The Power of GNU Make for Building Anything

3rd Edition
Completely Revised & Updated

Managing Projects with

GNU Make

O'REILLY®

Robert Mecklenburg

Managing Projects with GNU Make, Third Edition
by Robert Mecklenburg

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Andy Oram
Production Editor:	Matt Hutchinson
Production Services:	Octal Publishing, Inc.
Cover Designer:	Edie Freedman
Interior Designer:	David Futato

Printing History:

1986:	First Edition.
October 1991:	Second Edition.
November 2004:	Third Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00610-1
[M]

Table of Contents

Foreword

The make utility is an enticing servant, always there and always accommodating. Like the indispensable sidekicks found in many novels and movies, make starts out as the underappreciated supplicant to whom you throw a few odd jobs, and then gradually takes over the entire enterprise.

I had reached the terminal stage of putting make at the center of every project I touched when Steve Talbott, my supervisor and the author of the original O'Reilly classic *Managing Projects with make*, noticed my obsession and asked me to write the second edition. It proved to be a key growth experience for me (as well as a pretty wild ride) and my entry into the wonderful world of O'Reilly, but we didn't really think about how long the result would stay on the market. Thirteen years for one edition?

Enthralled in the memories of those days long ago when I was a professional technical writer, I'll indulge myself with a bulleted list to summarize the evolution of make since the second edition of *Managing Projects with make* hit the stands:

- The GNU version of make, already the choice of most serious coders when the second edition of the book came out, overran the industry and turned into the de facto standard.
- The rise of GNU/Linux made the GNU compiler tool chain even more common, and that includes the GNU version of make. As just one example, the Linux kernel itself relies heavily on extensions provided by GNU make, as documented in Chapter 11 of this book.
- The adoption of a variant of BSD (Darwin) as the core of Mac OS X continues the trend toward the dominance of the GNU tool chain and GNU make.
- More and more tricks are being discovered for using make in a robust, error-free, portable, and flexible way. Standard solutions to common problems on large projects have grown up in the programming community. It's time to move many of these solutions from the realm of folklore to the realm of documented practices, as this book does.

- In particular, new practices are required to adapt make to the C++ and Java™ languages, which did not exist when make was invented. To illustrate the shifting sands of time, the original make contained special features to support two variants of FORTRAN—of which vestiges remain!—and rather ineffective integration with SCCS.)
- Against all odds, make has remained a critical tool for nearly all computer development projects. None of make's many (and insightful) critics would have predicted this 13 years ago. Over these years, replacements sprang up repeatedly, as if dragon's teeth had been sown. Each new tool was supposed to bypass the limitations in make's design, and most were indeed ingenious and admirable. Yet the simplicity of make has kept it supreme.

As I watched these trends, it had been in the back of my mind for about a decade to write a new edition of *Managing Projects with make*. But I sensed that someone with a broader range of professional experience than mine was required. Finally, Robert Mecklenburg came along and wowed us all at O'Reilly with his expertise. I was happy to let him take over the book and to retire to the role of kibitzer, which earns me a mention on the copyright page of this book. (Incidentally, we put the book under the GNU Free Documentation License to mirror the GPL status of GNU make.)

Robert is too modest to tout his Ph.D., but the depth and precision of thinking he must have applied to that endeavor comes through clearly in this book. Perhaps more important to the book is his focus on practicality. He's committed to making make work for you, and this commitment ranges from being alert about efficiency to being clever about making even typographical errors in *makefile*s self-documenting.

This is a great moment: the creation of a new edition of one of O'Reilly's earliest and most enduring books. Sit back and read about how an unassuming little tool at the background of almost every project embodies powers you never imagined. Don't settle for creaky and unsatisfying *makefile*s—expand your potential today.

—Andy Oram
Editor, O'Reilly Media
August 19, 2004

PART I
Basic Concepts

In Part I, we focus on the features of `make`, what they do, and how to use them properly. We begin with a brief introduction and overview that should be enough to get you started on your first *makefile*. The chapters then cover `make` rules, variables, functions, and finally command scripts.

When you are finished with Part I, you will have a fairly complete working knowledge of GNU `make` and have many advanced usages well in hand.

CHAPTER 1

How to Write a Simple Makefile

The mechanics of programming usually follow a fairly simple routine of editing source files, compiling the source into an executable form, and debugging the result. Although transforming the source into an executable is considered routine, if done incorrectly a programmer can waste immense amounts of time tracking down the problem. Most developers have experienced the frustration of modifying a function and running the new code only to find that their change did not fix the bug. Later they discover that they were never executing their modified function because of some procedural error such as failing to recompile the source, relink the executable, or rebuild a jar. Moreover, as the program's complexity grows these mundane tasks can become increasingly error-prone as different versions of the program are developed, perhaps for other platforms or other versions of support libraries, etc.

The `make` program is intended to automate the mundane aspects of transforming source code into an executable. The advantages of `make` over scripts is that you can specify the relationships between the elements of your program to `make`, and it knows through these relationships and timestamps exactly what steps need to be redone to produce the desired program each time. Using this information, `make` can also optimize the build process avoiding unnecessary steps.

GNU `make` (and other variants of `make`) do precisely this. `make` defines a language for describing the relationships between source code, intermediate files, and executables. It also provides features to manage alternate configurations, implement reusable libraries of specifications, and parameterize processes with user-defined macros. In short, `make` can be considered the center of the development process by providing a roadmap of an application's components and how they fit together.

The specification that `make` uses is generally saved in a file named *makefile*. Here is a *makefile* to build the traditional "Hello, World" program:

```
hello: hello.c
        gcc hello.c -o hello
```

To build the program execute `make` by typing:

```
$ make
```

at the command prompt of your favorite shell. This will cause the make program to read the *makefile* and build the first target it finds there:

```
$ make
gcc hello.c -o hello
```

If a target is included as a command-line argument, that target is updated. If no command-line targets are given, then the first target in the file is used, called the *default goal*.

Typically the default goal in most *makefile*s is to build a program. This usually involves many steps. Often the source code for the program is incomplete and the source must be generated using utilities such as `flex` or `bison`. Next the source is compiled into binary object files (*.o* files for C/C++, *.class* files for Java, etc.). Then, for C/C++, the object files are bound together by a linker (usually invoked through the compiler, `gcc`) to form an executable program.

Modifying any of the source files and reinvoking `make` will cause some, but usually not all, of these commands to be repeated so the source code changes are properly incorporated into the executable. The specification file, or *makefile*, describes the relationship between the source, intermediate, and executable program files so that `make` can perform the minimum amount of work necessary to update the executable.

So the principle value of `make` comes from its ability to perform the complex series of commands necessary to build an application and to optimize these operations when possible to reduce the time taken by the edit-compile-debug cycle. Furthermore, `make` is flexible enough to be used anywhere one kind of file depends on another from traditional programming in C/C++ to Java, TEX, database management, and more.

Targets and Prerequisites

Essentially a *makefile* contains a set of rules used to build an application. The first rule seen by `make` is used as the *default rule*. A *rule* consists of three parts: the target, its prerequisites, and the command(s) to perform:

```
target: prereq1 prereq2
        commands
```

The *target* is the file or thing that must be made. The *prerequisites* or *dependents* are those files that must exist before the target can be successfully created. And the *commands* are those shell commands that will create the target from the prerequisites.

Here is a rule for compiling a C file, *foo.c*, into an object file, *foo.o*:

```
foo.o: foo.c foo.h
        gcc -c foo.c
```

The target file *foo.o* appears before the colon. The prerequisites *foo.c* and *foo.h* appear after the colon. The command script usually appears on the following lines and is preceded by a tab character.

When make is asked to evaluate a rule, it begins by finding the files indicated by the prerequisites and target. If any of the prerequisites has an associated rule, make attempts to update those first. Next, the target file is considered. If any prerequisite is newer than the target, the target is remade by executing the commands. Each command line is passed to the shell and is executed in its own subshell. If any of the commands generates an error, the building of the target is terminated and make exits. One file is considered newer than another if it has been modified more recently.

Here is a program to count the number of occurrences of the words "fee," "fie," "foe," and "fum" in its input. It uses a flex scanner driven by a simple main:

```
#include <stdio.h>

extern int fee_count, fie_count, foe_count, fum_count;
extern int yylex( void );

int main( int argc, char ** argv )
{
    yylex();
    printf( "%d %d %d %d\n", fee_count, fie_count, foe_count, fum_count );
    exit( 0 );
}
```

The scanner is very simple:

```
        int fee_count = 0;
        int fie_count = 0;
        int foe_count = 0;
        int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
```

The *makefile* for this program is also quite simple:

```
count_words: count_words.o lexer.o -lfl
        gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
        gcc -c count_words.c

lexer.o: lexer.c
        gcc -c lexer.c

lexer.c: lexer.l
        flex -t lexer.l > lexer.c
```

When this *makefile* is executed for the first time, we see:

```
$ make
gcc -c count_words.c
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

We now have an executable program. Of course, real programs typically consist of more modules than this. Also, as you will see later, this *makefile* does not use most of the features of make so it's more verbose than necessary. Nevertheless, this is a functional and useful *makefile*. For instance, during the writing of this example, I executed the *makefile* several dozen times while experimenting with the program.

As you look at the *makefile* and sample execution, you may notice that the order in which commands are executed by make are nearly the opposite to the order they occur in the *makefile*. This *top-down* style is common in *makefile*s. Usually the most general form of target is specified first in the *makefile* and the details are left for later. The make program supports this style in many ways. Chief among these is make's two-phase execution model and recursive variables. We will discuss these in great detail in later chapters.

Dependency Checking

How did make decide what to do? Let's go over the previous execution in more detail to find out.

First make notices that the command line contains no targets so it decides to make the default goal, *count_words*. It checks for prerequisites and sees three: *count_words.o*, *lexer.o*, and -lfl. make now considers how to build *count_words.o* and sees a rule for it. Again, it checks the prerequisites, notices that *count_words.c* has no rules but that the file exists, so make executes the commands to transform *count_words.c* into *count_words.o* by executing the command:

```
gcc -c count_words.c
```

This "chaining" of targets to prerequisites to targets to prerequisites is typical of how make analyzes a *makefile* to decide the commands to be performed.

The next prerequisite make considers is *lexer.o*. Again the chain of rules leads to *lexer.c* but this time the file does not exist. make finds the rule for generating *lexer.c* from *lexer.l* so it runs the flex program. Now that *lexer.c* exists it can run the gcc command.

Finally, make examines -lfl. The -l option to gcc indicates a system library that must be linked into the application. The actual library name indicated by "fl" is *libfl.a*. GNU make includes special support for this syntax. When a prerequisite of the form -l<NAME> is seen, make searches for a file of the form *libNAME.so*; if no match is found, it then searches for *libNAME.a*. Here make finds */usr/lib/libfl.a* and proceeds with the final action, linking.

Minimizing Rebuilds

When we run our program, we discover that aside from printing fees, fies, foes, and fums, it also prints text from the input file. This is not what we want. The problem is that we have forgotten some rules in our lexical analyzer and `flex` is passing this unrecognized text to its output. To solve this problem we simply add an "any character" rule and a newline rule:

```
            int fee_count = 0;
            int fie_count = 0;
            int foe_count = 0;
            int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
.
\n
```

After editing this file we need to rebuild the application to test our fix:

```
$ make
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

Notice this time the file *count_words.c* was not recompiled. When `make` analyzed the rule, it discovered that *count_words.o* existed and was newer than its prerequisite *count_words.c* so no action was necessary to bring the file up to date. While analyzing *lexer.c*, however, `make` saw that the prerequisite *lexer.l* was newer than its target *lexer.c* so `make` must update *lexer.c*. This, in turn, caused the update of *lexer.o* and then *count_words*. Now our word counting program is fixed:

```
$ count_words < lexer.l
3 3 3 3
```

Invoking make

The previous examples assume that:

- All the project source code and the `make` description file are stored in a single directory.
- The `make` description file is called *makefile*, *Makefile*, or *GNUMakefile*.
- The *makefile* resides in the user's current directory when executing the `make` command.

When make is invoked under these conditions, it automatically creates the first target it sees. To update a different target (or to update more than one target) include the target name on the command line:

```
$ make lexer.c
```

When make is executed, it will read the description file and identify the target to be updated. If the target or any of its prerequisite files are out of date (or missing) the shell commands in the rule's command script will be executed one at a time. After the commands are run make assumes the target is up to date and moves on to the next target or exits.

If the target you specify is already up to date, make will say so and immediately exit, doing nothing else:

```
$ make lexer.c
make: `lexer.c' is up to date.
```

If you specify a target that is not in the *makefile* and for which there is no implicit rule (discussed in Chapter 2), make will respond with:

```
$ make non-existent-target
make: *** No rule to make target `non-existent-target'.  Stop.
```

make has many command-line options. One of the most useful is `--just-print` (or `-n`) which tells make to display the commands it would execute for a particular target without actually executing them. This is particularly valuable while writing *makefile*s. It is also possible to set almost any *makefile* variable on the command line to override the default value or the value set in the *makefile*.

Basic Makefile Syntax

Now that you have a basic understanding of make you can almost write your own *makefile*s. Here we'll cover enough of the syntax and structure of a *makefile* for you to start using make.

*Makefile*s are usually structured top-down so that the most general target, often called `all`, is updated by default. More and more detailed targets follow with targets for program maintenance, such as a `clean` target to delete unwanted temporary files, coming last. As you can guess from these target names, targets do not have to be actual files, any name will do.

In the example above we saw a simplified form of a rule. The more complete (but still not quite complete) form of a rule is:

$$\begin{aligned} &target_1\ target_2\ target_3 : prerequisite_1\ prerequisite_2 \\ &\quad command_1 \\ &\quad command_2 \\ &\quad command_3 \end{aligned}$$

One or more targets appear to the left of the colon and zero or more prerequisites can appear to the right of the colon. If no prerequisites are listed to the right, then only the target(s) that do not exist are updated. The set of commands executed to update a target are sometimes called the *command script*, but most often just the *commands*.

Each command *must* begin with a tab character. This (obscure) syntax tells `make` that the characters that follow the tab are to be passed to a subshell for execution. If you accidentally insert a tab as the first character of a noncommand line, `make` will interpret the following text as a command under most circumstances. If you're lucky and your errant tab character is recognized as a syntax error you will receive the message:

```
$ make
Makefile:6: *** commands commence before first target.  Stop.
```

We'll discuss the complexities of the tab character in Chapter 2.

The comment character for `make` is the hash or pound sign, #. All text from the pound sign to the end of line is ignored. Comments can be indented and leading whitespace is ignored. The comment character # does not introduce a `make` comment in the text of commands. The entire line, including the # and subsequent characters, is passed to the shell for execution. How it is handled there depends on your shell.

Long lines can be continued using the standard Unix escape character backslash (\). It is common for commands to be continued in this way. It is also common for lists of prerequisites to be continued with backslash. Later we'll cover other ways of handling long prerequisite lists.

You now have enough background to write simple *makefile*s. Chapter 2 will cover rules in detail, followed by `make` variables in Chapter 3 and commands in Chapter 5. For now you should avoid the use of variables, macros, and multiline command sequences.

CHAPTER 2

Rules

In the last chapter, we wrote some rules to compile and link our word-counting program. Each of those rules defines a target, that is, a file to be updated. Each target file depends on a set of prerequisites, which are also files. When asked to update a target, make will execute the command script of the rule if any of the prerequisite files has been modified more recently than the target. Since the target of one rule can be referenced as a prerequisite in another rule, the set of targets and prerequisites form a chain or graph of *dependencies* (short for "dependency graph"). Building and processing this dependency graph to update the requested target is what make is all about.

Since rules are so important in make, there are a number of different kinds of rules. *Explicit rules*, like the ones in the previous chapter, indicate a specific target to be updated if it is out of date with respect to any of its prerequisites. This is the most common type of rule you will be writing. *Pattern rules* use wildcards instead of explicit filenames. This allows make to apply the rule any time a target file matching the pattern needs to updated. *Implicit rules* are either pattern rules or suffix rules found in the rules database built-in to make. Having a built-in database of rules makes writing *makefile*s easier since for many common tasks make already knows the file types, suffixes, and programs for updating targets. *Static pattern rules* are like regular pattern rules except they apply only to a specific list of target files.

GNU make can be used as a "drop in" replacement for many other versions of make and includes several features specifically for compatibility. *Suffix rules* were make's original means for writing general rules. GNU make includes support for suffix rules, but they are considered obsolete having been replaced by pattern rules that are clearer and more general.

Explicit Rules

Most rules you will write are explicit rules that specify particular files as targets and prerequisites. A rule can have more than one target. This means that each target has

the same set of prerequisites as the others. If the targets are out of date, the same set of actions will be performed to update each one. For instance:

```
vpath.o variable.o: make.h config.h getopt.h gettext.h dep.h
```

This indicates that both *vpath.o* and *variable.o* depend on the same set of C header files. This line has the same effect as:

```
vpath.o: make.h config.h getopt.h gettext.h dep.h
variable.o: make.h config.h getopt.h gettext.h dep.h
```

The two targets are handled independently. If either object file is out of date with respect to any of its prerequisites (that is, any header file has a newer modification time than the object file), `make` will update the object file by executing the commands associated with the rule.

A rule does not have to be defined "all at once." Each time `make` sees a target file it adds the target and prerequisites to the dependency graph. If a target has already been seen and exists in the graph, any additional prerequisites are appended to the target file entry in `make`'s dependency graph. In the simple case, this is useful for breaking long lines naturally to improve the readability of the *makefile*:

```
vpath.o: vpath.c make.h config.h getopt.h gettext.h dep.h
vpath.o: filedef.h hash.h job.h commands.h variable.h vpath.h
```

In more complex cases, the prerequisite list can be composed of files that are managed very differently:

```
# Make sure lexer.c is created before vpath.c is compiled.
vpath.o: lexer.c
...
# Compile vpath.c with special flags.
vpath.o: vpath.c
        $(COMPILE.c) $(RULE_FLAGS) $(OUTPUT_OPTION) $<
...
# Include dependencies generated by a program.
include auto-generated-dependencies.d
```

The first rule says that the *vpath.o* target must be updated whenever *lexer.c* is updated (perhaps because generating *lexer.c* has other side effects). The rule also works to ensure that a prerequisite is always updated before the target is updated. (Notice the bidirectional nature of rules. In the "forward" direction the rule says that if the *lexer.c* has been updated, perform the action to update *vpath.o*. In the "backward" direction, the rule says that if we need to make or use *vpath.o* ensure that *lexer.c* is up to date first.) This rule might be placed near the rules for managing *lexer.c* so developers are reminded of this subtle relationship. Later, the compilation rule for *vpath.o* is placed among other compilation rules. The command for this rule uses three `make` variables. You'll be seeing a lot of these, but for now you just need to know that a variable is either a dollar sign followed by a single character or a dollar sign followed by a word in parentheses. (I will explain more later in this chapter and

a lot more in Chapter 3.) Finally, the *.o*/*.h* dependencies are included in the *makefile* from a separate file managed by an external program.

Wildcards

A *makefile* often contains long lists of files. To simplify this process make supports wildcards (also known as *globbing*). make's wildcards are identical to the Bourne shell's: ~, *, ?, [...], and [^...]. For instance, *.* expands to all the files containing a period. A question mark represents any single character, and [...] represents a *character class*. To select the "opposite" (negated) character class use [^...].

In addition, the tilde (~) character can be used to represent the current user's home directory. A tilde followed by a user name represents that user's home directory.

Wildcards are automatically expanded by make whenever a wildcard appears in a target, prerequisite, or command script context. In other contexts, wildcards can be expanded explicitly by calling a function. Wildcards can be very useful for creating more adaptable *makefile*s. For instance, instead of listing all the files in a program explicitly, you can use wildcards:*

```
prog: *.c
        $(CC) -o $@ $^
```

It is important to be careful with wildcards, however. It is easy to misuse them as the following example shows:

```
*.o: constants.h
```

The intent is clear: all object files depend on the header file *constants.h*, but consider how this expands on a clean directory without any object files:

```
: constants.h
```

This is a legal make expression and will not produce an error by itself, but it will also not provide the dependency the user wants. The proper way to implement this rule is to perform a wildcard on the source files (since they are always present) and transform that into a list of object files. We will cover this technique when we discuss make functions in Chapter 4.

Finally, it is worth noting that wildcard expansion is performed by make when the pattern appears as a target or prerequisite. However, when the pattern appears in a command, the expansion is performed by the subshell. This can occasionally be important because make will expand the wildcards immediately upon reading the *makefile*, but the shell will expand the wildcards in commands much later when the command is executed. When a lot of complex file manipulation is being done, the two wildcard expansions can be quite different.

* In more controlled environments using wildcards to select the files in a program is considered bad practice because a rogue source file might be accidentally linked into a program.

Phony Targets

Until now all targets and prerequisites have been files to be created or updated. This is typically the case, but it is often useful for a target to be just a label representing a command script. For instance, earlier we noted that a standard first target in many *makefile*s is called `all`. Targets that do not represent files are known as *phony targets*. Another standard phony target is `clean`:

```
clean:
        rm -f *.o lexer.c
```

Normally, phony targets will always be executed because the commands associated with the rule do not create the target name.

It is important to note that `make` cannot distinguish between a file target and phony target. If by chance the name of a phony target exists as a file, `make` will associate the file with the phony target name in its dependency graph. If, for example, the file *clean* happened to be created running `make clean` would yield the confusing message:

```
$ make clean
make: `clean' is up to date.
```

Since most phony targets do not have prerequisites, the `clean` target would always be considered up to date and would never execute.

To avoid this problem, GNU `make` includes a special target, `.PHONY`, to tell `make` that a target is not a real file. Any target can be declared phony by including it as a prerequisite of `.PHONY`:

```
.PHONY: clean
clean:
        rm -f *.o lexer.c
```

Now `make` will always execute the commands associated with `clean` even if a file named *clean* exists. In addition to marking a target as always out of date, specifying that a target is phony tells `make` that this file does not follow the normal rules for making a target file from a source file. Therefore, `make` can optimize its normal rule search to improve performance.

It rarely makes sense to use a phony target as a prerequisite of a real file since the phony is always out of date and will always cause the target file to be remade. However, it is often useful to give phony targets prerequisites. For instance, the `all` target is usually given the list of programs to be built:

```
.PHONY: all
all: bash bashbug
```

Here the `all` target creates the `bash` shell program and the `bashbug` error reporting tool.

Phony targets can also be thought of as shell scripts embedded in a *makefile*. Making a phony target a prerequisite of another target will invoke the phony target script

before making the actual target. Suppose we are tight on disk space and before executing a disk-intensive task we want to display available disk space. We could write:

```
.PHONY: make-documentation
make-documentation:
        df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
        javadoc ...
```

The problem here is that we may end up specifying the `df` and `awk` commands many times under different targets, which is a maintenance problem since we'll have to change every instance if we encounter a `df` on another system with a different format. Instead, we can place the `df` line in its own phony target:

```
.PHONY: make-documentation
make-documentation: df
        javadoc ...

.PHONY: df
df:
        df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
```

We can cause `make` to invoke our `df` target before generating the documentation by making `df` a prerequisite of `make-documentation`. This works well because `make-documentation` is also a phony target. Now I can easily reuse `df` in other targets.

There are a number of other good uses for phony targets.

The output of `make` can be confusing to read and debug. There are several reasons for this: *makefile*s are written top-down but the commands are executed by `make` bottom-up; also, there is no indication which rule is currently being evaluated. The output of `make` can be made much easier to read if major targets are commented in the `make` output. Phony targets are a useful way to accomplish this. Here is an example taken from the `bash` *makefile*:

```
$(Program): build_msg $(OBJECTS) $(BUILTINS_DEP) $(LIBDEP)
        $(RM) $@
        $(CC) $(LDFLAGS) -o $(Program) $(OBJECTS) $(LIBS)
        ls -l $(Program)
        size $(Program)

.PHONY: build_msg
build_msg:
        @printf "#\n# Building $(Program)\n#\n"
```

Because the `printf` is in a phony target, the message is printed immediately before any prerequisites are updated. If the build message were instead placed as the first command of the `$(Program)` command script, then it would be executed after all compilation and dependency generation. It is important to note that because phony targets are always out of date, the phony `build_msg` target causes `$(Program)` to be regenerated even when it is not out of date. In this case, it seems a reasonable choice since most of the computation is performed while compiling the object files so only the final link will always be performed.