

# Managing Software With Makefiles

M. D. Jones, Ph.D.

Center for Computational Research  
University at Buffalo  
State University of New York

High Performance Computing I, 2013

# Background

Make is an old `UNIX` tool, along with `awk`, `sed`, `vi`, etc.

- Original by S. I. Feldman of Bell Labs (1975).
- If you program in a `UNIX` environment, you should be using Make
- Main idea: automate and optimize the building of programs
- GNU Make, or `gmake`, has added quite a lot of additional functionality, only some of which I will have time to cover here

# Motivation

The basic philosophy of **make**:

- You have a collection of (many) source files, some of which depend on other (header/module) files as well as each other, together which build a *target* (program)
- **make** is sensitive to these **dependencies**, and has default rules for common dependencies
- In choosing which targets to build, and how, **make** uses the names and timestamps of the files along with a set of rules

# The Description File

**make** is going to automatically look for a file called **Makefile** in the current working directory. Failing that, it will look for **makefile**, or you can pass it any file with the **-f** option. The contents of your Makefile:

**Comments** are lines that start with the **#** sign

**Macros** are simple name pairs, separated by an **=** sign, e.g.

```
CFLAGS = -O3 -ffast-math
```

You can use the **make -p** command to see a list of default macros. Can be overridden on invocation:

```
make "CFLAGS= -g"
```

**Continuation** of lines uses a **\** character

**Targets** are a particular goal, e.g.

```
my.x: my.c
      $(CC) $(CFLAGS) -o my.x my.c $(LIBS)
```

Note that the second line leads with a TAB (very significant). The general target syntax is:

```
target [target ...] : [dependent ...]
      [command ...]
```

**Special Macros** are predefined for easy use:

- \$@** Name of file to be made (target)
- \$?** Name(s) of changed dependents
- \$<** Name of file that triggered action
- \$\*** Prefix shared by target and dependent files

# Simple Target Examples

```
clean:
    -rm -f *.o *~ core.*
```

a very conventional target which will remove old stuff (the hyphen preceding the `rm` tells `make` to ignore any errors returned by that command),

```
CC = gcc
CFLAGS = -g -O
OBJS = main.o sub1.o sub2.o
LIBS = -L/my/proj/library -lmylib

my.x : ${OBJS}
    $(CC) $(CFLAGS) -o $@ ${OBJS} ${LIBS}

main.o : main.c
    $(CC) $(CFLAGS) -c -o $@ $?
sub1.o : sub1.c
    $(CC) $(CFLAGS) -c -o $@ $?
sub2.o : sub2.c
    $(CC) $(CFLAGS) -c -o $@ ?
```

# Suffix Rules

Suffix rules can be used to automate common compilation steps based on the suffix found on dependent files.

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

(actually that one is a default rule, but it's still a good example). Make considers all the suffixes defined this way to be significant, and will seek a default rule to apply.

# Defaults for Make

Note that you can print out the current set of defaults (it's a few hundred lines in most cases) by using the **-p** option to make, e.g.

```
make -p -f /dev/null
```

The “-f /dev/null” will just feed make a dummy (empty) Makefile.



# Some Handy Rules

Personally I like to declare my suffix rules explicitly - here are some handy ones (yes, I know that several are the defaults, at least for GNU make):

```
.c.o: $(CC) $(CFLAGS) -c $<
.cpp.o: $(CXX) $(CXXFLAGS) -c $<
.f.o: $(FC) $(FFLAGS) -c $<
.f90.o: $(F90) $(F90FLAGS) -c $<
```

and I tend to explicitly set the values for the compiler and flag macros.

# Make includes

Note that frequently used suffix rules or other customizations can be placed together into a single file and then loaded into multiple Makefiles using the **include** statement:

```
include file
```

# Make in Subdirectories

## GNU Make

### This is a GNU Make feature

Larger codes are often split into subdirectories in which each has its own Makefile. The macro `$(MAKE)` will pass the original flags to a secondary make process:

```
libsub.a libsubsub.a : force_look
    $(ECHO) looking into subdir : $(MAKE) -C subdir
    $(MAKE) -C subdir
force_look:
    true
```

# Debugging Makefiles

Debugging Makefiles can be simplified by using the **-d** option to make, which should cause make to display details of timing and execution.

# String Manipulation

## GNU Make

### This is a GNU Make feature

The GNU make utility has quite a bit of string handling capability - here we consider only the simplest form, best illustrated through an example:

```
SOURCES = source1.c source2.c source3.c  
OBJS = ${SOURCES:.c=.o}
```

generally most useful for handling patterns other than suffixes.

# Parallel Make

## GNU Make

This is a GNU Make feature

Note that you can (very) easily run make in parallel (multithreaded) using the **-j *njobs*** options, which causes make to try to run up to **njobs** simultaneously.

# Conditionals in Makefiles

## GNU Make

### This is a GNU Make feature

It is frequently helpful to place conditionals in Makefiles, e.g.

```
CFLAGS_icc = -fast -xN
CFLAGS_gcc = -ffast-math -O3 -fexpensive-optimizations

my.x: $(OBJECTS)
    ifeq ($(CC), icc)
        $(CC) $(CFLAGS_icc) -o my.x $(LIBS_icc)
    else
        $(CC) $(CFLAGS_gcc) -o my.x $(LIBS_gcc)
    endif
```

You can put as many branches in the conditional as you like, and note that `ifneq` is the negative version of `ifeq`, and `ifdef`/`ifndef` exist for testing if variables are empty.

# The `shell` Function

## GNU Make

### This is a GNU Make feature

You can communicate with the outside environment using the `shell` function (which is like using backticks in most scripting languages):

```
# list all c files in current directory
files := $(shell echo *.c)
# determine OS
OS := $(shell uname -s)
```

Typically these commands are evaluated using `/bin/sh`, but you can change the default shell if you wish (via the environmental `SHELL` variable).



# More GNU Make Stuff

GNU Make has many features - most of which are pretty safe to use now since **gmake** has become so widespread. It has almost come full circle to the point where GNU make has its own scripting capabilities. For more information, refer to the User's Guide:

<http://www.gnu.org/software/make/manual/make.html>

# Simple Makefile Example

```
1 C = gcc
2 CFLAGS = -g -m32 -O3
3 LDFLAGS = $(CFLAGS)
4 OBJS = fitp.o brent.o
5 OBJS_23 = fit23.o brent.o
6 .c.o:
7     $(CC) $(CFLAGS) -c $<
8
9 fitp: $(OBJS)
10     $(CC) -o $@ $(CFLAGS) $(OBJS) -L. -lnrsvd -lnr -lmdj -lm
11 fit23: $(OBJS_23)
12     $(CC) -o $@ $(CFLAGS) $(OBJS) -L. -lnrsvd -lnr -lmdj -lm
13
14 clean:
15     -rm -f *.o
```

# More Advanced GNU Make Example

```
1  # Collect info from environment (requires GNU make)
2  #
3  SHELL = /bin/sh
4  OS = $(shell uname -s)
5  MACH = $(shell uname -m)
6  PLATFORM = $(shell uname -p)
7  echo "SHELL = "$SHELL
8  echo "OS = "$OS
9  echo "MACH = "$MACH
10 echo "PLATFORM = "$PLATFORM
11 #
12 # the info function is only gnu Makefile >= 3.81
13 #
14 $(info SHELL=${SHELL})
15 $(info OS=${OS})
16 $(info MACH=${MACH})
17 $(info PLATFORM=${PLATFORM})
18
19 FC = dumb
```

```
20 ifeq (${OS}, Linux)
21     # Intel ifort
22     FC = ifort
23     FPPFLAGS = -fpp -D_HAVE_ETIME
24     FFLAGS = -O3 -g -traceback
25     LDFLAGS = -i-static
26     #FC = pgf90
27     #FPPFLAGS = -D_HAVE_ETIME
28     #FPPFLAGS = -Mpreprocess
29     #FFLAGS = -g -O
30     #LDFLAGS = -g77libs
31     LIB = -L/util/intel/cmk1/8.1/lib/32 -lsvml -lmkl_lapack -lmkl_ia32 -lguide -lpthread
32 endif
33 ifeq (${OS}, Darwin)
34     ifeq (${MACH}, i386)
35         # Intel ifort for Mac on Intel
36         FC = ifort
37         FPPFLAGS = -fpp -D_HAVE_ETIME
38         FFLAGS = -g -fast
39         LDFLAGS = -i-static
40         LIB = -L/Library/Frameworks/Intel_MKL.framework/Libraries/32 -lmkl_lapack \
41             -lmkl_ia32 -lguide -lpthread
42     else
43         # Absoft f95
44         FC = f95
45         FPPFLAGS =
46         # flag -m64 is for 64-bit code (requires G5 and Mac OS X 10.4 or later):
47         FFLAGS = -g -O2 -m64 -cons -IU77 -N11 -w -f free -YEXT_SFX='_' -YEXT_NAMES=LCS
48         LDFLAGS = -m64 -llapack -lf90math_altivec -lblas_altivec -lfio -lf77math -IU77
49     endif
50 endif
```

```

51 ifeq (${FC}, dumb)
52     $(info MACH=${MACH})
53     $(error Error, operating system/architecture unsupported. Fix Makefile)
54 endif
55 #
56 # generate general .o files from source (all suffixes are .f90)
57 #
58 SUFFIXES: .f90 .c
59 .f90.o:
60     $(FC) $(FFLAGS) -o $@ -c $<
61 .c.o:
62     $(CC) $(CFLAGS) -c $<
63 OBJ_mods = f2kcli.o parse.o modules.o lm.o global.o stopwatch.o
64 OBJ_gen = bigband.o bravaiss.o blockd.o dos.o eband.o epairpot.o \
65     eshift.o etot.o forcedbx.o getdat.o geomfact.o getopt.o gettbp.o \
66     HSupdate.o HSO.o HSO_ym.o isort.o kunif.o ntbls.o opt.o optband.o \
67     outtbp.o outqlm.o ptgrp.o readin.o scanv.o tb.o
68 OBJ_NRL = tbpdist_nrl.o H0_nrl.o H0_nrl2.o
69 OBJ_NN = tbpdist_nn2.o H0_nrl.o H0_nrl2.o
70 OBJ_XBMC = tbpdist_xbmc.o H0_xbmc.o
71 OBJ_LIB = dtimer.o va10a.o cpu_second.o
72 default: all
73
74 all : tb tb.nn tb.xbmc
75
76 tb: $(OBJ_mods) $(OBJ_gen) $(OBJ_LIB) $(OBJ_NRL) $(LIB_util)
77     $(FC) $(LDFLAGS) -o tb $(OBJ_mods) $(OBJ_gen) $(OBJ_LIB) $(OBJ_NRL) \
78     $(LIB) $(LIB_util)
79
80 tb.nn: $(OBJ_mods) $(OBJ_gen) $(OBJ_LIB) $(OBJ_NN) $(LIB_util)
81     $(FC) $(LDFLAGS) -o tb.nn $(OBJ_mods) $(OBJ_gen) $(OBJ_LIB) $(OBJ_NN) \
82     $(LIB) $(LIB_util)

```

```
84 tb.xbmc: $(OBJ_mods) $(OBJ_gen) $(OBJ_LIB) $(OBJ_XBMC) $(LIB_util)
85     $(FC) $(LDFLAGS) -o tb.xbmc $(OBJ_mods) $(OBJ_gen) $(OBJ_LIB) \
86     $(OBJ_XBMC) $(LIB) $(LIB_util)
87
88 clean:
89     -rm -f *.o *.a *.d *.mod *.il *.stb *.lst
90
91 # Individual Routines in lib
92 #
93 dtimer.o : lib/dtimer.f90
94     $(FC) $(FFLAGS) -c -o $@ $<
95 global.o : lib/global.f90
96     $(FC) $(FFLAGS) -c -o $@ $<
97 lm.o : lib/lm.f90
98     $(FC) $(FFLAGS) -c -o $@ $<
99 lmdif1.o: lib/lmdif1.f
100     $(FC) $(FFLAGS) -c lib/lmdif1.f
101 ...
102 ...
```

# Make Resources

- Stuart I. Feldman, “Make – A Program for Maintaining Computer Programs”, Bell Laboratories Computing Science Technical Report 57 (1978).
- Andrew Oram and Steve Talbott, “Managing Projects with Make,” 2nd Ed. (O’Reilly & Associates, Sebastapol, CA, 1991).
- Richard M. Stallman and Roland McGrath, “GNU Make: A Program for Directing Recompilation,” (Free Software Foundation Inc., 1993).