Nokia nmake Product Builder

Tutorial: A Little Help With Nokia nmake

December 28, 1995

Abstract

Nokia **nmake** is a Unix program that builds other programs. It's powerful and popular, but getting started can be difficult. This paper is written for beginners and will help you learn about **nmake** without tackling difficult problems. When you finish you'll understand simple makefiles and have a background that will let you make good use of the **nmake** manuals.

Table of Contents

---

1. Introduction

nmake is a Unix program that builds other programs. It began over a decade ago as an improved **make**, and it has evolved steadily over the years. Today's version is powerful and sophisticated, and gives you lots of help building software and keeping everything up to date. **nmake** is popular, but getting started can be difficult; all too often it ends up as a big black box that's documented by a mysterious manual. But **nmake** is the best build engine you can find - spend some time learning how it works and you will be amply rewarded.

This paper is aimed at beginners, but everyone has to bring some baggage. We'll assume you can program the shell, that you can read and write simple C programs, and that you know something about building software. The bottom line is you need to be a programmer; if you're not you won't get much out of this paper. We'll try to teach you things about **nmake**, and your job, at least at the start, is to believe you can learn something important from our trivial examples. In fact, we won't build a program until late in the paper, and by that time you will have learned a great deal.

It's important that you trust our examples, so we automated whenever possible. Makefiles included in this paper are, for the most part, separate files. Most results were automatically generated by running **nmake** on the example makefile, capturing the output in a file, and then including that file in the paper. We managed everything with **nmake** - you'll find some of the details in Appendix A.

We're almost ready, but first a warning: what you read here shouldn't be viewed as gospel, and our examples, even simple ones, won't always be completely correct. We've taken liberties when we felt the full truth would complicate matters. If you notice omissions, particularly early in the paper, you're probably not in our target

audience. On the other hand, if everything seems fine remember you only have part of the story, so don't toss your manual.

---

2. An Example

Let's start with some simple instructions that tell **nmake** to write a message on our screen. Anything will do, so we'll be traditional and put

```
hello :
     echo "hello, world"
```

in a file named `hello.mk`. Get the positioning right when you type it in - `hello` starts in column 1 and `echo` is indented by a tab. Blank lines and extra tabs or spaces won't usually matter; you can even get rid of white space around the colon.

2.1 Background

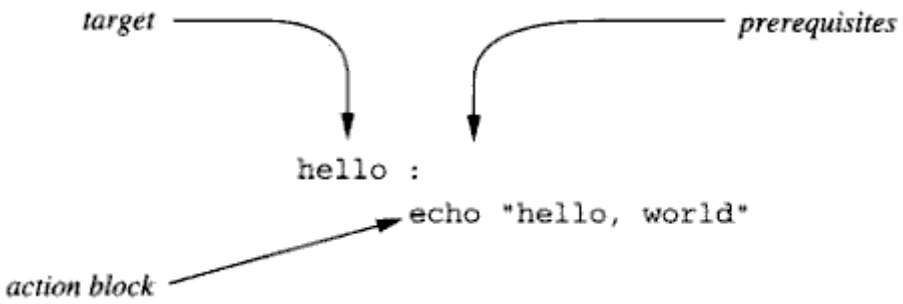
We need some common ground, mostly simple definitions and a few words about **nmake**, before we can go much farther. Don't worry, we'll keep it short so you'll hardly notice.

2.1.1 Makefiles

A file like `hello.mk`, with stuff in it that means something to **nmake**, is called a *makefile*. The name can be anything you want. Popular choices are `Makefile`, `makefile`, or a name that ends in `.mk`. `Makefile` and `makefile` are the defaults, so they'll save you a few keystrokes when you run **nmake**; more than one makefile in a directory is a good reason to use a suffix.

2.1.2 Assertions

The two lines in `hello.mk` constitute an *assertion*. The components of the assertion in our example are shown below:



It's easy to parse assertions, even in complicated makefiles. The *target* list is a single line that starts in the first column and ends at a colon; *prerequisites* follow the colon, usually on the same line. Indented lines after the prerequisites are called the *action block*; the end of the makefile or the first line that's not indented (blank lines and comments don't count) ends the action block. Names that appear as targets or prerequisites are collectively called *atoms*.

All three components are optional. An empty prerequisite list, as in our example, or a missing action block, is not unusual. But an assertion with no target list isn't particularly useful, and if you find one you may be looking at a mistake. Don't be surprised to see the same target in several assertions: normally one assertion will have an action block and the others just add prerequisites to the target.

The assertions in a makefile usually describe the components of a software project. Targets often refer to programs or libraries, and have source files as prerequisites and shell commands that build the target from the source files as action blocks. Obviously our example is an exception: `hello` is the target, but it's not the name of a program, there aren't any prerequisites, and the action block doesn't build anything.

You'll often use words like "the *target* depends on the *prerequisites*," when you're describing an assertion. For example, if you were reading

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

you would probably say "`prog` depends on `a.c` and `b.c`."

2.1.3 Looks Can Be Deceiving

Programs like **make** and **nmake** read makefiles and try to build targets as correctly and efficiently as possible. "Correctly" means they follow instructions, build everything that's needed, and stop if something goes wrong. "Efficiently" means they try to avoid unnecessary work, so both programs deal with prerequisites before targets, and in most cases only build a target, usually by handing an action block to the shell, when they find a prerequisite that's newer (i.e., younger) than the target.

But the two programs are very different, and their treatment of prerequisites is a prime example. They're usually files, but **nmake** gives you an easy way to include things, like compiler options, as prerequisites. It's important, but it means extra work for **nmake**. The information needs to be preserved between runs, but it's not necessarily in the makefile or directly available from a permanent resource, like the file system. But with **make** there's no easy way to associate abstract things, like compiler options, with programs or object files.

nmake also knows how to look through source files to find implicit prerequisites, such as header files in C programs. If a source file includes a header file and that header file changes, then most people would agree the source file should be recompiled. Header files often include other files, so dependencies can get complicated, but **nmake** figures them all out automatically.

make, on the other hand, only knows what it reads in your makefile, and that means dependencies encoded as `#include` directives in source files also need to appear as assertions in makefiles. The duplication can be a source of errors. Change a source file and you may also have to update a makefile; but header files are often shared, so it's not just a matter of one source file affecting one makefile.

2.2 Getting Started

Setting things up is easy - if **nmake** is in your `PATH` you're ready to go.[1] **nmake** can be installed anywhere, so check with your system administrator if you can't find it. What's in your environment is also important, but right now there aren't any magic shell variables you'll need to define and export.

We want you to participate, so set your `PATH` up and follow along. You'll get the most out of the paper if you type the examples in, run **nmake**, experiment a bit, and make mistakes. As you read along, either here or in the manual, think about *eliminating duplication*. It's an important theme and keeping it in mind will help you appreciate this tool.

2.3 Running nmake

We can run our example by typing

```
nmake -f hello.mk hello
```

and we get:

```
+ echo hello, world
hello, world
```

That's lots of work for a simple greeting and we ended up with more than we really wanted. We'll talk about the noise and how to get rid of it shortly, but first a few words about the command line.

2.3.1 The Option

We used the `-f` option to point at our makefile. The white space separating `-f` and `hello.mk` isn't needed [2] but the option is. If we leave it out

```
nmake hello
```

we get:

```
make: a makefile must be specified when Makefile,makefile omitted
```

When you don't choose a makefile **nmake** looks for `Makefile` and then `makefile`, and complains if it can't find either file. By the way, you can get the same error message if you're using viewpathing, which is something we'll talk about later in the paper, and your `VPATH` shell variable is wrong or just not exported.

2.3.2 The Argument

The `hello` argument tells **nmake** what target to build. Move it left

```
nmake hello -f hello.mk
```

or leave it out

```
nmake -f hello.mk
```

and nothing changes. **nmake** reads your makefile before it builds targets, and usually picks the first target in your makefile when you don't tell it what to do. The real story is more involved - we'll come back to it when we talk about `.MAIN` later in the paper.

2.4 The Noise

We can quiet things down from the command line using the `-s` option

```
nmake -f hello.mk -s
```

or by sending standard error to `/dev/null`:

```
nmake -f hello.mk  2>/dev/null
```

Either way we get,

```
hello, world
```

which is what we originally wanted. If you're familiar with **make** you may recognize the `-s` option, but redirecting standard error is new, because **make**'s noise shows up on standard output.

2.4.1 `silent`

We can also control the noise from a makefile. Putting `silent` in front of a simple shell command [3] stops the noise, but only for that command. For example, we could put

```
hello :
     silent echo "hello, world"
```

in a file named `silent.mk`, type

```
nmake -f silent.mk
```

and end up with:

```
hello, world
```

That's the right answer again, but this time we got it without doing anything special on the command line. We'll use `silent` in many of our examples, mostly to avoid cluttering command lines with `-s` options or file redirection.

We should also mention there's a related command named `ignore`, that tells **nmake** to keep going when a shell command fails. Learn to use `silent` and you'll know how to use `ignore`, and if you're familiar with **make**'s `@` and `-` special characters you'll probably already appreciate `silent` and `ignore`.

2.5 Another Assertion

Let's add a second assertion to our example. We don't need to get fancy, so another simple message will do. If we put

```
goodbye :
   silent echo "goodbye, world"

hello :
   silent echo "hello, world"
```

in a file named `goodbye.mk`, then we can type

```
nmake -f goodbye.mk goodbye
```

when we want to say `goodbye`. Name two different targets on the command line

```
nmake -f goodbye.mk hello goodbye
```

and we get two messages:

```
hello, world
goodbye, world
```

Order makes a difference - rearrange the command line and see for yourself. Try more than one `hello` (or `goodbye`)

```
nmake -f goodbye.mk hello hello
```

and you may be surprised by what happens. **nmake** usually only builds a target once per invocation, but in this case that's not the real explanation. We'll give you the full story later when we talk about `.ARGS`.

2.6 Some Easy Mistakes

Making mistakes is an important part of learning, particularly when you're trying to master a complicated subject like **nmake**. We've already talked about one mistake (forgetting the `-f` option); there are a few others that deserve a brief mention.

2.6.1 A Missing Makefile

Point **nmake** at a makefile that doesn't exist

```
nmake -f missing.mk
```

and we get:

```
make: missing.mk: cannot read
```

2.6.2 No Targets

If we create an empty file and try use it as a makefile

```
>empty.mk
nmake -f empty.mk
```

we get,

```
make: empty.mk: a main target must be specified
```

because **nmake** usually complains, no matter what's in the makefile, if it doesn't find at least one assertion.

2.6.3 A Missing Target

Try to build a target that's not in a makefile

```
nmake -f hello.mk goodbye
```

and **nmake** complains with:

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

2.6.4 A Missing Prerequisite

We can get the same kind of error message when there's a mistake in a makefile. For example, put

```
hello : greeting
   echo "hello, world"
```

in a file named `mistake.mk` and type

```
nmake -f mistake.mk hello
```

and we get:

```
make: don't know how to make hello : greeting
```

nmake builds prerequisites before targets, but there's nothing in `mistake.mk` that explains how to build `greeting`, and that's why **nmake** complained.

Try to understand the error message, because you'll see it again and again. The colon-separated list is how **nmake** tells you what went wrong, and it's not just a copy of the first line of the assertion. The list can get long, but it always describes how **nmake** got from the target it was trying to build (the first name) to the prerequisite that caused the problem (the last name) [4]

2.6.5 An Action Block Failure

Commands that return a non-zero exit status usually stop **nmake**. For example, put

```
failed :
   cat /xxx/yyy
```

in `failed.mk` and type

```
nmake -f failed.mk failed
```

and we get:

```
+ cat /xxx/yyy
cat: cannot open /xxx/yyy: No such file or directory
make: *** exit code 2 making failed
```

nmake quit because `cat` exited with a non-zero status. Use `ignore` when you don't care about errors or when you run commands, like `grep`, that can return a non-zero exit status when there aren't any errors.

2.7 Variables

We've talked a little about assertions; now it's time to introduce variables [5] Put

```
AUDIENCE = world

goodbye :
   silent echo "goodbye, $(AUDIENCE)"

hello :
   silent echo "hello, $(AUDIENCE)"
```

in `variable.mk`, type

```
nmake -f variable.mk goodbye
```

and we get,

```
goodbye, world
```

which is exactly what happened in the last makefile. That's good news, but there's lots to explain.

2.7.1 Names

Names are usually made up of letters, digits, underscores, and periods. They can be as long as you want and all characters are significant [6] Be careful with periods, particularly at the beginning or end of upper case names - **nmake** has claimed some of them. Variable and target names are completely independent, so confusing makefiles like

```
hello = world

hello :
   echo hello $(hello)
```

are allowed, but overloading names is bad style. Instead, we recommend you pick a convention that visually separates the name spaces, say upper case variables and lower case targets, and stick with it as long as possible. But remember, targets can refer to programs or files that you don't control, so you won't always be able to follow strict rules.

2.7.2 Not Quite Reserved

A few words mean something special to **nmake** [7] - at this point the important ones are:

```
break      else       eval       include    print      rules
continue   end        for        let        read       set
elif       error      if         local      return     while
```

They're not officially reserved, but you'll have trouble using them as variable or target names. Double quotes around any of the special words, as in

```
"print" :
   echo "the target must be quoted"
```

is one solution. Forget the quotes and **nmake** usually complains, sometimes in a way that depends on the unquoted word, and other times in a way that depends on the context near the mistake. These errors are confusing, so get some experience. See what happens when you remove the quotes; then try substituting other special words, like `error` and `include`, for `print`.

2.7.3 Assignment Operators

nmake supports string and integer variables, and five different assignment operators. String variables and three assignment operators will get us through most of this paper.

The `=` and `:=` operators assign a new value to a variable; the `+=` operator appends a space and a value to whatever's currently stored in a variable. The `:=` and `+=` operators share an important property that we'll talk about when we get to the section on variable expansion. Until then, we'll stick with `=` when we want to assign a new value to a variable.

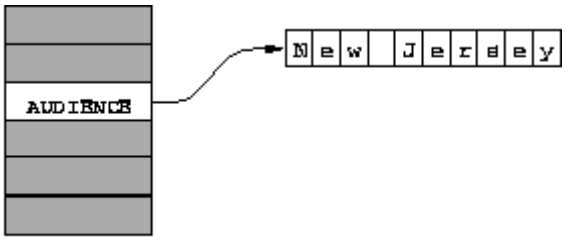
The strings picked up by assignment operators start right after the operator and go to the end of the line; a backslash at the end of the line means it's continued on the next line, though the newline itself is discarded. Assignment operators also remove leading and trailing white space from strings before they carry out an assignment, so there's no difference between

```
AUDIENCE=New Jersey
```

and

```
AUDIENCE =      New Jersey
```

Either way we would find the string definition shown in the picture below if we could look through **nmake**'s variable symbol table.



Variable Symbol Table

We could build the same string up in steps using the `=` and `+=` operators. The two assignment statements

```
AUDIENCE = New
AUDIENCE += Jersey
```

do the job, but only because = overwrites the existing definition and += adds a single space before appending Jersey. We could even do it in three steps

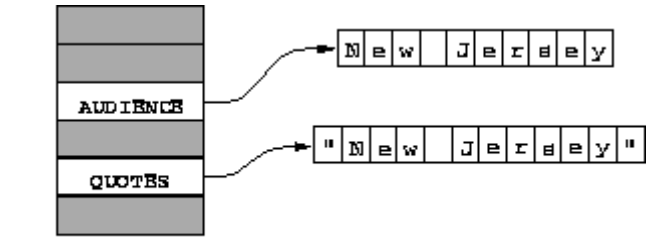
```
AUDIENCE =
AUDIENCE += New
AUDIENCE += Jersey
```

because nothing on the right side of = clears the definition, and that means we won't see the space separator from the first += assignment statement.

Be careful about bringing too much of your C or shell programming experience along when you talk to **nmake**. For example, quotes need to be balanced, but they're not string delimiters as they are in C, so

```
QUOTES = "New Jersey"
```

defines another string, but this one has a double quote at each end. Once again, if we could look through **nmake**'s variable symbol table we would find the two definitions shown in the following picture:



Variable Symbol Table

2.7.4 Referencing Variables

Putting $(and) around a variable name, as we did with $(AUDIENCE), is how we ask **nmake** for the value represented by the variable. The replacement process is officially called *variable expansion*, and *when* it really happens is an important and confusing topic. We will postpone our discussion for a few sections. Until then, just believe **nmake** expands the variables it finds in an action block right before it hands anything to the shell.

We'll often say "variable reference" when we're talking about expressions, like $(AUDIENCE), that will be expanded by **nmake**. If we want some variety we may call it a reference to a particular variable: in this case the words would be, "a reference to AUDIENCE." It's convenient terminology, but you won't find it defined in the manual's glossary or listed in the index, so it's not officially blessed.

References to undefined variables are quietly replaced by empty strings, so don't expect warning messages about typing mistakes. It means mistakes can linger until **nmake** tries to execute the offending code, and even then you may not notice.

2.7.5 Command Line Assignments

Variables can be defined on the command line, so

```
nmake -f variable.mk AUDIENCE='New Jersey' hello
```

prints:

```
hello, New Jersey
```

How we arrange the command line doesn't make much difference. Move the assignment right

```
nmake -f variable.mk hello AUDIENCE='New Jersey'
```

or left

```
nmake AUDIENCE='New Jersey'  -f variable.mk hello
```

and nothing changes. Command line assignments are handled after **nmake** reads your makefile, and that makes it easy to override hard-coded definitions. There's nothing special about the `=` operator; you can use `+=` or any other assignment operator on the command line. In fact, you can even do complicated things like define assertions on the command line, but don't get carried away because there aren't many good reasons to do so.

2.7.6 An Automatic Variable

It's been a while, and what we're going to talk about next is very important, so here's `variable.mk` again:

```
AUDIENCE = world

goodbye :
   silent echo "goodbye, $(AUDIENCE)"

hello :
   silent echo "hello, $(AUDIENCE)"
```

Do you notice any duplication? If we change the name of a target, say from `goodbye` to `farewell`, we probably would also want to edit the action block and update the arguments of the `echo` command. Target names are mentioned in action blocks, and that duplication means extra work and more opportunity for mistakes. A mechanism that would let us talk about the components of an assertion (e.g., the target or prerequisites) in an action block without actually mentioning names would help.

There are about a dozen **nmake** variables, called automatic variables, that are designed to be used in action blocks. They're automatically assigned values by **nmake** and many are closely connected to the target that's being built. Automatic variables have cryptic names - only a few are easy to remember. The one we need is `$(<)`, which happens to be one of the easy ones. `$(<)` stands for the name of the target we're building, so the last example can be written as:

```
AUDIENCE = world

goodbye :
   echo "$(<), $(AUDIENCE)"

hello :
   echo "$(<), $(AUDIENCE)"
```

That's a good start, but there's more. The two assertions now have identical action blocks, so we can combine them

```
AUDIENCE = world

goodbye hello :
   silent echo "$(<), $(AUDIENCE)"
```

and eliminate the last bit of duplication, because each target mentioned in an assertion inherits its own copy of the prerequisites and the action block. Automatic variables are a valuable resource, so make sure you look for similar opportunities in your own makefiles. As simple as this example is, it's not quite right. We'll save it in a file named `message.mk` and fix it up later when we introduce `.FORCE` and `.VIRTUAL`.

2.8 Variable Expansion

We've already relied on variable expansion in several examples - now it's time for the details.

2.8.1 The General Idea

nmake expands a variable reference, like `$(AUDIENCE)`, by copying the variable's current definition into a buffer. The process is repeated if **nmake** finds a variable reference while it's copying the definition, so expanding one variable can trigger the expansion of another variable, and so on. It's not hard to imagine problems:

```
AUDIENCE = you and $(AUDIENCE)

hello :
   silent echo "hello, $(AUDIENCE)"
```

The definition of `AUDIENCE` includes a reference to `AUDIENCE`, so the process we just described looks like it might never end. Let's find out for sure - we can always hit interrupt if we get stuck. Save the example in a file named `recursive.mk`, type

```
nmake -f recursive.mk hello
```

and we get:

```
make: AUDIENCE: recursive variable definition
```

The error message is good news; **nmake** caught the problem and warned us. It's even easy to figure out when **nmake** noticed the mistake. Take the variable reference out of the action block

```
AUDIENCE = you and $(AUDIENCE)

hello :
   silent echo "hello, world"
```

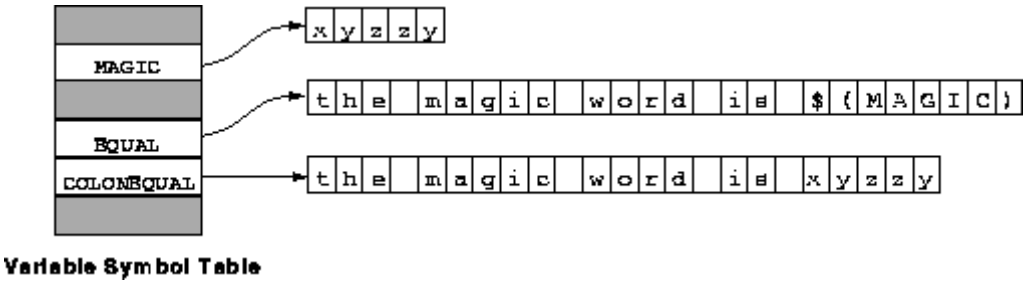
and the error message goes away, so **nmake** only complains about recursive variable definitions if it tries to use them.

2.8.2 In Assignment Statements

The `=` and `:=` operators replace existing definitions by new ones, but they behave differently when they find variable references. For example, start with

```
MAGIC = xyzzy
EQUAL = the magic word is $(MAGIC)
COLONEQUAL := the magic word is $(MAGIC)
```

and look through **nmake**'s variable symbol table and we would find:



Variable Symbol Table

There's a variable reference left in `EQUAL`, but not in `COLONEQUAL`, because the `:=` operator expands variable references, but `=` doesn't. Refer back to our description of the expansion process in the last section and you should appreciate the consequences: assign a new value to `MAGIC` and **nmake** gives back a different result when it expands `$(EQUAL)`, but there's nothing left to expand in `COLONEQUAL`, so `$(COLONEQUAL)` doesn't change.

By the way, the += operator works just like := when it finds a variable reference, so we wouldn't see a difference if we typed

```
PLUSEQUAL += the magic word is $(MAGIC)
```

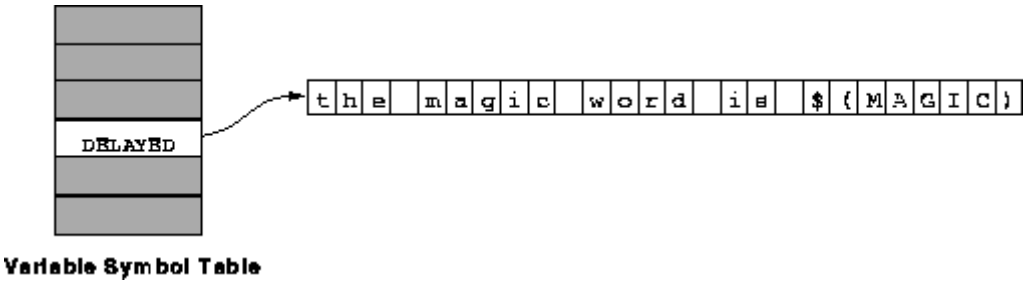
and then compared COLONEQUAL and PLUSEQUAL (assuming PLUSEQUAL started out undefined).

2.8.3 Adjusting Expansion Time

There are simple techniques that let you delay or trigger a variable expansion. Each extra dollar sign in front of a variable reference postpones the expansion one step, so after **nmake** reads

```
DELAYED := the magic word is $$(MAGIC)
```

we would find



in the variable symbol table. One dollar sign was removed by the := operator, but the expansion of $(MAGIC) has been delayed and we ended up with a string named DELAYED that looks exactly like EQUAL.

There's also an easy way to force variable expansion. We'll mention it here for completeness and never talk about it again, because it's a feature few users ever need. Surrounding one or more **nmake** statements with eval and end forces an additional variable expansion. eval and end can be important if you're doing complicated things, like writing your own assertion operators, but not many of you ever will, so we won't even include an example.

2.8.4 In Action Blocks

Variable references in an action block are expanded when **nmake** executes the action block. The implementation is straightforward: **nmake** copies the action block into a temporary buffer, expands variable references when it finds them, and then usually hands whatever's in the temporary buffer to the shell. Using an action block doesn't change **nmake**'s copy, so variables are expanded each time the action block is used.

2.8.5 In Target Lists

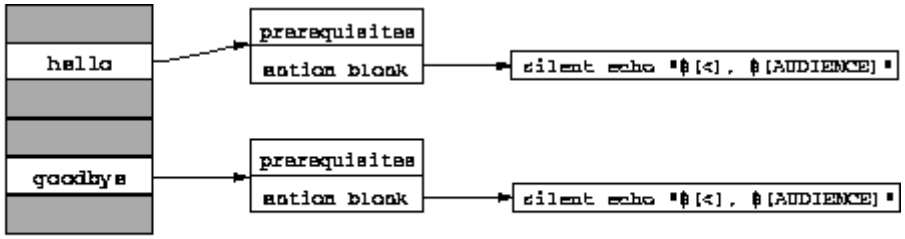
Variable references in a target list are expanded when **nmake** builds its internal representation of the assertion. Here's an easy example that we'll save in a file named targets.mk:

```
AUDIENCE = world
TARGETS = goodbye hello

$(TARGETS) :
   silent echo "$(<), $(AUDIENCE)"
```

We started with message.mk, added a variable named TARGETS, and referenced TARGETS in the assertion. When **nmake** reads targets.mk and gets to the assertion, it expands $(TARGETS), ends up with goodbye and hello, and

from that point on behaves exactly like `message.mk`. If we could look through **nmake**'s assertion symbol table we would find the two definitions shown in the following picture:



Assertion Symbol Table

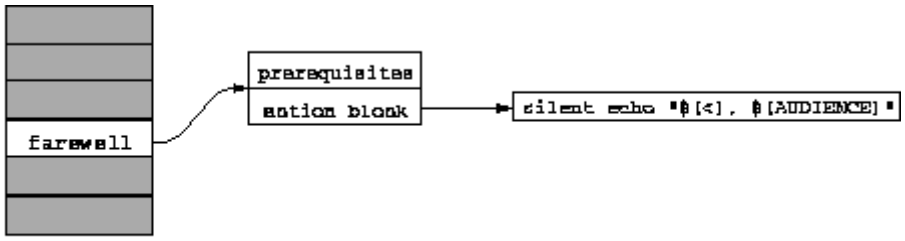
It usually won't matter which makefile we use, even when we make a mistake, but there is an important difference: `TARGETS` is a variable and it can be defined on the command line, so when we type

```
nmake -f targets.mk TARGETS=farewell farewell
```

we get:

```
farewell, world
```

This time **nmake** got `farewell` when it expanded `$(TARGETS)`, so now when we look through the assertion symbol table we find,



Assertion Symbol Table

but there's no trace of `hello` and `goodbye`. Change the command line assignment operator to `+=` and we can build `goodbye`, `hello`, and `farewell`.

2.8.6 In Prerequisite Lists

Variable references in a prerequisite list are expanded when **nmake** builds its internal representation of the assertion, just like targets. Here's an example. It's not unusual to keep track of source files using a variable

```
SOURCE = a.c b.c
```

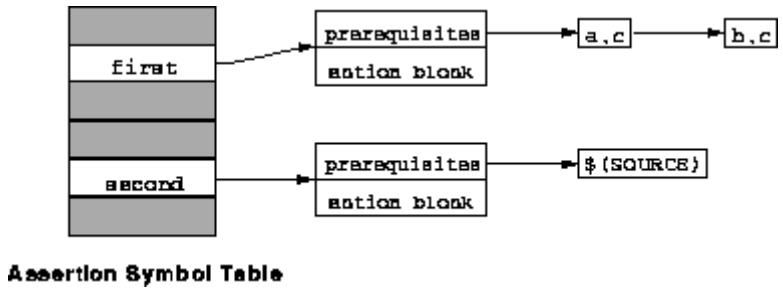
and reference that variable in an assertion:

```
first : $(SOURCE)
```

When **nmake** reads the assertion it expands `$(SOURCE)` and ends up building an internal representation of an assertion that has `first` as the target, and `a.c` and `b.c` as prerequisites. Adding a dollar sign

```
second : $$(SOURCE)
```

delays the expansion, so we would find the following definitions



Assertion Symbol Table

in the assertion symbol table after **nmake** finished reading the two assertions. They're clearly different, but they're also equivalent (as long as `SOURCE` isn't changed), because variable references left in the prerequisite list are expanded when **nmake** builds the target.

2.9 A Look Back

Real makefiles handle tough jobs and they can get complicated, so don't be misled by our examples. But we also want to make sure you don't underestimate what you've learned so far. We kept things simple, and that let us introduce fundamental concepts without getting lost in the details of the example.

FOOTNOTES:

[1]
Running **nmake** using a full pathname may work, but it's not something we recommend.

[2]
make users should appreciate the remarkable flexibility.

[3]
Use something like,

```
silent ksh -c '...'
```

`silent` to complicated commands, like `if` or `case` statements, that need to be interpreted by the shell.

[4]
Can you figure out how to change `mistake.mk` and get three names in the error list?

[5]
See chapter 3 of the *nmake User's Guide* and chapter 4 of the *nmake Reference Manual* for more about variables.

[6]
Of course we're exaggerating, but the limits are so big we don't think you'll ever notice.

[7]
See chapter 7 of the *nmake User's Guide* for more details.

---

3. Extra Files

nmake does some things behind your back that you should know about. You won't need all the details, but a little knowledge is important. For example, realizing the `.mo`, `.ms`, and `.ml` suffixes have been claimed could prevent accidents; knowing something about the extra files **nmake** creates will help you understand important aspects of **nmake**'s behavior.

3.1 The Objectfile

nmake builds an *objectfile* [8] from a makefile and uses its internal machinery to keep track of both files. If the objectfile is up to date **nmake** reads it instead of the makefile; otherwise **nmake** reads the makefile and builds a new objectfile that can be used next time. As far as **nmake** is concerned, the two files are supposed to be equivalent, but reading the objectfile saves time.

Objectfiles are associated with makefiles using a simple technique: **nmake** picks the name by removing the makefile's suffix, if there is one, and attaching `.mo`. For example, `hello.mo` is the objectfile associated with `hello.mk`. Even though there's some ambiguity [9] **nmake** isn't easily confused, because the name of the associated makefile is also recorded in the objectfile. If you're really curious, look at an objectfile with `cat -v` or `od -c` and see if you can find the name.

An objectfile is also sometimes called a "compiled makefile," but the terminology is a bit misleading. It suggests equivalence and implies you get exactly the same results no matter whether **nmake** reads the makefile or the objectfile. In reality there are differences - we'll give you an easy example.

3.1.1 A Small Surprise

We mentioned `print` earlier in the paper when we talked about the special words that sometimes need quoting. It's a simple command [10] that tells **nmake** to copy the rest of the line to standard output. If we put

```
print this message comes from print

hello :
   silent echo "hello, world"
```

in a file named `print.mk` and type

```
nmake -f print.mk
```

we get:

```
this message comes from print
hello, world
```

No surprises yet, but run **nmake** again and the message from `print` disappears. It's easy to show this is due to the objectfile - remove `print.mo` or tell **nmake** to ignore the objectfile using the `-o reread` option

```
nmake -f print.mk -o reread hello
```

and the message from `print` reappears.

The explanation is important, so pay attention. It may look like a mistake, but it's not because the objectfile isn't a translation of the makefile. Instead, most of what you'll find in an objectfile is a dump of several internal symbol tables in a portable format that **nmake** can digest quickly. **nmake** ends up with the same data in those symbol tables no matter which file (makefile or objectfile) it reads. But `print` doesn't create any data, so there's no record of it in the objectfile, and that's why the message disappeared.

By the way, there is an easy way to make sure `print` runs whenever **nmake** does, even if the objectfile is used. We'll show you how later in the paper when we talk about `.INIT`.

3.2 The Statefile

nmake needs a *statefile* to preserve information, like compiler flags, that may not be available the next time it runs, but **nmake** goes much farther and remembers lots of important information (e.g., file time stamps and when targets were built) in the statefile. **nmake** reads the statefile when it starts, updates it right before quitting,

and builds targets when information it needs is missing from the statefile or conflicts with what's available from another source, like the file system.

Statefiles are associated with makefiles using the technique that we just described for objectfiles, except **nmake** attaches `.ms` instead of `.mo` as the identifying suffix. For example, `hello.ms` would be the statefile associated with `hello.mk`.

3.2.1 A State Variable

Put parentheses around a variable name, add it to a prerequisite list, and you've made the target depend on the variable. Here's a simple example that we'll save in a file named `state.mk`; it should help, as long as you close your eyes when you come to `.VIRTUAL` in the prerequisite list - we'll explain that one in a few sections:

```
AUDIENCE = world

goodbye : .VIRTUAL (AUDIENCE)
   silent echo "$(<), $(AUDIENCE)"

hello : .VIRTUAL
   silent echo "$(<), $(AUDIENCE)"
```

We started with `message.mk`, split the assertion up, added `.VIRTUAL` to both prerequisite lists, and then turned `AUDIENCE` into a *state variable* by saying `goodbye` depends on `(AUDIENCE)`. State variables are saved in the statefile and targets, like `goodbye`, that depend on a state variable are rebuilt when that variable changes.

It's easy to demonstrate the behavior, but first let's see what happens when we build `hello`, because it doesn't have `AUDIENCE` as a prerequisite. We get the usual message when we type,

```
nmake -f state.mk hello
```

but try it again and nothing happens, and that's why we added `.VIRTUAL` to the prerequisite list. Change `AUDIENCE` on the command line and the silence continues. But try the same thing with `goodbye` and we get the expected message from

```
nmake -f state.mk goodbye
```

and another one when we type:

```
nmake -f state.mk AUDIENCE='New Jersey' goodbye
```

nmake rebuilds `goodbye` whenever we change `AUDIENCE`, and the state variable prerequisite is the reason why.

3.3 The Lockfile

There's one more special file, but this one is harder to find. **nmake** creates a *lockfile* [11] when it starts and removes it right before it quits, so we have to look for the lockfile while **nmake** is running. If we put

```
lock :
   ls -l lock.ml
```

in a file named `lock.mk` and type

```
nmake -f lock.mk lock
```

we get:

```
+ ls -l lock.ml
----------   1 drexler  sablime        0 Dec 28 15:06 lock.ml
```

That's the lockfile - it's empty, has no permissions, ends in `.ml`, and is associated with a makefile, just like the statefile and objectfile. The lockfile is there to protect the statefile from multiple updates, but it's not much of an obstacle. You get a message something like

```
make: warning: another make has been running on lock.mk in . for the past 2.00s
make: use -K to override
```

when **nmake** finds a lockfile that it didn't create. Follow **nmake**'s advice and use the `-K` option or just remove the lockfile and **nmake** will run without complaining [12] but first be sure **nmake** really isn't using the makefile.

3.3.1 MAKEFILE

Our last example works because the makefile is named `lock.mk`, but renaming the makefile breaks the example. **nmake** remembers the name of your makefile [13] in a variable named `MAKEFILE`, so

```
lock :
   ls -l `echo $(MAKEFILE) | sed 's/. [^.]*$//' | sed 's/$/.ml/'`
```

works no matter where we put the makefile. Incidentally, we used `sed` instead of `basename`, because nothing says we can't use a non-standard suffix, like `.MK`, in a makefile name. The `sed` stuff is ugly, so here's how you would do the same thing if you had more experience with **nmake** and knew about edit operators:

```
lock :
   ls -l $(MAKEFILE:B:S=.ml)
```

We're not going to explain edit operators in this paper, but we think the example shows their advantage. Edit operators are efficient and often provide elegant solutions.

FOOTNOTES:

[8] Existing documentation says object file, rather than objectfile.

[9] For example, makefiles named `hello.mk`, `hello.MK`, and `hello` all map to the same objectfile.

[10] The **nmake** programming language is described in chapter 5 of the **nmake** manual.

[11] Existing documentation says lock file, rather than lockfile.

[12] Telling **nmake** to skip the statefile update using the `-o nowritestate` option also works.

[13] Actually just the first makefile from your command line.

---

4. Comments

Makefile comments begin with `/*`, end with `*/`, and don't nest. The C-style comments go anywhere, even in action blocks that are shell scripts. You should want proof, because `*` means something special to the shell, so it's not hard to imagine problems. If you're not sure what we're talking about, see what the shell does when you type:

```
echo /* disappears */ "hello, /* shouldn't disappear */ world"
```

The opening `/*` matches everything in the root directory on your system, and the closing `*/` matches some of your own directories. We could have done some damage if `echo` had actually been `rm -r`. Now let's try the same command, but in a makefile. Put

```
hello :
   silent echo /* disappears */ "hello, /* shouldn't disappear */ world"
```

in `comment.mk` and type

```
nmake -f comment.mk
```

and we get:

```
hello, /* shouldn't disappear */ world
```

The comment that was supposed to disappear really did, and the stuff that looked like a comment didn't fool **nmake** because it noticed the quoted string. Obviously, handling C-style comments in action blocks isn't trivial. They need to be removed, but finding comments means **nmake** has to look through action blocks the way the shell would, because throwing too much away is no good either.

4.1 Some Comments About Comments

Single line shell-style comments introduced by `#` are also allowed, but sometimes what happens can be surprising: use `#` too early in a makefile and **nmake** may decide to run `/bin/make`; follow the `#` by words, like `include` or `define`, that mean something to `cpp`, and **nmake** may decide to preprocess your makefile [14] If you don't like surprises stick to the C-style comments.

Makefiles are programs and they often deserve some documentation. Comments are convenient and inexpensive, so don't leave them out just because you're concerned about efficiency. In fact, most of the time **nmake** won't even notice comments, because it usually reads objectfiles.

FOOTNOTES:

[14]
Preprocessing makefiles is not recommended and may not be supported in future releases.

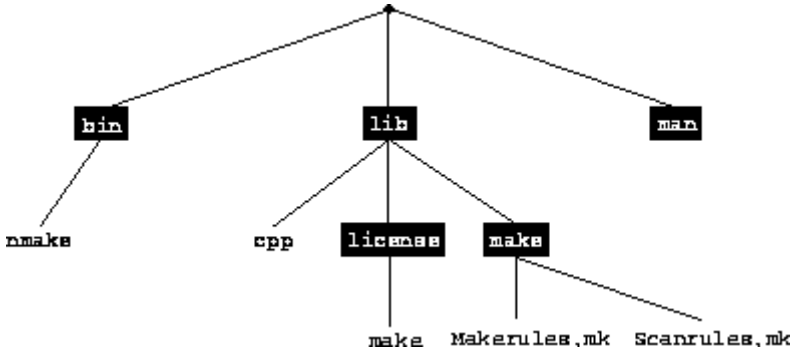
---

5. The Package

The **nmake** command is part of a large package that can be installed anywhere on your system. If **nmake** is in your `PATH` and you're running `ksh` then

```
dirname `dirname \`whence nmake\``
```

will show you how to get to the root of the package. A few of the components you'll find there are shown in the following picture:



The picture is simplified - we only included **nmake**, the man page directory, and the four files we'll talk about briefly in this section.

5.1 cpp

It's a bit surprising at first, but the **nmake** package comes with its own C preprocessor. It's required because **nmake** needs precise control over where `cpp` finds include files, but the preprocessor your compiler uses may have a mind of its own. For example, there are situations where the native `cpp` could include the wrong file (i.e., not the one **nmake** wanted) if it finds something like,

```
#include "b.h"
```

while it's reading `/tmp/include/a.h` and always insists on first looking for `b.h` in the directory of the including file: in this case it would first try file `/tmp/include/b.h`. **nmake**'s `cpp` recognizes special options, like `-I-`, that gives **nmake** the control it needs over include files.

5.2 The License File

Walk down the tree to `lib/license/make` and you've found your license file. You get a license when you buy **nmake** or renew an existing contract. **nmake** checks the license file every time it runs and complains if it's missing. When your license expires all **nmake** will do for you is print an error message that points at the invalid license; you will have to renew your license and install the new copy here.

5.3 Makerules.mk

The file `lib/make/Makerules.mk` is called the *base rules* file. It's used [15] every time **nmake** runs and it controls much of **nmake**'s behavior. Even though `Makerules.mk` is just another makefile, it is over 2000 lines of difficult reading, and you have to be fluent in **nmake** before you can make much of a dent in it.

Still, it's important to know about `Makerules.mk`. It can be a convenient source of examples, even if most of the file is a complete mystery. As you get better you'll find it can be a big help when you're debugging problems or when you want to change **nmake**'s behavior slightly. Eventually, you may even be able to go directly to `Makerules.mk` and answer many of your own questions.

But perhaps the most important reason for mentioning `Makerules.mk` is that it helps put the existing documentation in the proper perspective. Even though everything's presented on pretty much an equal footing, much of what's discussed in the manual is needed in `Makerules.mk` and nowhere else. Keep that in mind and you'll be a more discriminating reader when you do go back to the manual.

5.4 Scanrules.mk

We mentioned **nmake**'s built-in dependency scanner when we talked about implicit prerequisites early in the paper. The scanner is programmable and the default behavior is controlled by the rules in file

`lib/make/Scanrules.mk`. We're not going to teach you how to program the scanner. If you're interested you'll find the details described in Appendix G of the **nmake** manual, and if you ever need custom scan rules use `Scanrules.mk` as a good source of examples.

FOOTNOTES:

[15]
Actually **nmake** reads a file named `makerules.mo` that's the objectfile built from `Makerules.mk`, but with the name changed to prevent accidents.

---

6. Special Atoms

When we stopped working on `message.mk` in section 2.7.6 we told you something was wrong. It's time to show you the problem and the fix, and in the process introduce some magic tokens, called *special atoms*, that give you extra control over **nmake**.

There are about a hundred special atoms, but we'll only need a few, and we're not going to classify them. Special atoms have the same look - a period followed by some upper case letters. A few of the more complicated ones include several periods and occasionally a percent sign as a pattern-matching character. The names are wired into **nmake**. Special atoms won't often collide with Unix file names, so it's not hard to understand **nmake**'s choice of names.

6.1 What's Wrong?

Flip through the manual and you may decide that targets, like `hello` and `goodbye` in `message.mk`, are *labels*. The official definition of a label, taken from manual's glossary, looks something like:

> If an assertion has no prerequisites but has an update action block, or if an assertion has prerequisites but no actions and the target is not a file, the target represents a label in the makefile. The time stamps of the labels are stored in the statefiles but are ignored by **nmake**; therefore, they are made on subsequent invocations of **nmake**.

If you don't read carefully you might assume that targets that don't have prerequisites are automatically labels and are always built. It's easy to overlook, "and the target is not a file," but the words explain why `message.mk`, and most of our other makefiles, aren't quite right. We can show you the problem by creating a file that has the same name as one of our targets

```
touch hello
```

and typing:

```
nmake -f message.mk hello
```

We get the usual message the first time, but try it again and nothing happens. This time it's easy to blame the statefile - remove `message.ms` or tell **nmake** to ignore the statefile using the `-S` option

```
nmake -f message.mk -S hello
```

and the message reappears.

The explanation isn't difficult, but you'll probably need some help. **nmake** looked around after it finished building `hello`, found a file named `hello`, and decided the target was responsible for the file. Information about

the connection was saved in the statefile, and after that **nmake** refused to build `hello`, because everything available from the statefile and file system indicated it was up to date.

6.2 .FORCE

So how do we make sure a target is always built, even if it appears to be up to date? Adding a special atom named `.FORCE` to the prerequisite list does it:

```
AUDIENCE = world

goodbye hello : .FORCE
   silent echo "$(<), $(AUDIENCE)"
```

When **nmake** finds `.FORCE` it marks the targets, in this case `hello` and `goodbye`, with a flag that says they're supposed to be built whenever we ask. Save the example in a file named `force.mk` and now something happens whenever we type,

```
nmake -f force.mk hello
```

and it doesn't matter if the dummy `hello` file is there or not.

6.3 .VIRTUAL

Something's still wrong with `force.mk`, and we'll show you the problem using `clobber`. We'll talk about more `clobber` later in section 11.2 when we talk about the common actions you get for free with **nmake**. It's what you do when you're really finished and want to clean everything up. Even though `force.mk` doesn't leave much lying around, when we type

```
touch hello
nmake -f force.mk clobber
```

we get:

```
+ ignore rm -f hello force.mo force.ms
```

Look carefully at what happened - you should be concerned. Deleting the objectfile and statefile is normal, but **nmake** also removed `hello`, because it believed the file was built by the target named `hello`. So how do we protect innocent files, like `hello`, that just happen to have the same name as a makefile target? Start with `force.mk`, add `.VIRTUAL` to the prerequisite list,

```
AUDIENCE = world

goodbye hello : .VIRTUAL .FORCE
   silent echo "$(<), $(AUDIENCE)"
```

and save the result in `virtual.mk`. Now when we type

```
touch hello
nmake -f virtual.mk clobber
```

we get

```
+ ignore rm -f virtual.mo
```

so `hello` was spared. The `.VIRTUAL` special atom breaks the connection between targets and files. Adding it to the prerequisite list means **nmake** won't associate `hello` and `goodbye` with files, and that's what protected the file when we ran `clobber`. By the way, `.FORCE` is still needed. Take it out of the prerequisite list in `virtual.mk` and

you only get one message, because **nmake** uses the statefile to remember when targets marked by `.VIRTUAL` are built.

---

7. Programming nmake

nmake recognizes some standard programming constructs that are described in chapter 5 of the manual. Programmers usually pick the important points up quickly, so we won't spend time showing you how to use `if` statements or write `for` loops. Instead, we'll say a few words about `print`, introduce two other important commands, and then show you how to write action blocks designed for **nmake** rather than the shell.

7.1 `print`

We've already used `print` in an example, and it wasn't difficult. When **nmake** finds a `print` statement it takes the rest of the line, expands variable references, and copies the result to standard output. The syntax is:

```
print [ options ] message ...
```

Options were introduced in 3.1's implementation of `print`. We're not going to describe them here, except to say that `--` (i.e., two minus signs) marks the end of the options. In other words, if you're using a new version of **nmake**

```
print --  message
```

prints your message, even if it starts with a minus sign.

7.2 `error`

When **nmake** finds an `error` statement it takes the rest of the line, expands variable references, and usually copies the result to standard error. The syntax is:

```
error [ level ] message ...
```

`level` is an optional integer [16] that controls the format of the message and determines what happens after **nmake** prints the message. Setting `level` to 0 is the same as leaving it out, as long as your message doesn't begin with a number. A `level` greater than 2 means abort after printing the message. We can experiment with different error `level`s by putting

```
error $(LEVEL) this message comes from error

hello :
   silent echo "hello, world"
```

in a file named `error.mk`. For example, if we type

```
nmake -f error.mk LEVEL=3
```

we get the message from `error`

```
make: this message comes from error
```

and nothing else, because a `level` of 3 (or higher) means abort. Increase `level` and nothing obvious changes, but check **nmake**'s exit status after it aborts and you'll see it depends on `level`.

7.3 `include`

Cooperating makefiles usually need to share information, and that's why a mechanism for including files is so important. When **nmake** finds an `include` statement it takes the rest of the line, expands variable references, and reads the files named in the result. The syntax is:

```
include [ - ] file ...
```

The optional minus sign disables the warning about a file that can't be found. Individual files are space separated and optionally quoted, so you'll often see

```
include "file"
```

even though the quotes are unnecessary. **nmake** complains when it can't find an include file, but it's just a warning that usually disappears when **nmake** reads the objectfile. For example, put

```
include global.mk

hello :
   silent echo "$(<), $(AUDIENCE)"
```

in `include.mk` and type

```
nmake -f include.mk hello
```

and we get:

```
make: "include.mk", line 1: global.mk: cannot read include file
hello,
```

nmake couldn't find `global.mk` and told us so, but run it again and the warning goes away. The message from `hello` is still incomplete, because `AUDIENCE` isn't defined, but put

```
AUDIENCE = world
```

in a file named `global.mk` and both problems are fixed.

7.4 .MAKE

Action blocks are usually handed to the shell, but adding `.MAKE` to a prerequisite list means reading the action block (just like it's another makefile) whenever it builds the target. It's not hard to figure out who should get an action block. You let the shell do the work when you want to run Unix commands (e.g., a compiler) or do things in the file system, and you let **nmake** handle the action block when you want to define variables or assertions that can be used later on.

7.4.1 Sometimes You Have A Choice ...

Most of our examples have been so simple that we could easily rewrite the action blocks and hand them to **nmake**. For example,

```
AUDIENCE = world

goodbye hello : .MAKE
   print $(<), $(AUDIENCE)
```

and `message.mk` seem to be equivalent, even though the shell handles one action block and **nmake** handles the other. But even these two simple makefiles can behave differently. Add things to `AUDIENCE` that mean something special to the shell, `print`, or `echo` and you may notice a difference. For example, save the last makefile in `make.mk` and we get different results from

```
nmake -f message.mk AUDIENCE='$LOGNAME'
```

and

```
nmake -f make.mk AUDIENCE='$LOGNAME'
```

because the shell understands `$LOGNAME`, but **nmake** doesn't.

We haven't talked much about environment variables, but put parentheses around `LOGNAME` and we get the same thing from both makefiles. **nmake** reads your environment when it starts, so it recognizes `LOGNAME`, and that's why we get the right answer when **nmake** expands `$(LOGNAME)`. Collisions aren't a problem, because variables defined on the command line or in a makefile override environment variables.

7.4.2 ... But Usually You Won't

Even though the shell is usually the right choice, there may be times when we need to exercise some control over **nmake** using an action block. Here's an example that we'll save in `setup.mk` - it's contrived, but it's also easy and will help you understand what happens when **nmake** handles an action block:

```
setup : .MAKE
   AUDIENCE = world

   goodbye hello :
      silent echo "$(<), $(AUDIENCE)"
```

We started with a target named `setup`, added `.MAKE` as a prerequisite, and copied `message.mk` into `setup`'s action block. You deserve an explanation if you're wondering why we didn't use

```
setup : .MAKE
   include message.mk
```

as our example: we wanted to make you look at a variable definition, a blank line, and an assertion in an action block. Even though we can see the assertion that defines `hello`, it's hidden in an action block, so when we type

```
nmake -f setup.mk hello
```

we get,

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

because **nmake** won't find it, but add `setup` to the command line

```
nmake -f setup.mk setup hello
```

and we get:

```
hello, world
```

Building `setup` defined `AUDIENCE`, `hello`, and `goodbye`, and that allowed **nmake** to build `hello`. By the way, you should have no trouble understanding what happens when we type:

```
nmake -f setup.mk AUDIENCE='New Jersey' setup hello
```

The command line assignment defined `AUDIENCE`, but building `setup` replaced the definition, and that's why the message doesn't change.

FOOTNOTES:

[16] Negative numbers work with **nmake**'s debugging option, but won't be described here.

---

8. Promises

We made some promises early in the paper that we haven't kept yet and now is a good time to deliver. The explanations depend on the `.INIT`, `.ARGS`, and `.MAIN` special atoms that we'll introduce in this section.

8.1 `.INIT`

There's a target named `.INIT` that's automatically built right before anything else in your makefile. It can be convenient when you're debugging and is important if you want to do something, like execute a shell script, every time **nmake** process runs.

We've already seen an example where `.INIT` would help. Remember the disappearing `print` message that we blamed on the objectfile? If we go back to `print.mk`, make a target named `.INIT`, mark it with the `.MAKE` special atom, and move the `print` command into `.INIT`'s action block

```
.INIT : .MAKE
   print this message comes from print

hello :
   silent echo "hello, world"
```

then we get the message from `print` whenever we run **nmake**. The explanation is easy: the symbol tables written to the objectfile include our definition of `.INIT`, so the `print` command isn't lost because this time it's part of `.INIT`'s action block.

8.2 `.ARGS`

nmake collects the targets from your command line, adds them to the prerequisite list of a special atom named `.ARGS`, and eventually goes back and builds everything it finds in `.ARGS`'s prerequisite list. We'll need `.ARGS` to explain why we only get one message from,

```
nmake -f message.mk hello hello
```

but first we have to show you how to look at a prerequisite list.

8.2.1 Another Automatic Variable

There's an automatic variable named `$(~)` that stands for all the explicit prerequisites [17] of the current target. We happen to be interested in the prerequisites of `.ARGS`, so `$(~)` isn't quite what we want, but there's an easy way to use automatic variables to look at any target. Follow the character that identifies the automatic variable by the name of a target, and **nmake** gives you information about that target. For example, `$(~.ARGS)` is how we ask for the explicit prerequisites of `.ARGS`, so if we put

```
.INIT : .MAKE
   print .ARGS : $(~.ARGS)

goodbye hello :
 silent echo "$(<), world"
```

in `args.mk` and type

```
nmake -f args.mk hello goodbye
```

we get:

```
.ARGS : hello goodbye
hello, world
goodbye, world
```

The first line is from the `print` statement that we put in `.INIT`'s action block, and the two prerequisites are the targets we named on the command line.

8.2.2 Disappearing Prerequisites

Mention the same target twice on the command line and we know something unexpected happens when we use `message.mk`. Try it with `args.mk`

```
nmake -f args.mk hello hello
```

and the same thing happens,

```
.ARGS : hello
hello, world
```

but now we can understand why. There's only one `hello` in `.ARGS`'s prerequisite list and that's why we only get one message. But that's usually how prerequisite lists work: no matter how many times we say a target depends on a prerequisite, we'll only see it once when we look through the list.

8.3 `.MAIN`

nmake usually builds the first target in your makefile that doesn't look like a special atom [18] when there aren't any command line arguments, but there's a target named `.MAIN` that lets you pick your own defaults. For example, if we put

```
AUDIENCE = world

.MAIN : hello

goodbye hello :
   silent echo "$(<), $(AUDIENCE)"
```

in `main.mk` then all we have to do is type

```
nmake -f main.mk
```

when we want to say `hello`, because the prerequisites of `.MAIN` end up as the default targets.

You have to be careful where you put `.MAIN`. Move it to the end of `main.mk` and see what **nmake** builds when you don't tell it what to do. The explanation is easy, so we'll leave it to you. Hint: `$(~.MAIN)` and the technique we just used in `args.mk` will show you the prerequisites.

FOOTNOTES:

[17] Explicit prerequisites are the ones you see in assertions. They don't include the implicit prerequisites **nmake** finds by scanning source files.

[18]

Actually, **nmake** picks the first target that it finds while reading your makefile that begins with a letter, digit, or underscore, or includes a `/` or `$` in the name, that's not also marked by the `.SPECIAL` or `.OPERATOR` special atoms.

---

9. A Program - Finally

Our next few examples will finally build a real program. We need somewhere to work, so let's start with:

```
mkdir /tmp/n2
cd /tmp/n2
```

We picked `/tmp/n2`, but you can use any directory you want, as long it's empty and you remember to substitute your directory when we use `/tmp/n2`.

9.1 Three Source Files

We'll need a few source files that build a program, but we won't care what the program does, as long as it compiles. A `main` program that calls routines in two other files will be good enough, so we'll put

```
main() {
   a();
   b();
}
```

in `main.c`,

```
#include <stdio.h>

a() {
   puts("In routine a");
}
```

in `a.c`, and

```
#include <stdio.h>

b() {
   puts("In routine b");
}
```

in `b.c`.

9.2 And A Makefile

Building our program by hand is simple. Just type

```
cc -o prog main.c a.c b.c
```

and the compiler handles everything, but it's not efficient, even for our trivial program. The compiler does exactly what we ask, so all three files are compiled and linked every time we run the command. **nmake** could help, because it has a good memory; it's even easy to go directly from the command line to a makefile. Build an assertion that has `prog` as the target, the three source files as prerequisites, and our command line as the action block

```
prog : main.c a.c b.c
   cc -o prog main.c a.c b.c
```

and we can let **nmake** keep everything up to date. It's a start and it even works, but there's a bit more to do before we use the makefile.

9.2.1 Automatic Variables

The first line of the assertion in our makefile is almost repeated in the action block. We already know about `$(<)` and `$(~)`, so

```
prog : main.c a.c b.c
   cc -o $(<) $(~)
```

looks like a good way to clean things up. But don't jump the gun because `$(~)` in our action block stands for all the explicit prerequisites of `prog`, so we could get more than we want. We'll show you the problem and an easy fix after we make one more change.

9.2.2 CC

There's a variable named `CC` that's defined to be `cc` in `Makerules.mk`, and it's customary to use `$(CC)` to talk about a C or C++ compiler. We get `CC` for free, so we can replace `cc` in our action block

```
prog : main.c a.c b.c
   $(CC) -o $(<) $(~)
```

and not notice a difference. `CC` is a variable, so changing compilers is easy. We can redefine `CC` in our makefile if we want a different default, or change it on the command line when we want to try another compiler. For example, if you're using C++ your makefile probably should look something like:

```
CC = CC

prog : main.c a.c b.c
   $(CC) -o $(<) $(~)
```

9.2.3 A State Variable

Our action block uses `$(CC)` as the compiler, so it's natural to want to have **nmake** rebuild `prog` when `CC` changes. We already know about state variables, but there's a problem when we add `(CC)` to `prog`'s prerequisite list. Save

```
prog : main.c a.c b.c (CC)
   $(CC) -o $(<) $(~)
```

in `broken.mk` and type

```
nmake -f broken.mk
```

and we get some strange complaints from the shell:

```
ksh[52]: syntax error at line 2 : `(' unexpected
ksh[52]: Cset:  not found
```

You may see something different on your system, but that shouldn't matter - figuring out what's wrong from the shell's error messages isn't easy. Fortunately, there's a way to find out what **nmake** handed to the shell. The `-n` option means print shell actions rather than execute them, so when we type

```
nmake -f broken.mk -n
```

we get,

```
+ cc -o prog main.c a.c b.c (CC)
```

and now it's easy to see what happened. **nmake** picked up the three source files and `(CC)` when it expanded `$(~)`, and the shell didn't like the parentheses around `CC`. But there's more wrong than just parentheses: `(CC)` isn't a file, so the compiler would complain even if the shell didn't.

9.2.4 A New Automatic Variable

We get too much with `$(~)`, but there's an automatic variable named `$(*)` that stands for all the explicit *file* prerequisites of the current target. `(CC)` is a prerequisite, but it's not a file so it shouldn't show up when `$(*)` is expanded. Let's replace the `$(~)` in `broken.mk` by `$(*)`

```
prog : main.c a.c b.c (CC)
   $(CC) -o $(<) $(*)
```

and save the result in `fixed.mk`.

9.2.5 Let's Build The Program

We can finally build our program by typing:

```
nmake -f fixed.mk prog
```

What you see will depend on your compiler. We get

```
+ cc -o prog main.c a.c b.c
main.c:
a.c:
b.c:
Linking:
```

and we end up with an executable program named `prog`. It's easy to prove **nmake** has already helped. Try to build the program again and nothing happens, and that's exactly as it should be, because `prog` is up to date.

9.3 A Better Makefile

Our makefile works, but it could be better. We'll show you two problems and then spend the rest of this section trying to improve things. It's going to take some time and we'll even go backwards at first, but we'll eventually end up with a simple solution that you should understand and appreciate.

9.3.1 What's Wrong

The assertion in `fixed.mk` says `prog` depends on `main.c`, `a.c`, and `b.c`, so **nmake** should rebuild `prog` when a source file changes. Let's see what happens - touch one of the source files and run **nmake** again

```
touch a.c
nmake -f fixed.mk prog
```

and we get:

```
+ cc -o prog main.c a.c b.c
main.c:
a.c:
b.c:
Linking:
```

nmake built a new `prog`, but we did more work than was necessary. Any prerequisite that are newer than the target triggers the build [19] but `$(*)` hands the three source files to the compiler, so they're all recompiled when **nmake** executes our action block. There's another reason why we're not happy with `fixed.mk`. Try to clean up using `clobber`

```
nmake -f fixed.mk clobber
```

and we get:

```
+ ignore rm -f fixed.mo fixed.ms prog
```

nmake removed everything it knew about, but the compiler created `main.o`, `a.o`, and `b.o` when **nmake** ran our action block, and they weren't removed.

9.3.2 Object Files

We can address both complaints using object files, but our first few tries won't be simple. We'll start with an assertion that says `prog` depends on `main.o`, `a.o`, and `b.o`,

```
prog : main.o a.o b.o
   $(CC) -o $(<) $(*)
```

add three assertions that tell **nmake** how to build the object files,

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

a.o : a.c (CC)
   $(CC) -c $(*)

b.o : b.c (CC)
   $(CC) -c $(*)
```

and save everything in a file named `object.mk`. If we build `prog` using `object.mk` we get,

```
+ cc -c main.c
+ cc -c a.c
+ cc -c b.c
+ cc -o prog main.o a.o b.o
```

but now when we touch `a.c` and rebuild we get:

```
+ cc -c a.c
+ cc -o prog main.o a.o b.o
```

The only source file that was recompiled is the one we touched, and that's exactly the behavior we want. There's also good news when it comes to `clobber`, because **nmake** now knows about `main.o`, `a.o`, and `b.o`, but we'll leave the proof to you.

9.3.3 CCFLAGS

`object.mk` passed our first few tests, but it's still not right. We mentioned **nmake**'s `cpp` earlier in the paper, but so far we're not using it. Compilers often let you plug in your own preprocessor using options or environment variables, and **nmake** does a good job figuring out how. The details are interesting, but complicated. If you ever run **nmake** and see something like

```
probing C language processor /bin/cc for make information
probing C language processor /bin/cc for pp information
```

then you're watching **nmake** figure important things out about your compiler, including how to select an alternate preprocessor.

The results we're interested in (and much more) are hidden in a special variable named `CCFLAGS`. We haven't talked about the auxiliary value that can be associated with any variable using the `&=` assignment operator, but it's a good guess it was implemented to accommodate sophisticated variables like `CCFLAGS`. **nmake** can hide important stuff, like the instructions needed to use **nmake**'s preprocessor, in the auxiliary value associated with `CCFLAGS` and be sure it's not trashed when we do things like:

```
CCFLAGS = -g
```

There's much more to `CCFLAGS`, but most of it is well beyond the scope of this paper. Fortunately, we can use `CCFLAGS` even if we don't understand all the details. If we start with `object.mk`, reference `CCFLAGS` in the assertions that build our object files, add `(CCFLAGS)` to the three prerequisite lists,

```
prog : main.o a.o b.o
   $(CC) -o $(<) $(*)

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

a.o : a.c (CC) (CCFLAGS)
   $(CC) $(CCFLAGS) -c $(*)

b.o : b.c (CC) (CCFLAGS)
   $(CC) $(CCFLAGS) -c $(*)
```

save our work in `ccflags.mk`, and type

```
nmake -f ccflags.mk prog
```

we get something like:

```
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c main.c
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c a.c
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c b.c
+ cc -o prog main.o a.o b.o
```

Most of the options come from expanding `$(CCFLAGS)`, and all but `-O` are stored in the auxiliary value. The `-Q`*path* option points our compiler at **nmake**'s `cpp`. The file following `-I-D` is called the probe file: it's an initialization file that helps tune **nmake**'s `cpp` to the compiler we're using. `CCFLAGS` is also responsible for the `-I-` option; additional `-I` options are strategically placed around `-I-` when the source file that's being compiled includes header files from non-standard directories.

9.3.4 Metarules

We've done more than necessary in `ccflags.mk`. Remove the three assertions that build the object files, save what's left

```
prog : main.o a.o b.o
   $(CC) -o $(<) $(*)
```

in `meta.mk`, and type

```
nmake -f meta.mk prog
```

and everything works exactly as it did when we used `ccflags.mk`! We're down to a two-line makefile, but how? The assertion says `prog` depends on `main.o`, `a.o`, and `b.o`, but there's nothing left in `meta.mk` that explains how the

object files are built.

It turns out **nmake** can guess how to build certain files using special instructions called metarules. The metarule we're relying on is defined in `Makerules.mk`; it looks like

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

and tells **nmake** how to build a `.o` file from a `.c` file. It's not hard to understand what happened. **nmake** found `main.o` in the prerequisite list, but we deleted the instructions it needed, so **nmake** looked around and found a file named `main.c` and a metarule that it could use to build `main.o` from `main.c` and was perfectly happy. Mentally replace the two `%` characters in the metarule by `main`, `a`, and `b` and you'll see how **nmake** resurrected the three assertions that we deleted from `ccflags.mk` [20].

9.3.5 Not Quite

`meta.mk` is simple, but it's still not right. The action block builds `prog` by linking the three object files, but we could miss some flags by completely ignoring `CCFLAGS`, and there are other variables (e.g., `LDFLAGS`) that contain important information for the link editor and should also be used in the action block.

The `clean` common action also misbehaves. It should remove intermediate files like `main.o`, `a.o`, and `b.o`, but it's not supposed to touch programs. However, when we type

```
nmake -f meta.mk clean
```

we get,

```
+ ignore rm -f b.o a.o main.o prog
```

so `prog` is also removed. There are easy solutions - we'll show you the one that uses the `::` assertion operator.

9.3.6 Assertion Operators

Assertion operators have the same look - a colon, an optional letter or underscore followed by zero or more letters, digits, or underscores, and a closing colon. There are about twenty assertion operators defined in `Makerules.mk` and most are also documented in the manual. Find `Makerules.mk` and you can list the ones you get for free using a command something like:

```
grep '. OPERATOR' Makerules.mk
```

The assertion operator we're interested in is `::`, which is sometimes called the main source dependency operator. It's easy to use - put `prog` on the left side of `::` and our three source files on the right side,

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

and we're done. It's a perfectly general one-line solution, and is the way we recommend you build programs. Save the makefile in `prog.mk`, because we'll need it again when we talk about viewpathing in the next section.

One way to understand `prog.mk` is to pretend **nmake** calls a function with three arguments whenever it finds an assertion operator. Everything to the left of the assertion operator is one argument, everything to the right is another, and the action block, if there is one, is the last argument. The assertion operator itself (i.e., the body of our function) is written in **nmake**'s programming language [21]. The fact that we used an assertion operator in our makefile isn't recorded in the objectfile, so assertion operators work by creating variables and assertions that end up in the objectfile.

That's exactly what happened when we used `::` in `prog.mk`. If you can imagine using a `for` loop to march through the three source files (i.e., the second argument) and an edit operator to replace the `.c` suffix by `.o`, then you can understand how the `::` operator was able to build an improved version of the assertion we used in `meta.mk`. The changes that `::` threw in to fix the problems we mentioned in the last section are simple and could easily be included in `meta.mk`, but they would have to be duplicated in other makefiles. Having everything isolated in the `::` operator in `Makerules.mk` is an enormous advantage.

FOOTNOTES:

[19]
Touching `prog` also triggers a build, because the statefile and the file system disagree about `prog`'s time stamp.

[20]
`$(>)` instead of `$(*)` is the only difference, and in this example it's insignificant. You can read about `$(>)` in chapter 4 of the *nmake Reference Manual*.

[21]
The three arguments are referenced using `$(<)`, `$(>)`, and `$(@)` in the assertion operator.

---

10. Viewpathing

If you're working on a big project you may consider viewpathing [22] **nmake**'s most valuable feature. It's often described using an elaborate directory structure that's designed to look like a real product. We're only interested in a few easy demonstrations, so we'll take a simpler approach.

10.1 A Source Node

Let's clean up in `/tmp/n2` before we do anything else. That's where we've been working lately, so typing

```
nmake -f prog.mk clobber
```

will do it. We should be left with `main.c`, `a.c`, `b.c`, `prog.mk`, and a few other makefiles that we don't care about anymore. A clean source directory isn't a viewpathing requirement, but we want to make sure something happens when we build `prog` in the empty node that we'll create next.

10.2 An Empty Node

We'll need another directory for our viewpathing demonstrations, so let's type:

```
mkdir /tmp/n1
```

We picked `/tmp/n1` but you can use any directory you want, as long it's empty and you remember to substitute your directory when we use `/tmp/n1`.

10.3 VPATH

There's a shell variable named `VPATH` that tells **nmake** where to look for files. `VPATH` is a colon-separated list of directories, much like the familiar `PATH` variable that the shell consults when it needs to find a command. We'll set our `VPATH` variable using:

```
VPATH=/tmp/n1:/tmp/n2
export VPATH
```

Remember to substitute your directories and export VPATH, and do it at the shell level, not in the makefile. The first directory, in our case /tmp/n1, is called the top node; the last directory in the list is called the bottom node. We're only using two nodes, but viewpaths with three or four nodes are common.

The top node is where you work, and is often called the developer's node. The bottom node is often just source, and is sometimes called the official node. **nmake** starts looking for files in the top node, and continues trying nodes down the viewpath until it either finds the file or runs out of nodes.

10.4 A Build In An Empty Node

We're in /tmp/n1, and even though it's empty, when we type

```
nmake -f prog.mk prog
```

we get:

```
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c /tmp/n2/main.c
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c /tmp/n2/a.c
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c /tmp/n2/b.c
+ cc -O -o prog main.o a.o b.o
```

Our directory was empty when we started, so **nmake** found everything, even prog.mk, by looking at the second node in our VPATH.

10.5 A Build With A Local Source File

If we clean up by typing

```
nmake -f prog.mk clobber
```

we should end up with an empty directory again. The top node is where we're supposed to work, so let's grab a copy of a.c that we can change without disturbing the official version,

```
cp /tmp/n2/a.c .
```

and rebuild prog by typing:

```
nmake -f prog.mk prog
```

This time we get,

```
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c /tmp/n2/main.c
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c a.c
+ cc -O -Qpath /nmake/lib -I-D/nmake/lib/probe/C/pp/835E4F4F5bincc -I- -c /tmp/n2/b.c
+ cc -O -o prog main.o a.o b.o
```

and if you look carefully you'll notice we built prog using our local copy of a.c. If you need proof, change a.c so it prints a different message, rebuild prog, and you'll get the new message when you run prog. Using viewpathing we can have our production source code safely stored in another directory and deal only with the files we're changing for a new version of the program.

FOOTNOTES:

[22] You'll find more details in chapter 10 of the *nmake User's Guide*.

11. Common Actions

There are some chores, like cleaning up and installing files, that should be familiar to **make** users. For example, building a target named `clean` usually removes intermediate (e.g., `.o`) files but not final products (e.g., programs or libraries), while building `clobber` removes everything, including the final products, built by a makefile. Makefiles written for **make** often include targets named `clean`, `clobber`, and `install`, but **nmake** provides these capabilities (and many more) in an extensive collection of targets, called *common actions* [23], that are defined in `Makerules.mk`.

Common actions are important because they're general implementations that don't need to be duplicated in makefiles. As a result you'll rarely find targets like `clean`, `clobber`, or `install` in makefiles written for **nmake**, although they work when you try to build them. For example, we can type

```
nmake -f message.mk clobber
```

and **nmake** does the job without complaining, even though there's no `clobber` target defined in `message.mk`.

11.1 Install

The `install` common action is probably the most frequently used common action. It is primarily used to install files after they have been built or updated, but it can also install files that are not built (for example a data file that must be delivered to customers and is controlled with the source code but not generated.)

A file will only be re-installed if it is different than the currently installed file [24] (according to the UNIX `cmp` command.) Before re-installing a file **nmake** will backup the currently installed file to *`filename`*`.old` [25] in the installation directory.

The `::` assertion operator will setup a default install assertion for you. Executable targets defined on the left side of `::` will be installed in the directory defined by variable `BINDIR`. Archive library (`.a`) files on the left side of `::` will be installed in `LIBDIR`.

```
BINDIR = ../bin
prog :: main.c a.c b.c
```

To install files you specify the `install` common action on the **nmake** command line. Say we have already built this makefile so nothing needs to be re-compiled, **nmake** will just do the install. Of course if anything were out-of-date **nmake** would rebuild whatever is necessary before installing files. So we can type,

```
nmake -f prog.mk install
```

and we get:

```
+ cp prog ../bin/prog
```

You may also define explicit install assertions using the `:INSTALL:` or `:INSTALLDIR:` assertion operators [26]. `:INSTALLDIR:` is most common, it takes a directory on the left side and one or more files on the right to be installed in the specified directory.

```
../mydir :INSTALLDIR: abc xyz
```

When we run

```
nmake -f abc.mk install
```

we get:

```
+ cp abc ../mydir/abc
+ cp xyz ../mydir/xyz
```

The `:INSTALL:` operator takes a file on the left side and one file on the right. The file specified on the right will be installed as the file on the left. Use this operator when you need to rename the file being installed.

11.2 Clobber

The `clobber` common action is what you use when you really want to clean everything up. It removes the generated targets, intermediate files such as the `.o` files, and the **nmake** objectfile and statefile.

When we clobber our `prog.mk` makefile

```
nmake -f prog.mk clobber
```

we get:

```
+ ignore rm -f main.o prog b.o a.o Makefile.mo Makefile.ms
```

You can also clobber only the files installed by the `install` common action. After building `prog.mk` with `install` we can run

```
nmake -f prog.mk clobber.install
```

and get

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

11.3 Clean

The `clean` common action is a little more conservative than `clobber`. `clean` will remove the intermediate files but leave the final targets and **nmake** objectfile and statefile. Running

```
nmake -f prog.mk clean
```

Will give us:

```
+ ignore rm -f main.o b.o a.o
```

FOOTNOTES:

[23]
See chapter 2 of the *nmake User's Guide* for the whole list of common actions support by **nmake**.

[24]
Set variable `compare=0` to turn off this comparison check.

[25]
Set variable `clobber=1` prevent **nmake** from making the *`filename`*`.old` files during install.

[26]
See chapter 3 in the *nmake Reference Manual* for more information on the `:INSTALL:` and `:INSTALLDIR:` assertions.

---

12. Conclusion

We promised to take it easy and we hope you weren't disappointed, even if you did find some of this difficult. When we started our goal was to teach you things about **nmake**. We'll be happy if you leave with a good understanding of assertions and variables, and the ability to experiment - if you do you'll have the tools you need to make good use of the manual.

We want feedback, so send your comments to nmake@alcatel-lucent.com. There's plenty left, so if there's enough interest we'll gladly tackle a more advanced paper.

Appendix

We used **nmake** and the following makefile to build this paper:

```
include scan.mk

paper.ps : paper.text
   troff -mm -mpictures $(*) | dpost >$(<)

ghostview lpr : paper.ps
   $(<) $(OPTIONS) $(*)
```

We haven't explained how to program **nmake**'s dependency scanner, but there's not much involved in what we did. We borrowed `.SCAN.nroff` from `Scanrules.mk`, changed the name to `.SCAN.troff`, deleted two entries, simplified one, and added a line that recognized a macro named `.sX` that we used when we wanted to run **nmake** and include the results in the paper. Our scan rules are shown below:

```
/*
 * Scan rule that recognizes .so, .BP, and our own .sX macro. The
 * The .ATTRIBUTE.%.text definition means using .SCAN.troff
 * to scan files that have a .text suffix.
 */

.SCAN.troff : .SCAN
   I|.so % |
   I|.BP % |
   I|.sX % |

.ATTRIBUTE.%.text : .SCAN.troff
```

The `.ATTRIBUTE.%.text` assertion is more magic that we didn't talk about in the paper, but it's pretty easy. All it does is tell **nmake** to scan source files that end in `.text` (in our case it's `paper.text`) using the rules we defined for `.SCAN.troff`.

Last Update: Friday,12-Aug-2016 12:18:54 EDT