are listed as if they were prerequisites of .QUERY. To exit the debugging loop, enter Control-D from the keyboard.

The assertion:

```
.INTERRUPT : .QUERY
```

causes the debugging loop to start when nmake is interrupted.

Example

The following example demonstrates the use of .QUERY to examine the state of the object, qry. Text following the make> prompt is entered from the keyboard during nmake execution.

Contents of Makefile

```
qry :: qry.c            /* define the target qry */

.DONE : .QUERY          /* define the target .DONE; this begins an
                        interactive debugging loop after all targets
                        have been made */
```

Run nmake

```
$ nmake -j0
```

The –j flag ensures that nmake finishes the work for qry before going to .DONE.

Output

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

Debug prompt

```
make>
```

Debug prompt and user response (to end interactive session)

```
make> CTRL-d
```

.QUERY (Immediate Rule) Special Atom

.QUERY - send information to stderr

Type

Immediate Rule

Description

This Special Atom causes full atom and state information to be listed to stderr (standard error) for each prerequisite.

Example

In the following example, information is sent to stderr for the prerequisites .SOURCE.h and qry because of the .QUERY atom.

Contents of Makefile

```
.SOURCE.h : incl1                /* list the prerequisites for which you
                                 want to see atom and state information */

qry :: qry.c

.DONE : getinfo           /* request that the information be gathered
                          after all targets have been made */

getinfo : .MAKE                  /* define the target getinfo */
        .QUERY : .SOURCE.h qry
```

Run nmake

```
$ nmake all
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/\
pp/B49DF4E0.bincc -I- -c qry.c
+ cc -O -o qry qry.o
make: warning: getinfo has been replaced by an older version

.SOURCE.h : [not found] target cached unbound
prerequisites: . incl1 /usr/include

qry : [Dec 01 15:48:03 1997] must=6 command target \
compiled regular triggered EXISTS
prerequisites: qry.o .COMMAND.o (CCLD) (CCFLAGS) (F77FLAGS) \
(LDFLAGS)
```

```
()qry : [Dec 01 15:48:06 1997] event=[Dec 01 15:48:05 1997] \
force built compiled
 prerequisites: qry.o .COMMAND.o (CCLD) (CCFLAGS) (F77FLAGS) \
(LDFLAGS)
```

.READ Special Atom

.READ - read the standard output of target action’s action block as nmake statements

Type

Dynamic Attribute

Description

This Special Atom causes the standard output of an action to be read and parsed as nmake statements. This atom allows variable definitions and makefiles to be modified dynamically.

The .READ Special Atom is intended as a replacement for the _make_read shell command.

Example

The following example sets the variables logname and date to the current user's login name and the current time, respectively. In this example, the output of the UNIX system commands specified in the backquotes is read into the variable on the left-hand side of the assignment operator.

Contents of Makefile

```
defvars : .READ
        echo logname = `logname`
        echo date = `date '+%m/%d/%y'`

.DONE:
        : $(logname)
        : $(date)
```

Run nmake

```
$ nmake
```

Output

```
+ logname
+ echo logname = nmake
+ date +%m/%d/%y
+ echo date = 06/29/98
+ : nmake
+ : 06/29/98
```

.REBIND Special Atom

.REBIND - refresh time-stamp on prerequisites

Type

Immediate Rule

Description

When .REBIND is read, each prerequisite is time-stamped immediately, as though it had been made.

Example

The following example shows a line in a makefile that causes the time-stamp to be changed on files a.c and b.c.

Change the time-stamp

```
.REBIND : a.c b.c
```

.REGULAR Special Atom

.REGULAR - marks atom bound to file

Type

Readonly Attribute

Description

This Special Atom marks atoms that are bound to regular files (not directories). An atom is bound to a regular file if the name of the atom can be found as a regular file (not a directory or special file).

Once bound, the atom is given the .REGULAR attribute to describe its object type, and the modification time of the file becomes the atom's time-stamp value. The expanded value of the atom is the path to the regular file.

In the process of binding the atom, nmake searches the current directory by default. The .SOURCE Special Atom can also be specified in the makefile to designate one or more other directories to search.

Example

In the output, the first line in the output indicates that file1 was created, the second that the directory dir1 was created. The third line is the list of all the explicit file prerequisites, and the fourth line lists only those items given the .REGULAR attribute.

Contents of Makefile

```
VAR = file1 dir1
target : $(VAR)
        : $(*)
        : $(*:A=.REGULAR)
file1 :
        >$(<)
dir1 :
        mkdir $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ 1> file1
+ mkdir dir1
+ : file1 dir1
+ : file1
```

.REPEAT Special Atom

.REPEAT - makes target atom repeatedly

Type

Dynamic Attribute

Description

This Special Atom marks targets that are to be made repeatedly. To trigger an action each time the target is made, the .FORCE Special Atom must also be specified. The .REPEAT Special Atom is useful when an nmake variable must be re-evaluated or when a target must be made repeatedly.

Examples

In the following example, the .REPEAT Special Atom is not used. Compare the output to that of the second example, which uses the .REPEAT Special Atom.

Contents of Makefile

```
x : .READ .FORCE .FUNCTIONAL .SPECIAL
        echo $(<) = ‘date; sleep 2‘

b :
        : $(x)
        : $(x)
```

Run nmake

```
$ nmake
```

Output

```
+ date
+ echo x = Thu Jan 13 08:57:27 EST 1998
+ : Thu Jan 13 08:57:27 EST 1998
+ sleep 2
+ : Thu Jan 13 08:57:27 EST 1998
```

In the following example, the .REPEAT Special Atom is used. Note that x contains a different value the second time because x is marked to be made repeatedly. When $(x) is encountered in the action of b, the value of x is regenerated.

Contents of Makefile

```
x : .READ .FORCE .FUNCTIONAL .SPECIAL .REPEAT
        echo $(<) = ‘date‘

b :
        : $(x)
        sleep 2
        : $(x)
```

Run nmake

```
$ nmake
```

Output

```
+ date
+ echo x = Thu Jan 13 09:00:24 EST 1998
+ date
+ echo x = Thu Jan 13 09:00:25 EST 1998
+ : Thu Jan 13 09:00:24 EST 1998
+ sleep 2
+ : Thu Jan 13 09:00:25 EST 1998
```

.REQUIRE.*pattern* Special Atom

.REQUIRE.*pattern* - associate bound atom with list of bound atoms

Type

Pattern Association

Description

This Special Atom modifies the binding algorithm to allow a bound atom to map to a list of bound atoms.

Example

Contents of Makefile

```
.REQUIRE.file1% : .FUNCTION
        return file1 file2
target : file1
        : $(*)
```

Run nmake

```
$ nmake
```

Output

```
+ : file1 file2
```

The following example shows how a specific atom can be replaced by a list of atoms using .REQUIRE.*pattern*. In this case, the prerequisite requiretest is replaced by all the files in the current directory matching test*.c, assuming that files test1.c, test2.c, and test3.c exist in the current directory.

Contents of Makefile

```
requiretest : .VIRTUAL .REPEAT
.REQUIRE.require% : .FUNCTION
        return $(".":L=$(%:/^require//)*.c)
target : requiretest
        : $(*)
```

Run nmake

```
$ nmake
```

Output

```
+ : test1.c test2.c test3.c
```

.RETAIN Special Atom

.RETAIN - retain prerequisite variables in statefile

Type

Immediate Rule

Description

This Special Atom causes the prerequisites to be interpreted as variable names whose values are to be retained in the statefile.

Example

In the following example, the .RETAIN Special Atom forces COUNT to be saved in the statefile; therefore, each time the 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

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

Run nmake

```
$ nmake
```

Output

```
1
```

Run nmake

```
$ nmake
```

Output

```
2
```

.SCAN Special Atom

.SCAN - define scan rules

Type

Dynamic Attribute

Description

This Special Atom is used to define scanning rules for a product-specific language, if rules for that language do not appear in the default base rules. Refer to Chapter 8, *Defining Custom Scan Rules*, in the *Alcatel-Lucent nmake Product Builder User's Guide* for further information on the dependency scanner.

.SCAN.*x* Special Atom

.SCAN.*x* - scan target using *x* scan strategy

Type

Dynamic Attribute

Description

This Special Atom marks a bound target to be scanned for implicit prerequisites using one of the *x* strategies detailed in the following manual pages. This convention is followed in the default scan rules, which contain assertions of the form:

.ATTRIBUTE.*pattern* **: .SCAN.***x*

.SCAN.c Special Atom

.SCAN.c - scan target using C language scan strategies

Type

Dynamic Attribute

Description

This Special Atom scans for C #include file prerequisites and candidate state variable references. Include statements requiring macro substitutions are not recognized. " " include files are bound using the directories of .SOURCE.c first, and then the directories of .SOURCE and .SOURCE.h. This order is the opposite of the default directory binding order. < > include files are bound using only the directories of .SOURCE.h.

" " include files are assigned the .LCL.INCLUDE attribute; < > include files are assigned the .STD.INCLUDE attribute. .LCL.INCLUDE takes precedence if a file appears as both " " and < >.

To support include file packages, if an including file is included as *prefix/*includer, a file includee included by *prefix*/includer is first bound using *prefix*/includee and, if that fails, it is bound using .SOURCE.h, .SOURCE.c, and .SOURCE, respectively.

Example

See the example for .SCAN.m4.

.SCAN.f Special Atom

.SCAN.f - scan target using FORTRAN scan strategies

Type

Dynamic Attribute

Description

This Special Atom scans for FORTRAN include and INCLUDE file prerequisites on generic SQL include file prerequisites.

Example

See the example for .SCAN.m4.

.SCAN.F Special Atom

.SCAN.F - scan target using generic SQL scan strategies

Type

Dynamic Attribute

Description

This Special Atom scans for generic SQL include file prerequisites of the form $include *file* and for everything included in the .SCAN.sql, .SCAN.f, and elements of the .SCAN.c strategies.

Example

See the example for .SCAN.m4.

.SCAN.idl Special Atom

.SCAN.idl - scan target using IDL language scan strategies

Type

Dynamic Attribute

Description

Scan for IDL #include file prerequisites and candidate state variable references. Include statements requiring macro substitutions are not recognized. " " include files are bound using the directories of .SOURCE.idl first and then .SOURCE. < > include files are bound using only the directories of .SOURCE.idl.

" " include files are assigned the .LCL.INCLUDE.idl attribute; < > include files are assigned the .STD.INCLUDE.idl attribute.

To support popular coding style, if a file is included as #include "*prefix*/file1" (where *prefix* represents at least one level of a directory structure), and if file1 includes another file as #include "file2", nmake first looks in *prefix* for file2. If it is not found there, nmake uses .SOURCE.idl and .SOURCE respectively to locate file2.

Example

See the example for .SCAN.m4.

.SCAN.m4 Special Atom

.SCAN.m4 - scan target using m4 scan strategies

Type

Dynamic Attribute

Description

This Special Atom scans for m4(1) include( ), sinclude( ), and INCLUDE( ) file prerequisites. The INCLUDE( ) form also handles S (statistical analysis package) source file prerequisites. Include statements requiring macro substitutions are not recognized.

Example

The following example applies the m4 scanning strategy to .mc files. It assumes the existence of an m4 command that accepts -I flags, accepts .mc files as input, and generates .c files as output. Similar methods can be used for other scanning strategies.

The DASHI rule defined in the example generates the -I flag..

The processing that creates the rule DASHI is as follows:

.CLEAR	It is good practice to clear a rule in case that same atom name has been used somewhere else and has been initialized.
$(!$(>>)	nmake accesses the value of $(!) for the parent assertion (e.g., the %.c : %.mc rule in the example).
A=.SCAN.m4	nmake uses only those atoms with the .SCAN.m4 attribute.
D	nmake extracts only the directory portion of the following atoms.
:C/^/-I/)	nmake edits each string and precedes it with -I
.U	.U contains the unique list of -I flags of searching directories.
DASHI := $(*.U)	nmake accesses the prerequisite list for the atom .U and returns that as the function value.

In the following example, it is assumed that file a.gc is in the current directory and file MultiRole.gc in the incl1 directory.

Contents of Makefile

```
M4FGS =                              /* include the M4FLAGS state variable */

.ATTRIBUTE.%.mc : .SCAN.m4           /* assign the m4 scan strategy to all
                                     .mc files */

.SOURCE : INCL1                      /* define the directories to be
                                     included in the process */

%.c : %.mc (M4) (M4FGS)              /* define a rule to generate .c files
                                     from .mc files */
        $(M4) $(M4FGS) $(DASHI) $(>) $(<)

program :: main.c file1.mc                   /* define the target */

DASHI : .MAKE .FUNCTIONAL .FORCE .REPEAT .VIRTUAL/* define the rule
                to generate -I flags from the list generated in $(!)*/
        .U : .CLEAR $(!$(>>):A=.SCAN.m4 :D:C/^/-I/)
        DASHI := $(*.U)
```

Contents of file1.mc

```
include (MultiRole.gc)
include (a.gc)
```

Run nmake

```
$ nmake
```

Output

```
+ m4 -I. file1.mc
+ 1> file1.c
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c file1.c
+ cc -O -o program main.o file1.o
```

.SCAN.nroff Special Atom

.SCAN.nroff - scan target using nroff/troff scan strategies

Type

Dynamic Attribute

Description

This Special Atom scans for nroff(1) and troff(1) .so file prerequisites. Only .so *file* starting at the beginning of a line is recognized. .so statements requiring macro substitutions are not recognized. (See the *UNIX System V User 's Reference Manual* for information about nroff and troff.)

Example

The following example shows how .SCAN.nroff works with an nroff file containing a .so directive. The file docinc.nroff contains the line
.so inc.nroff to include the file inc.nroff when processing takes place.

The first time nmake is run, both prerequisite files are listed. When inc.nroff is touched, docinc.nroff alone becomes out-of-date and it alone is listed.

The .VIRTUAL atom maintains the time-stamp for the target nroff in the statefile.

The line assigns the .SCAN.nroff attribute to files with the .nroff suffix.

Contents of Makefile

```
ROFFCMD = ls                          /* define the state variables */
ROFFFLAGS = -l

.ATTRIBUTE.%.nroff : .SCAN.nroff               /* define scan strategy */

nroff : docinc.nroff docin2.nroff .VIRTUAL/* define the target */
        $(ROFFCMD) $(ROFFFLAGS) $(>)
```

Run nmake

```
$ nmake
```

Output

```
+ ls -l docinc.nroff docin2.nroff
-rw-r--r--  1 nmake nmake  0 Jun 16 22:21 docin2.nroff
-rw-r--r--  1 nmake nmake  14 Jun 16 22:27 docinc.nroff
```

Touch inc.nroff

```
touch inc.nroff
```

Run nmake again

```
$ nmake
```

Output

```
+ ls -l docinc.nroff
-rw-r--r--   1 nmake nmake   14 Jun 16 22:27 docinc.nroff
```

.SCAN.r Special Atom

.SCAN.r - scan target using RATFOR scan strategies

Type

Dynamic Attribute

Description

This Special Atom scans for RATFOR include and INCLUDE file prerequisites. Include statements requiring macro substitutions are not recognized.

Example

See the example for .SCAN.m4

.SCAN.sh Special Atom

.SCAN.sh - scan for state variable references

Type

Dynamic Attribute

Description

This Special Atom scans for candidate state variable references (i.e., those state variables initialized with the == operator).

Example

In the default base rules, files with the .sh suffix are assigned the .SCAN.sh attribute. The following example shows how a shell script can be changed by using state variables and the SCAN.sh strategy.

Contents of Makefile

```
X == 1                                /* initialize state variables */
Y == 1

script :: script.sh                   /* define the target */
```

Contents of script.sh

```
if [ $Y = $X ]
then
        echo EQUAL
else
        echo NOT EQUAL
fi
```

Run nmake

```
$ nmake
```

Output

```
+ 1> script
+ + 0< script.sh
+ read x
+ echo if [ $Y = $X ]
i=if [ $Y = $X ]
+ echo X=1 Y=1
+ cat script.sh
+ chmod u+w,+x script

$ script
EQUAL
```

Contents of script

```
X=1 Y=1
if [ $Y = $X ]
then
        echo EQUAL
else
        echo NOT EQUAL
fi
```

.SCAN.sql Special Atom

.SCAN.sql - scan target using SQL scan strategies

Type

Dynamic Attribute

Description

This Special Atom scans for SQL include file prerequisites of the form EXEC SQL include *filename* or #include *filename* where *filename* is "*file*", *file*;, or *file*.

Example

In the following example, suppose that a header file called sqlhdr.h exists in the current directory.

Contents of Makefile

```
.ATTRIBUTE.%.sql : .SCAN.sql

t : t.sql
        : $(!)
```

Contents of t.sql

```
EXEC SQL include "sqlhdr"
```

Run nmake

```
$ nmake
```

Output

```
+ : t.sql sqlhdr.h
```

.SCAN.IGNORE Special Atom

.SCAN.IGNORE - inhibit scanning of target atom

Type

Dynamic Attribute

Description

This Special Atom inhibits scanning. It is used to override .ATTRIBUTE.*pattern* scan strategies.

Example

The following example demonstrates the use of .SCAN.IGNORE to inhibit scanning for $(FILES). The purpose of the makefile is to move $(FILES) to the directory $(DIR). The file space.c includes a header file (sys/driver.h) that does not exist. To inhibit scanning and prevent an error, .SCAN.IGNORE is assigned as a prerequisite to $(FILES). Note that when nmake is run a third time, after the .SCAN.IGNORE Special Atom has been removed, it generates an error message because of the missing header file.

Contents of Makefile

```
FILES = space.c
DIR = ../src
$(FILES) : .SCAN.IGNORE
$(DIR) :INSTALLDIR: $(FILES)
```

Run nmake

```
$ nmake install
```

Output

```
+ ignore cp space.c ../src/space.c
```

Remove the line $(FILES) : .SCAN.IGNORE from the Makefile

Contents of Makefile

```
FILES = space.c
DIR = ../src
$(DIR) :INSTALLDIR: $(FILES)
```

Run nmake with clobber common action

```
$ nmake clobber
```

Output

```
+ rm -rf Makefile.mo Makefile.ms ../src/space.c
```

Run nmake again

```
$ nmake install
```

Output

```
make: don't know how to make .INSTALL : ../src/space.c : \
space.c : sys/driver.h
```

SCAN.NULL Special Atom

.SCAN.NULL - inhibit scanning of target atom

Type

Dynamic Attribute

Description

This Special Atom inhibits scanning. It is used to override .ATTRIBUTE.*pattern* scan strategies.

.SCAN.STATE Special Atom

.SCAN.STATE - mark state variable for scanning

Type

Dynamic Attribute

Description

This Special Atom marks candidate state variables for scanning.

.SCANNED Special Atom

.SCANNED - marks archive or directory atoms that have been scanned

Type

Readonly Attribute

Description

This Special Atom marks archive and directory atoms that have been scanned for members. It can also be removed from files being scanned to cause rescanning at a later time; the syntax for removing the .SCANNED attribute is -SCANNED, e.g., a.c : -SCANNED. Applying -SCANNED to a directory will remove the directory and its contents from the internal directory cache.

Example

In the following example, a target called tst is created from the prerequisite, main.c (which includes a header file called main.h). main.h is located in the ./include directory. After tst has been created, the main.c and include/main.h atoms have the .SCANNED attribute.

Contents of main.c

```
#include "main.h"
main(){
}
```

Contents of Makefile

```
.SOURCE.h : include
tst :: main.c
.DONE :
        : $(!tst:A=.SCANNED)
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -Iinclude -I- -c main.c
+ cc -O -o tst main.o
+ : main.c include/main.h
```

.SEMAPHORE Special Atom

.SEMAPHORE - limit number of concurrent shell executions of action block

Type

Dynamic Attribute

Description

The .SEMAPHORE Special Atom can appear more than once in an assertion. The number of .SEMAPHORE attributes in an assertion is called the *semaphore count*. The maximum semaphore count is seven. This attribute limits the number of concurrent shell executions to the semaphore count. The limitation is applied only to assertions having this attribute. For example,

```
semx : .SEMAPHORE .SEMAPHORE
```

allows up to two actions that depend on semx to run concurrently.

Example

In the following example, the set jobs = 10 line allows nmake to perform up to 10 jobs concurrently. The .SEMAPHORE attribute is used to separate those jobs that must be done serially.

In the first version of the makefile (without the .SEMAPHORE attribute), nmake can make the targets in any order, so if nmake were run again, the output might differ from that displayed here. Because of this, targets are not always generated from the correct prerequisites. For example, a has the contents of bb, not aa.

The second version of the makefile contains the .SEMAPHORE Special Atom to regulate concurrent actions and produce the correct results.

Contents of Makefile

```
set jobs=10
TARGETS = a b c d e f g

all : $(TARGETS)

a : aa
        sleep 10
        cp $(*) file1
        cp file1 $(<)
```

```
b : bb
        sleep 10
        cp $(*) file1
        cp file1 $(<)

c : cc
        sleep 10
        cp $(*) file1
        cp file1 $(<)

d : dd
        cp $(*) file1
        cp file1 $(<)

e : ee
        cp $(*) $(<)

f : ff
        cp $(*) $(<)

g : gg
        cp $(*) $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ sleep 10
+ sleep 10
+ sleep 10
+ cp ee e
+ cp dd file1
+ cp gg g
+ cp file1 d
+ cp ff f
+ cp bb file1
+ cp aa file1
+ cp cc file1
+ cp file1 b
+ cp file1 c
+ cp file1 a
```

Remove old output

```
$ nmake clobber
+ ignore rm -f Makefile.mo Makefile.ms a b c d e f g
```

Contents of Makefile

```
set jobs=10
TARGETS = a b c d e f g

all : $(TARGETS)
```

```
.sema1 : .SEMAPHORE .SEMAPHORE
.sema : .SEMAPHORE

a : aa .sema
        sleep 10
        cp $(*) file1
        cp file1 $(<)

b : bb .sema
        sleep 10
        cp $(*) file1
        cp file1 $(<)
c : cc .sema1
        sleep 10
        cp $(*) file1
        cp file1 $(<)
d : dd .sema
        cp $(*) file1
        cp file1 $(<)
e : ee
        cp $(*) $(<)
f : ff
        cp $(*) $(<)
g : gg
        cp $(*) $(<)
```

Run nmake

```
$ nmake
```

Output

```
+ sleep 10
+ sleep 10
+ cp ee e
+ cp ff f
+ cp gg g
+ cp aa file1
+ cp cc file1
+ cp file1 a
+ cp file1 c
+ sleep 10
+ cp bb file1
+ cp file1 b
+ cp dd file1
+ cp file1 d
```

.SOURCE/.SOURCE.*pattern* Special Atom

.SOURCE/.SOURCE.*pattern* - atoms whose prerequisites name directories used in searching for files

Type

Dynamic List

Description

The prerequisites of the .SOURCE Special Atom name directories to be scanned when searching for files. The makefile and prerequisite files can be located in any directory. The first matching file found in a listed directory in left-to-right order is used. The current directory (.) is always searched first.

The .SOURCE.*pattern* Special Atoms are used to specify the directories to be scanned for file names that match the specified *pattern*. A percent sign (%) is used as the wildcard character.

The .SOURCE Special Atom is used to specify the directories to be scanned for files that were not found in the current directory or .SOURCE.*pattern*.

When searching for files that are explicit prerequisites (i.e, appearing on the right-hand side of an operator), nmake first looks in the current directory (.), then in the .SOURCE.*pattern* directories, and finally in the .SOURCE directories.

When searching for files that are local include files (i.e., #include "file.h"), nmake first looks in the current directory, then in the .SOURCE.c directories, then in the .SOURCE directories, and finally in the .SOURCE.h directories. This order follows the standard philosophy that local include files are closely tied to the including source files.

When searching for files that are standard include files (i.e., #include <stdio.h>), nmake searches the current directory and the .SOURCE.h directories.

The default value of a .SOURCE/.SOURCE.*pattern* atom depends on the value of the CC state variable. This value is initialized from the contents of the probe information file corresponding to that language processor. (The .SOURCE.h and .SOURCE.a atoms are set to the standard directories for the compiler that is specified in CC.)

The .FORCE special atom can be used with .SOURCE.*pattern* to inhibit the use of .SOURCE for file searches.

Example

The following assertions name the directories to be scanned for files that have a .h or .a suffix.

```
.SOURCE.h : include
.SOURCE.a : lib
```

The following assertion names the directory to be scanned for all files that have the user-defined .4GL.INCLUDE attribute.

```
.4GL.INCLUDE : .ATTRIBUTE
.SOURCE.%.4GL.INCLUDE : ./4gl
```

The following assertion names the directory to be scanned for all files which begin with x.

```
.SOURCE.x% : ./x
```

The following assertion tells nmake to search .SOURCE.h for standard include files. With .FORCE specified nmake will not search the .SOURCE directories.

```
.SOURCE.%.STD.INCLUDE : .FORCE $$(*.SOURCE.h)
```

.SOURCE.*pattern* Special Atom

.SOURCE.*pattern*

Type

Pattern Association

Description

Prerequisites of .SOURCE.*pattern* are directories to be checked for files with the suffix *.pattern*. If no matching file is found, the directories specified by .SOURCE are checked for the file.

The first matching file found in a listed directory in left-to-right order is used. The current directory (.) is always searched first.

When searching for files that are implicit prerequisites (i.e., C language #include header files), nmake first looks in the current directory (.), then in the .SOURCE directories, and finally in the .SOURCE.*pattern* directories.

If the #include indicates the name of the file between angle brackets (<>) instead of in quotes, only the .SOURCE.*pattern* directories will be searched since they are the standard locations in which to find included files.

Example

In the following example, nmake searches for implicit prerequisites in the paths specified in the makefile.

Contents of Makefile

```
.SOURCE.h : ../include ../../include
hello :: hello.c
```

Run nmake

```
$ nmake
```

Output

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

.SPECIAL Special Atom

.SPECIAL - do not add target atoms to prerequisites of .MAIN

Type

Assertion Attribute

Description

When this Special Atom is specified, the target atoms in the assertion are not appended to the prerequisites of .MAIN, and diagnostics about multiple actions are inhibited. This rule allows the user to redefine existing rules without triggering error messages.

Target names that begin with "." are considered .SPECIAL.

Examples

The following examples show how .SPECIAL changes the default target and the diagnostic information.

In the first example, .SPECIAL is not used. nmake sees x as the default target and performs only the last action for that target. A diagnostic message is issued because multiple actions exist.

Contents of Makefile

```
x :
        : First Action

x :
        : Last Action

test1 :: test1.c
```

Run nmake

```
$ nmake
```

Output

```
make: "Makefile", line 7: warning: multiple actions for x
+ : Last Action
```

In the second example, the .SPECIAL attribute is given to the second assertion of target x. nmake still sees x as the default target and performs only the last action for that target, but this time no diagnostic message is issued.

Contents of Makefile

```
x :
        : First Action

x : .SPECIAL
        : Last Action

test1 :: test1.c
```

Run nmake

```
$ nmake
```

Output

```
+ : Last Action
```

In the third example, the .SPECIAL attribute is given to both assertions of target x. As a result, x is not added to the target list of .MAIN. nmake sees test1 as the default target and performs the action for only that target.

Contents of Makefile

```
x : .SPECIAL
        : First Action

x : .SPECIAL
        : Last Action

test1 :: test1.c
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c test1.c
+ cc -O -o test1 test1.o
```

.STATE (Dynamic Attribute) Special Atom

.STATE - marks state atom

Type

Readonly Attribute

Description

This Special Atom marks atoms which have either the .STATEVAR or the .STATERULE attribute.

Example

The following example prints the tokens of variable X that have the .STATEVAR attribute and the tokens of variable X that have the .STATE attribute. (CC) is a state variable defined in the default base rules. (XYZ) is defined in the makefile. Since XX is not a state variable, it is not listed.

Contents of Makefile

```
(XYZ) :

XX = xx
x = (XYZ) (XX) (CC)

target:
        : '$(x:A=.STATEVAR)' have the .STATEVAR attribute
        : '$(x:A=.STATE)' have the .STATE attribute
```

Run nmake

```
$ nmake
```

Output

```
+ : (XYZ) (CC) have the .STATEVAR attribute
+ : (XYZ) (CC) have the .STATE attribute
```

.STATE (Immediate Rule) Special Atom

.STATE - interpret prerequisites as state variables

Type

Immediate Rule

Description

This Special Atom causes the prerequisites to be interpreted as variable names marked as candidate implicit state variable prerequisites (i.e., prerequisite names that are checked for usage by the file being scanned for implicit prerequisites). nmake interprets these prerequisites in the same way as it interprets variables initialized with the == operator.

Example

In the following example, X and Y are interpreted as state variables, but Z is not. nmake would generate the -D flag for X and Y only, if these symbols are used in progx.c.

Contents of Makefile

```
.STATE : X                   /* define the state variable */

Y == 1                       /* initialize variables */

X = 1
Z = 1

progx :: progx.c             /* define the target */
```

Run nmake

```
$ nmake
```

Output

```
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -DX -DY -c progx.c
+ cc -O -o progx progx.o
```

.STATERULE Special Atom

.STATERULE - makes internal state rule atoms

Type

Readonly Attribute

Description

This Special Atom marks internal state rule atoms used to store state information.

Example

This atom is not commonly used in user makefiles. It is documented here only for completeness.

.STATEVAR Special Atom

.STATEVAR - marks state variable atoms

Type

Readonly Attribute

Description

This Special Atom marks state variable atoms (i.e., those atoms associated with nmake variables).

Example

See the example for .STATE (Dynamic Attribute).

.SYNC Special Atom

.SYNC - synchronize statefile

Type

Immediate Rule

Description

This atom is used for synchronizing the statefile. It can be used with .ALARM and .INTERRUPT.ALRM to set a checkpoint periodically to synchronize the statefile during long builds.

Example

The following example synchronizes the statefile every 60 seconds.

```
.INTERRUPT.ALRM: .MAKE
        print "Synchronizing Statefile"
        .SYNC :
        .ALARM: 60              /* repetitive alarm */
.ALARM : 60                     /* initial alarm */
```

.TARGET Special Atom

.TARGET - marks target atoms

Type

Readonly Attribute

Description

This Special Atom marks atoms that appear as the target of an assertion.

Example

In the following example, targets a, b, and c are given the .TARGET attribute.

Contents of Makefile

```
TARGETS = a b c d                 /* initialize the variable */

all : a c                         /* define the targets */

a :COPY: aa

b :COPY: bb

c :COPY: cc

.DONE :           /* define the action for .DONE to list all targets */
        : $(TARGETS:A=.TARGET)
```

Run nmake

```
$ nmake all
```

Output

```
+ rm -f a
+ ln aa a
+ rm -f c
+ ln cc c
+ : a b c
```

.TERMINAL Special Atom

.TERMINAL

Type

Dynamic Attribute

Description

For files, this Special Atom allows only .TERMINAL metarules to be applied to the target atom when determining metarule prerequisites. Normally, metarules are applied to targets with no explicit actions and prerequisites. Using .TERMINAL on a file will restrict this set further by enforcing a correlation with .TERMINAL metarules. This attribute turns the .IMPLICIT attribute off at assertion time. For metarule assertions, .TERMINAL can be used to mark unconstrained metarules that can be applied only to .TERMINAL targets or non-generated source files.

For directories, this attribute causes nmake to mark each directory as having no subdirectories. This attribute is used to optimize processing when there are many directories that do not have subdirectories and many files with directory prefixes.

Examples

In the following example, the .TERMINAL metarule %.c : s.%.c is defined and is applied to the :: assertion, since .TERMINAL is automatically on for all the prerequisites. The SCCS get command retrieves cmd.c from the SCCS s-file s.cmd.c and compiles it.

Without .TERMINAL on the metarule, this metarule cannot be applied to the :: assertion, since cmd.c has the .TERMINAL attribute as a result of the :: operator. Thus, nmake would fail to build pgm because cmd.c does not exist.

Contents of Makefile

```
%.c : s.%.c .TERMINAL
        get $(>)
pgm :: cmd.c
```

Run nmake

```
$ nmake
```

Output

```
1.1
32 characters
+ cc -O -Qpath /nmake3.x/lib -I-D/nmake3.x/lib/probe/C/pp/\
B49DF4E0.bincc -I- -c cmd.c
+ cc -O -o pgm cmd.o
```

In the following example, the target m.x.y is made from m.x.z, which is the existing source file. According to the metarules specified in the makefile, m.x has to exist before m.x.y is made. Since .TERMINAL applies to the unconstrained % : %.z metarule, nmake knows to trigger this rule before triggering the %.x.y : %.x metarule.

Contents of Makefile

```
% : %.z .TERMINAL
        : make % from %.z
        $(CP) $(>) $(<)
%.x.y : %.x
        : make %.x.y from %.x
        $(CP) $(>) $(<)
```

Run nmake

```
$ nmake m.x.y
```

Output

```
+ : make % from %.z
+ cp m.x.z m.x
+ : make %.x.y from %.x
+ cp m.x m.x.y
```

In the following example, we assume the existence of a large number of files that include header files such as:

```
#include "dir1/a.h"
```

and dir1 is not a token of .SOURCE.h.

Contents of Makefile

```
.SOURCE.h : A B C D
A : .TERMINAL
B : .TERMINAL
C : .TERMINAL
t :: a.c b.c c.c ... zz.c
```

Result

Each time a #include directive is encountered, nmake knows not to waste time searching for dir1 in directories A, B, and C. When a large number of files is involved, the processing time saved by this use of .TERMINAL is significant.

The following example illustrates another way in which .TERMINAL can save on processing.

Contents of Makefile

```
SRC = db/foo.c
.SOURCE.c : $(VROOT)/src1 $(VROOT)/src2
$(VROOT)/src1 : .TERMINAL
.MAIN : foo
foo :: $(SRC)
```

Result

Because the $(VROOT)/src1 directory has the .TERMINAL attribute, nmake does not attempt to bind foo there and looks only in $(VROOT)/src2 to bind it.

.TMPLIST Special Atom

.TMPLIST - atom whose prerequisite is an internally used list

Type

Readonly List

Description

This Special Atom is only used internally and appears here for completeness.

.TRIGGERED Special Atom

.TRIGGERED - marks triggered atoms

Type

Readonly Attribute

Description

This 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.

Example

The following example shows how the triggered attribute is given to targets whose actions have begun.

Contents of Makefile

```
TARGETS = a b c d e                 /* initialize the variables */

a : b                               /* define the targets */
        touch $(<)
b : c
        touch $(<)
c :
        touch $(<)
d : e
        touch $(<)
e :
        touch $(<)

.DONE : /* define the action for .DONE to list all triggered targets*/
        : $(TARGETS:A=.TRIGGERED)
```

Run nmake

```
$ nmake
```

Output

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

.UNBIND Special Atom

.UNBIND - unbind atom prerequisites

Type

Immediate Rule

Description

This Special Atom causes each prerequisite to be unbound as if it had not been bound. It is useful if the nmake binding algorithm parameters have been changed (e.g., if .SOURCE.*pattern* has been modified).

An unbound directory will have the directory and its contents removed from the internal directory cache.

Example

The following example shows how the binding performed by nmake can be unbound.

Contents of Makefile

```
.BIND : VAR1                  /* assert .BIND with VAR1 */

T = VAR1                      /* initialize the variables */
U = VAR2

all : target1 target2 target3 target4 target5/*define the targets*/

target1 :
        : making $(<)
        : $(T:A=.BOUND)
        : $(U:A=.BOUND)

target2 : .MAKE
        .UNBIND : VAR1

target3 :
        : making $(<)
        : $(T:A=.BOUND)
        : $(U:A=.BOUND)

target4 : .MAKE
        .BIND : VAR1

target5 :
```

```
                    : making $(<)
                    : $(T:A=.BOUND)
                    : $(U:A=.BOUND)
```

Run nmake

```
$ nmake all
```

Output

```
+ : making target1
+ : VAR1
+ :
+ : making target3
+ :
+ :
+ : making target5
+ : VAR1
+ :
```

.USE Special Atom

.USE - use atom’s action when used as prerequisite

Type

Dynamic Attribute

Description

This Special Atom marks the target as a .USE atom. Any target having a .USE atom as a prerequisite will be built using the .USE atom action and attributes. If an action is specified, the action takes precedence, and the .USE rule is ignored.

Example

In the following example, the link target is marked with the .USE attribute; actions defined once for the link target can be used for both targets prog1 and prog2. Note that the automatic variables in the link actions are used in the same manner as if the action were directly listed with the parent (i.e., $(<) refers to prog1 and prog2 in the respective runs.)

Contents of Makefile

```
program : prog1 prog2
prog1 : link a.c b.c
prog2 : link a1.c
link : .USE
        $(CC) $(CCFLAGS) -c $(>)
        $(AR) $(ARFLAGS) $(<) $(>:B:S=.o)
```

Run nmake

```
$ nmake
```

Output

```
+ ppcc -i /tools/nmake/sparc5/3.x/lib/cpp cc -O -I-D/tools/\
nmake/sparc5/3.x/lib/probe/C/pp/890893F4obincc -I- -c a.c b.c
a.i:
b.i:
+ ar r prog1 a.o b.o
ar: creating prog1
+ ppcc -i /tools/nmake/sparc5/3.x/lib/cpp cc -O -I-D/tools/\
nmake/sparc5/3.x/lib/probe/C/pp/890893F4obincc -I- -c a1.c
+ ar r prog2 a1.o
ar: creating prog2
```

.VIEW Special Atom

.VIEW - atom whose prerequisites are view directory paths

Type

Readonly List

Description

The current directory(**.**) is by default the first view directory path.

.VIEW is initialized from the MAKEPATH and VPATH environment variables. .VIEW is a readonly atom and cannot be altered.

Example

The following example shows how to print the prerequisites of .VIEW. The viewpath environment variable is:

```
VPATH=/view/d1:/view/d2:/view/d3
```

VPATH has been set to three nodes and .VIEW has three directories in the prerequisite list. The contents of VPATH and .VIEW are searched from left to right.

Contents of makefile in directory /view/d2

```
target :
        : in d2
        : .VIEW = $(*.VIEW)
        cd dir
        $(MAKE)
```

Contents of makefile in directory /view/d2/dir

```
target :
        : in d2/dir
        : .VIEW = $(*.VIEW)
```

The dir directory should exist in all VPATH directories, i.e., /view/d1/dir, /view/d2/dir, and /view/d3/dir.

Run nmake from directory /view/d1, which has no makefile

```
$ nmake
```

Output

```
+ : in d2
+ : .VIEW = . /view/d2 /view/d3
+ cd dir
+ nmake
+ : in d2/dir
+ : .VIEW = . /view/d2/dir view/d3/dir
```

.VIRTUAL Special Atom

.VIRTUAL - neither bind nor generate target atom

Type

Dynamic Attribute

Description

A target atom marked by the .VIRTUAL Special Atom is not generated and is never bound to a file. This capability allows a list of prerequisites to be maintained even when the target is not created. The target atom's time-stamp is saved in the statefile.

Example

The following example shows the use of the .VIRTUAL atom on a target named list. The difference between the targets list and list1 is that the time-stamp of list is kept in the statefile.

The first time nmake is run, both shell actions are executed, because the targets list and list1 have not yet been made.

The second time only the target list1 is made. Because it does not use .VIRTUAL, and no list1 file was generated, nmake will try to generate the list1 target again. Notice $(>) expands to null because a.c and b.c are not out-of-date.

Before nmake is run for the third time, a.c is touched to change its time-stamp. At that point a.c is out-of-date and both actions are triggered only for a.c.

Contents of Makefile

```
all : list list1
list : a.c b.c .VIRTUAL
        : making target - list
        : $(>)
list1 : a.c b.c
        : making target - list1
        : $(>)
```

Run nmake

```
$ nmake
```

Output

```
+ : making target - list
+ : a.c b.c
+ : making target - list1
+ : a.c b.c
```

Run nmake again

```
$ nmake
```

Output

```
+ : making target - list1
+ :
```

Touch a.c

```
$ touch a.c
```

Run nmake again

```
$ nmake
```

Output

```
+ : making target - list
+ : a.c
+ : making target - list1
+ : a.c
```

- Special Atom

– - control synchronization between target atom’s prerequisites

Type

Dynamic List

Description

This Special Atom is used to control synchronization when making a list of prerequisites. When – appears in a prerequisite list, nmake waits until all preceding prerequisite's actions are complete before continuing.

Example

In the following example, nmake is executed with the jobs command-line option set to 3, which setting specifies that nmake can execute three jobs concurrently. The – is specified in the prerequisite list for the all targets; this specification informs nmake to build the a and b prerequisites before building the c prerequisite.

Contents of Makefile

```
TARGETS = a b c

all : a b - c

$(TARGETS) :
        sleep 10
        : $(<)
```

Run nmake

```
$ nmake -o jobs=3
```

Output

```
+ sleep 10
+ sleep 10
+ : a
+ : b
+ sleep 10
+ : c
```