



# GNU make

A Program for Directing Recompilation

## Auto-Dependency Generation

June 23rd, 2000 (updated July 4th, 2020)

*One of the most important, and yet potentially frustrating, tasks that is required to allow any make-based build environment to function properly is the correct listing of dependencies in the makefile.*

*This document describes a very useful method for having make itself create and maintain these dependencies completely automatically.*

*The basics of the advanced method below were invented by Tom Tromey <[tromey@cygnus.com](mailto:tromey@cygnus.com)>, I merely wrote it down here. Credits for the method go to him; problems with the explanation belong to me.*

- [TL;DR: The GCC Solution](#)
- Traditional make depend Method
- [The GNU make include Directive](#)
- [Basic Auto-Dependencies](#)
- [Advanced Auto-Dependencies](#)
  - [Avoiding Re-exec of make](#)
  - [Avoiding "No rule to make target ..." Errors](#)
  - [Handling Deleted Dependency Files](#)
- [Placement of Output Files](#)
- [Defining MAKEDEPEND](#)
  - `MAKEDEPEND = /usr/lib/cpp or cc -E`
  - `MAKEDEPEND = makedepend`

- [Combining Compilation and Dependency Generation](#)
  - [Handling unusual situations](#)
- [Placement of Object Files](#)
- [Dependencies For Non-C Files](#)

All make programs must know, with great accuracy, what files a particular target is dependent on in order to ensure that it is rebuilt when (and only when) necessary.

Keeping this list up-to-date by hand is not only tedious, but quite error prone. Most systems of any size prefer to provide automated tools for extracting this information. The traditional tool was the `makedepend` program, which reads C source files and generates a list of the header files in a target-dependency format suitable for inclusion in (or appending to) a makefile.

The modern solution, for those using suitably enlightened compilers or preprocessors (such as GCC), is to have the compiler or preprocessor generate this information.

The purpose of this paper isn't primarily to discuss ways in which this dependency information can be obtained, although this is covered. Rather, it describes some useful ways to combine the invocation and output of these tools with GNU make to ensure that dependency information is always accurate and up-to-date, as seamlessly (and efficiently) as possible.

These methods rely on features provided by GNU make. It may be possible to modify them to work with other versions of make; that's left as an exercise to the reader. However, before undertaking that exercise please review [Paul's First Rule of Makefiles](#) :).

## TL;DR: The GCC Solution

For those who are impatient, here's a complete best-practice solution. This solution requires support from your compiler: it assumes you are using [GCC](#) as your compiler (or a compiler which provides preprocessor flags compatible with GCC). If your compiler doesn't meet this criteria, keep reading for alternatives.

Add this to your makefile environment (the sections in blue are the changes to the built-in content provided by GNU make). Of course you can omit pattern rules which don't fit your environment (or add new ones as needed):

```
DEPDIR := .deps
DEPFLAGS = -MT $@ -MMD -MP -MF $(DEPDIR)/$.d

COMPILE.c = $(CC) $(DEPFLAGS) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c

%.o : %.c
%.o : %.c $(DEPDIR)/%.d | $(DEPDIR)
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

```
$(DEPDIR): ; @mkdir -p $@

DEPFILES := $(SRCS:%.c=$(DEPDIR)/%.d)
$(DEPFILES):

include $(wildcard $(DEPFILES))
```

Be aware that the `include` line must come *after* the initial, default target; otherwise the included dependency files will abscond with your default target setting. It's fine to add this to the very end of your makefile (or, keep it in a separate makefile and include it).

Also, this assumes the `SRCS` variable contains all the source files (not header files) you want to track dependencies for.

If you just want to understand what these changes mean, and consider some issues and solutions to them, [jump to the description](#). If you want a deeper understanding of how this works, and to learn some interesting things about GNU make, keep reading.

## Traditional `make depend` Method

The time-honored method of handling dependency generation is to provide a special target in your makefiles, typically `depend`, which can be invoked to create dependency information. The command for this target will invoke some kind of dependency tracking tool on all the relevant files in the directory, which will generate makefile-formatted dependency information.

If your version of `make` supports `include` you can redirect this to a file and simply include the file. If not, this usually also involves some shell hackery to append the list of dependencies generated to the end of the makefile itself.

Although it's simple, there are serious problems with this method. First and foremost is that dependencies are only rebuilt when the user explicitly requests it; if the user doesn't run `make depend` regularly they could become badly out-of-date and `make` will not properly rebuild targets. Thus, we cannot say this is seamless and accurate.

A secondary problem is that it is inefficient to run `make depend`, particularly after the first time. Since it modifies makefiles, you typically must do it as a separate build step, which means an extra invocation of every `make` in every subdirectory, etc., in addition to the overhead of the dependency-generation tool itself. Also, it rechecks every file's dependencies, even those which haven't changed.

So, we'll see how we can do better.

## The GNU make `include` Directive

Most versions of `make` support some sort of `include` directive (and indeed, `include` is required by the latest POSIX specification). Unsurprisingly, this allows one makefile to include other makefiles, as if they had been entered there.

One can immediately see how this would be useful, simply to avoid appending dependency information to a makefile as in the step above. However, there is a more [interesting capability](#) in GNU `make`'s handling of `include`: just as with the normal makefile, GNU `make` will attempt to *rebuild* the included makefile. If it is successfully rebuilt, GNU `make` will re-execute itself to read the new version.

This auto-rebuild feature can be harnessed to avoid requiring a separate `make depend` step: if you list all the source files as prerequisites to the file containing dependency information, then include that file into your makefile, it will be rebuilt every time a source file changed. As a result, the dependency information will always be up-to-date and the user doesn't need to run `make depend` explicitly.

Of course, this means dependency information is recalculated for *all* files every time *any* file changes, which is unfortunate. We can still do better than this.

For a detailed description of GNU `make`'s automatic rebuild feature, see the GNU `make` User's Manual, section *"How Makefiles Are Remade"*.

## Basic Auto-Dependencies

The GNU `make` User's Manual describes one way of handling auto-dependencies in section [Generating Dependencies Automatically](#).

In this method, a separate dependency file is created for each source file (in our examples we'll use a `.d` suffix on the base filename). This file contains a rule for the target that is created from that source file, listing the generated prerequisites of the target.

These dependency files are then all included by the makefile. An implicit rule is provided that describes how the dependency files are to be created. In short, something like this:

```
SRCS = foo.c bar.c ...

%.d : %.c
    $(MAKEDEPEND)

include $(SRCS:.c=.d)
```

In these examples I'll use the variable `$(MAKEDEPEND)` to stand for whatever method you choose for creating dependency files. Some possible values for this variable are described below.

What is the format of the generated dependency file? In this simple case, we need to declare that both the object file and the dependency file have the same set of prerequisites: the source file and all the header files. So a `foo.d` file might contain this:

```
foo.o foo.d: foo.c foo.h bar.h baz.h
```

Whenever GNU make reads this makefile, before it does anything else it will try to rebuild the included makefiles, in this case the `.d` files. We have a rule to build them, and we have a list of prerequisites which is the same as the list for the `.o` file itself. So, if any file changes that would cause the original target to appear out-of-date, it will also cause the `.d` file to be rebuilt.

Thus, when any source file or included file changes, make will rebuild the `.d` file(s), re-execute itself to read in the new makefiles, then continue with builds as usual, now with an updated and accurate dependency list.

Here we solve the two problems with the earlier solutions. First, the user doesn't have to do anything special to ensure accurate dependency lists; make will take care of it. Second, we are only updating dependency lists for those file which have actually changed, not all the files in the directory.

We have three new problems with this method, however. The first is still efficiency. Although we only re-examine changed files we still will re-exec make if anything changes, which could be slow for large build systems.

The second problem is merely annoying: when you add a new file or build for the first time, no `.d` file will exist. When make tries to include it and discovers it doesn't exist, it will generate a warning. GNU make will then proceed to rebuild the `.d` file and re-invoke itself, so this is not fatal; nevertheless it's annoying.

The third problem is more serious: if you remove or rename a prerequisite file (say a `C.h` file), make will stop with a fatal error, complaining that the target doesn't exist:

```
make: *** No rule to make target 'bar.h', needed by 'foo.d'. Stop.
```

This is because the `.d` file has a dependency on a file make can't find. It can't rebuild the `.d` file until all the prerequisites are there, and it won't realize it doesn't need the prerequisite until it rebuilds the `.d` file. Catch-22.

The only solution here is to go in by hand and remove any `.d` file that refers to the missing file—typically it's simpler to remove all of them than try to find the proper ones. You can even create a `clean-deps` target or similar to do it automatically (investigate the `MAKECMDGOALS` variable to see how you can write this target while still avoiding the rebuild attempt on the `.d` files). This is annoying to be sure, but given that files aren't removed or renamed all that often in a typical environment perhaps it's not fatal.

## Advanced Auto-Dependencies

The basis for the method described here was engineered by Tom Tromey <[tromey@cygnus.com](mailto:tromey@cygnus.com)>, who used it as the standard dependency generation method for the FSF's `automake` tool. I've made some changes to allow it to fit into a more generic build environment.

### Avoiding Re-exec of `make`

Let's address the first problem above: the re-invocation of `make`. If you think about it, this re-invocation is really not necessary. Since we know *some* prerequisite of the target changed, we must rebuild the target; having a more up-to-date list won't affect that decision. What we really need is to ensure that the prerequisite list is up-to-date for the *next* invocation of `make`, when we need to decide whether to update it again.

Since we don't need the up-to-date prerequisite list in this build, we can actually avoid re-invoking `make` at all: we can simply have the prerequisite list built *at the same time* as the target is rebuilt. In other words, we can change the build rule for our target to add in commands to update the dependency file.

In this case we must be very careful that we *don't* provide rules to build the dependencies automatically: if we do `make` will still try to rebuild them and re-exec: we don't want that.

Now that we don't care about dependency files that don't exist, solving the second problem (superfluous warnings) is easy: we can just use GNU `make`'s `wildcard` function so that dependency files which don't exist won't cause an error.

Let's take a look at an example so far:

```
SRCS = foo.c bar.c ...

%.o : %.c
    @$(MAKEDEPEND)
    $(COMPILE.c) -o $@ $<

include $(wildcard $(SRCS:.c=.d))
```

### Avoiding "No rule to make target ..." Errors

This one is a little trickier. However, it turns out we can convince `make` to not fail merely by mentioning the file explicitly as a target in the makefile. If a target exists, but has no commands (either implicit or explicit) or prerequisites, then `make` simply always considers it up-to-date. That's the normal case, and it behaves as we'd expect.

In the case where the above error occurs, the target *doesn't* exist. According to the GNU make User's Manual section [Rules without Recipes or Prerequisites](#):

*If a rule has no prerequisites or recipe, and the target of the rule is a nonexistent file, then `make' imagines this target to have been updated whenever its rule is run. This implies that all targets depending on this one will always have their recipes run.*

Perfect. It ensures make won't throw an error since it knows how to handle that non-existent file, and it ensures that any file depending on that target is rebuilt, which is exactly what we want.

So, all we need to do is modify the dependency file output so that each prerequisite (source and header file) are defined as targets with no commands or prerequisites. So the output of our MAKEDEPEND script would generate a foo.d file something like this:

```
foo.o: foo.c foo.h bar.h baz.h
foo.c foo.h bar.h baz.h:
```

Thus the .d file contains the original prerequisites definition, then adds each source file as an explicit target.

## Handling Deleted Dependency Files

There is one remaining issue with this configuration: if a user happens to delete dependency files without modifying any of the source files, make will not notice anything amiss and will not recreate the dependency files until it decides to rebuild the corresponding object file for some other reason. In the meantime make will be missing dependency information for those targets (so, for example, modifying a header file without changing the source file will not cause the object file to be rebuilt).

The problem is slightly complex because we do *not* want the dependency files to be considered “real” targets: if they are then when we use `include` to include them, make will rebuild them then re-exec itself. This isn't the end of the world, but it's overhead we'd prefer to do without.

The automake method doesn't address this problem. In the past I've suggested a “just don't do that” solution, combined with placing dependency files [in a separate directory](#) to make it difficult to accidentally delete them.

However, [Lukas Waymann](#) suggested a tidy solution: add the dependency file as a *prerequisite* to the target, and create an empty recipe for it:

```
SRCS = foo.c bar.c ...

%.o : %.c %.d
```

```

        @$(MAKEDEPEND)
        $(COMPILE.c) -o $@ $<

%.d: ;
include $(wildcard $(SRCS:.c=.d))

```

This solves the problem nicely: when make checks the target it will see the dependency file as a prerequisite and try to build that. If it exists nothing will be done since there's no prerequisites for the dependency file. If it doesn't exist, it will be marked out of date since it has an empty recipe, which will force the object target to be rebuilt (creating a new dependency file).

When make is trying to rebuild included files it will find the implicit rule for the dependency and use it. However since the rule doesn't update the target file, no included file was updated and make will not re-execute itself.

One problem with the above implementation is that make considers the .d files to be intermediate files and will delete them. I resolve this by declaring them as explicit targets rather than using a pattern rule:

```

DEPFILES := $(SRCS:.c=.d)
$(DEPFILES):
include $(wildcard $(DEPFILES))

```

## Placement of Output Files

You may decide you don't like all those .d files cluttering up your source directory. You can easily have your makefile put them somewhere else. Here's an example of doing that. Of course this assumes that you've modified your MAKEDEPEND value to generate output to this location as well—be aware that it may need to create this directory before writing to it! See below for another way to handle this, with order-only prerequisites.

```

SRCS = foo.c bar.c ...

DEPDIR = .deps

%.o : %.c $(DEPDIR)/%.d
        @$(MAKEDEPEND)
        $(COMPILE.c) -o $@ $<

DEPFILES := $(SRCS:%.c=$(DEPDIR)/%.d)
$(DEPFILES):
include $(wildcard $(DEPFILES))

```

## Defining MAKEDEPEND

Here I'll discuss some possible ways to define the MAKEDEPEND variable I've blithely been using above.



**MAKEDEPEND = /usr/lib/cpp or cc -E**

The most basic way to generate dependencies is by using the C preprocessor itself. This requires a bit of knowledge about the output format of your preprocessor—luckily most UNIX preprocessors have similar-enough output for our purposes. In order to preserve line number information for the compiler's error messages and debugging information, the output of the preprocessor must provide line number and *filename* information for each jump to an `#include` file and each return from one. These lines can be used to figure out what files were included.

Most UNIX preprocessors insert special lines in the output with this format:

```
# lineno "filename" extra
```

All we care about is the *filename* value. Using this, we can generate `.d` files in the format we desire with this command:

```
MAKEDEPEND = $(CPP) $(CPPFLAGS) $< \
    | sed -n 's,^\# *\[0-9\]\[0-9\]* *"\([^<]*\)" .*,$@: \1\n\1:,p' \
    | sort -u > $*.d
```

**MAKEDEPEND = makedepend**

The X Windowing System source tree comes with a program called `makedepend`. This program examines C source and header files and generates make dependency lines. It's geared towards adding those dependencies to the bottom of an existing makefile, and it is missing a number of capabilities. However for our purposes it can work well.

You can use the following setting to generate `.d` files with the format we want:

```
MAKEDEPEND = makedepend -f- $(CPPFLAGS) $< \
    | sed -n 's,^[^#\][^:]*: *\([^..*\]),$@: \1\n\1:,p' > $*.d
```

**Combining Compilation and Dependency Generation**

One continuing issue above is that we need to preprocess the source file twice: once in the `MAKEDEPEND` command and then again as part of the compilation step.

If you're using GCC (or a compiler which provides equivalent options) you can save yourself a lot of time during the build by *combining* the dependency generation and the object file generation into a single command, because these compilers have the ability to output the dependency information as a side-effect of the compilation. Here is an example implementation, copied from the *TL;DR* section at the top of this page:

```

DEPDIR := .deps
DEPFLAGS = -MT $@ -MMD -MP -MF $(DEPDIR)/$.d

COMPILE.c = $(CC) $(DEPFLAGS) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c

%.o : %.c
%.o : %.c $(DEPDIR)/%.d | $(DEPDIR)
    $(COMPILE.c) $(OUTPUT_OPTION) $<

$(DEPDIR): ; @mkdir -p $@

DEPFILES := $(SRCS:%.c=$(DEPDIR)/%.d)
$(DEPFILES):
include $(wildcard $(DEPFILES))

```

Let's go over the parts in blue, above, which are the differences from the standard make built-in rules:

#### **DEPDIR = ...**

This implementation places dependency files into a subdirectory named `.deps`.

#### **DEPFLAGS = ...**

These are the special GCC-specific flags which convince the compiler to generate the dependency file. Full descriptions can be found in [the GCC manual](#) section [Options Controlling the Preprocessor](#):

##### **-MT \$@**

Set the name of the target in the generated dependency file.

##### **-MMD**

Generate dependency information as a side-effect of compilation, not instead of compilation.

This version omits system headers from the generated dependencies: if you prefer to preserve system headers as prerequisites, use `-MD`.

##### **-MP**

Adds a target for each prerequisite in the list, to avoid errors when deleting files.

##### **-MF \$(DEPDIR)/\$.d**

Write the generated dependency file `$(DEPDIR)/$.d`.

##### **%.o : %.c**

Delete the built-in rules for building object files from `.c` files, so that our rule is used instead.

##### **... \$(DEPDIR)/%.d**

Declare the generated dependency file as a prerequisite of the target, so that if it's missing the target will be rebuilt.

**... | \$(DEPDIR)**

Declare the dependency directory as an order-only prerequisite of the target, so that it will be created when needed.

**\$(DEPDIR): ; @mkdir -p \$@**

Declare a rule for creating the dependency directory if it doesn't exist.

**DEPFILES := ...**

Generate a list of all the dependency files that could exist.

**\$(DEPFILES):**

Mention each dependency file as a target, so that make won't fail if the file doesn't exist.

**include ...**

Include the dependency files that exist. Use wildcard to avoid failing on non-existent files.

## Handling unusual situations

There have been reports of some problems with this solution in certain unusual situations:

- If the build is killed at just the wrong moment, a dependency file can be corrupted. This could cause subsequent invocations of make to fail with syntax errors, etc. trying to read the dependency file. To fix this you must delete the file by hand.
- In some cases it's been reported that GCC will set the timestamps incorrectly such that the dependency file it generates has a *more recent* modification time than the object file. In that situation you get spurious rebuilds of the object file.

You can fix both of these with some defensive coding in your makefile to harden it against problems. For example, something like:

```
DEPDIR := .deps
DEPFLAGS = -MT $@ -MMD -MP -MF $(DEPDIR)/$.Td
POSTCOMPILE = mv -f $(DEPDIR)/$.Td $(DEPDIR)/$.d && touch $@

COMPILE.c = $(CC) $(DEPFLAGS) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c

%.o : %.c
%.o : %.c $(DEPDIR)/%.d | $(DEPDIR)
    $(COMPILE.c) $(OUTPUT_OPTION) $<
    $(POSTCOMPILE)
```

```
$(DEPDIR): ; @mkdir -p $@

DEPFILES := $(SRCS:%.c=$(DEPDIR)/%.d)
$(DEPFILES):
include $(wildcard $(DEPFILES))
```

Here we change DEPFLAGS to write to a temporary file, then add a POSTCOMPILE step, which is only run if the compilation succeeds, that renames the temporary file to the good file and finally, touches the object file to make sure it's newer than the dependency file.

## Placement of Object Files

Often you will want to put object files in a remote location as well, not just dependency files. Here's a quick example of this:

```
OBJDIR := obj

DEPDIR := $(OBJDIR)/.deps
DEPFLAGS = -MT $@ -MMD -MP -MF $(DEPDIR)/$.d

COMPILE.c = $(CC) $(DEPFLAGS) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c

$(OBJDIR)/%.o : %.c $(DEPDIR)/%.d | $(DEPDIR)
    $(COMPILE.c) $(OUTPUT_OPTION) $<

$(DEPDIR): ; @mkdir -p $@

DEPFILES := $(SRCS:%.c=$(DEPDIR)/%.d)
$(DEPFILES):
include $(wildcard $(DEPFILES))
```

## Dependencies For Non-C Files

In general you need some way of producing dependency files in order to use these methods. If you're working with files that aren't C files you'll need to discover or write your own method. Anything that generates make dependency files will do. This usually isn't too difficult.

An interesting idea has been proposed by Han-Wen Nienhuys [hanwen@cs.uu.nl](mailto:hanwen@cs.uu.nl) and he has a small "proof of concept" [implementation](#), although it currently only works on Linux. He suggests using the LD\_PRELOAD environment to insert special shared library that contains a replacement for the open(2) system call. This version of open() would actually write out make dependency information for every file the commands read during operation. This would give you completely reliable dependency information for every kind of command without

needing any special dependency extraction tools at all. In his proof-of-concept implementation you can control the output file and exclude some kinds of files (shared libraries, maybe) via environment variables.

If you are interested in this concept, you should investigate the [Audited Objects](#) project from David Boyce <[boyski@users.sourceforge.net](mailto:boyski@users.sourceforge.net)>. This project provides all the pieces needed to automatically track dependencies, plus much, much more.

---