

An introduction to makefiles

By Daniel Stødle
daniels@stud.cs.uit.no

© 2003 all rights reserved.

This text will give you a quick introduction to understanding makefiles, as well as give you pointers to how you can make your own. It covers the basics of makefiles in a straightforward fashion, but avoids discussing conditionals and some of make's finer details. This is in no way intended to be a complete walkthrough of make, the make man pages are there for a reason ;)

A makefile is the configuration file parsed by the “make” executable. It consists of a number of variables, targets and rules. A variable is very similar to a shell variable. You assign values to it in this fashion (note that the ‘#’ character is used to start a comment):

```
# It is a makefile convention to set the CC variable
# to the executable used to compile C files
CC          = gcc
CFLAGS      = -Wall
MY_SOURCES  = main.c init.c utils.c
```

The make environment also allows you to replace text in one variable with some pattern, such as

```
MY_OBJECTS = $(MY_SOURCES:.c=.o)
```

The MY_OBJECTS variable will now expand to “main.o init.o utils.o” (without the quotes). The next thing to consider is targets. A target is defined by its name, and a number of dependencies (separated by a colon – the different dependencies are separated by a space). Almost every makefile out there defines the “all” target, which is the default build target. When you type “make” on the command line, this is equivalent to typing “make all”. It is also customary to define the “clean” and “distclean” targets, which remove object-files, executables and so on, essentially “cleaning” the source. Let's take a look at some simple target definitions:

```
all:
    $(CC) $(CFLAGS) $(MY_SOURCES) -o MyApp

clean:
```

```
rm -f *.o
rm -f MyApp
```

```
distclean: clean
```

The all-target here does nothing more than call the C-compiler to compile all our sources, producing the MyApp executable as a result. The clean-target removes all object files, and the executable (if it exists). The distclean-target has an associated dependency, that is, it requires that the target “clean” is up to date before it can be built. In this case, the distclean- and clean-targets are equal, since the distclean-target doesn’t define any additional commands compared to the clean-target. Note that a target may have more than one command attached (as is the case for the clean-target)). A command line *has* to start with a tab, otherwise make will not recognize it as a command.

The example given above isn’t very efficient, as it makes the C-compiler recompile all the C-files, even if you have only made changes to one of your files, say, main.c. This can be avoided by providing a number of dependencies for the all-target, as well as using implicit rules to rebuild modified files. An implicit rule is a rule that implicitly states how to build a certain type of file. For instance, we might define an implicit rule that states how to create an object file from a C-file:

```
%.o:%.c
$(CC) $(CFLAGS) -c -o $@ $<
```

The first line tells make that any file ending in .o can be made from a .c file with the same name. For instance, we can make “main.o” from “main.c” by executing the rules following the target definition. The next line gives the rule that builds the .o-file. Here we make use of two automatic variables – the \$@ variable contains the name of the current output-file (ie, main.o), and the \$< contains the name of the source file (main.c). (Actually, the \$< variable always contains the name of the *first* dependency, which in this case is main.c.) An automatic variable is a variable that make automatically sets for you, depending on the context. There are other automatic variables (we’ll come across one more shortly), but the two variables above are the most commonly used.

The -c flag tells the compiler to compile, but not link, the specified files. (As an aside, you may come across implicit rules of the form “.c.o:” which essentially do the same thing as the implicit rule above. The latter form is obsolete.)

Now, we can rewrite our all-target to make use of the implicit rule we just made, and also show off some of the powers of dependencies:

```
all: $(MY_OBJECTS)
$(CC) -o MyApp $^
```

First, note the use of a new automatic variable, \$^. This variable contains the names of all files that the target depends on. When make evaluates the all-target, it

will inspect the modification dates of all the listed .o files and compare them to their .c equivalents, and if necessary, remake the files that have changed by executing the implicit rule. Finally, when all dependencies have been satisfied, it will execute the rule specified by the all-target. This step consists of linking the object-files together to form an executable. In a real project, you might want to define some linker flags as well:

```
# The application needs the pthread and maths-libraries to work.
LDOPTS = -lpthread -lm
```

```
all: $(MY_OBJECTS)
    $(CC) $(LDOPTS) -o MyApp $^
```

Finally, you might have noticed that very often, you will create a number of C-files all including the same header file. You will want all of the C-files #including that header file to be re-compiled when you change the header file. To do this, you'll generally define a "depend" target. The depend target specifies how to generate this dependency information in a format that is suitable for make, and then you'll include the dependency info in the makefile:

```
depend: .depend
.depend:
$(CC) $(CFLAGS) -MM *.c > $@

-include .depend
```

The leading dash in front of the include-directive instructs make not to issue a warning if the specified file doesn't exist. If the file doesn't exist, make will look for a target to make the file (in this case, the .depend-target). The -MM flag passed to the compiler, instructs gcc to output dependency information for make. We redirect the output of the compiler to a separate file, called .depend. The contents of the .depend file will look similar to this:

```
main.o: main.h utils.h common.h
init.o: init.h utils.h common.h
utils.o: utils.h
```

If you don't redirect the preprocessor output to a separate file, make will append the information to your current makefile, and prepend the information with a comment stating that the information is used by make. Make knows how to keep this information up-to-date, so you should avoid manually editing the dependency-information.

That's about all you need to know to make some simple makefiles. There are however two more tricks that can be nice to know about. First, many libraries need special compiler flags to enable you to successfully use them (SDL is one such

library). Often they provide an application that outputs these flags. To include the output of the application in your CFLAGS variable, for instance, you can do the following (using SDL as an example here):

```
CFLAGS = -Wall -O2 `sdl-config --cflags`
```

Notice the use of the back-quotes. They instruct make to execute the command embedded between the quotes, and insert the output into the current context (such as a variable, as is done above). The second neat trick is instructing make to execute more than one job at a time. This is useful if you use an SMP machine for your development work, as more than one job at a time allows you to make more efficient use of the CPUs available (assuming the compiler isn't already SMP-aware). To instruct make to execute more than one command at a time, pass `-j<number of commands>` to make (make `-j2` or make `-j3` works well on a dual-processor machine). You can set up an alias in your `.bashrc` file to do this automatically whenever you type make:

```
alias 'make' = 'make -j2'
```

Make has other features apart from the ones mentioned here. You may want to consult the make man pages (“man make” or “info make”) for more in-depth information, however the information given here should serve as a good starting point. Finally, here is a sample makefile you can use as a base to create your own makefiles:

```
# Sample makefile
CC      = gcc
CFLAGS  = -Wall -O2
# Add linker flags and any libraries you use
# here (ie, LDOPTS = -lpthread -lm)
LDOPTS  =

OUTPUT  = MyApp
SRC      = main.c
OBJ      = $(SRC:.c=.o)

all: $(OBJ)
      $(CC) $(LDOPTS) -o $(OUTPUT) $^

clean:
      rm -f *.o
      rm -f $(OUTPUT)
distclean: clean
      rm -f *~
      rm -f *.bak
```

```
rm -f \#*

depend: .depend
.depend:
    $(CC) $(CFLAGS) -MM *.c > $@

%.o:%.c
    $(CC) $(CFLAGS) -c -o $@ $<

-include .depend
```