How to hand-make a makefile for a simple Fortran project

Here you can find a commented example of makefile, suitable to work with GNU make, for a simple Fortran project, it takes care of the most common situations generating non trivial dependencies, such as including files, calling subroutines in different source files, using modules.

This can be used as a tutorial for makefile-beginners who do not want to learn all the peculiarities and misteries of creating a makefile.

Makefile

```
# A simple hand-made makefile for a package including applications
# built from Fortran 90 sources, taking into account the usual
# dependency cases.

# This makefile works with the GNU make command, the one find on
# GNU/Linux systems and often called gmake on non-GNU systems, if you
# are using an old style make command, please see the file
# Makefile_oldstyle provided with the package.

# ======================================================================
# Let's start with the declarations
# ======================================================================

# The compiler
FC = gfortran
# flags for debugging or for maximum performance, comment as necessary
FCFLAGS = -g -fbounds-check
FCFLAGS = -O2
# flags forall (e.g. look for system .mod files, required in gfortran)
FCFLAGS += -I/usr/include

# libraries needed for linking, unused in the examples
#LDFLAGS = -li_need_this_lib

# List of executables to be built within the package
PROGRAMS = prog1 prog2 prog3 prog4

# "make" builds all
all: $(PROGRAMS)
```

```
# ======================================================================
# Here comes the most interesting part: the rules for prog1, prog2,
# prog3 and prog4, modify to suit your needs
# ======================================================================

# In order to understand the next section, the process of building an
# executable has to be clear: this is typically done in two steps:

# 1.Compilation: every source file required for our program (tipically
# x.f90 or x.F90 in case of Fortran) is compiled into an object file
# (usually x.o)
# 2.Linking: the final executable file is built by "linking" together
# all the object files compiled in the previous step; in this step
# additional pre-compiled libraries of can be added, but this will not
# be treated here.

# These two steps are often performed through the same command and can
# be combined into a single operation, so it is easy to confuse them,
# but, in order to understand what comes further, one has to keep in
# mind that they are logically different operations.

# A general suggestion: when building an executable called "prog", a
# good practice is to put the main program, i.e. the procedure
# declared as "PROGRAM" in Fortran, in a file named "prog.f90" or
# "prog.F90", after the name of the executable, this will simplify the
# creation of the rules in the makefile.
```

Example source files

prog1.f90

```
# The simplest case: prog1 is simply built from prog1.f90 through
# prog1.o; there is nothing to specify for prog1, because this simple
# situation is already handled by the general rules for building
# executables from Fortran sources.
```

```
PROGRAM prog1

PRINT*,'Hi, I am prog1'

END PROGRAM prog1
```

prog2.f90

```
# Including a file: prog2, as before, is simply built from prog2.f90
# through prog2.o; however prog2.o, besides depending on prog2.f90,
# depends also on prog2.incf through inclusion, if prog2.incf is newer
```

```
PROGRAM prog2
```

prog2.incf

```
INTEGER :: i = 8
```

```
# than prog2.o, prog2.o has to be rebuilt; this has to be specified in
# the makefile by means of a particular rule; it is not necessary to
# confirm the dependency of prog2.o on prog2.f90, as well as of prog2
# on prog2.o, since they are already handled by the general rules:
prog2.o: prog2.incf
```

```
INCLUDE 'prog2.incf'

PRINT*,'Hi, I am prog2, i=',i

END PROGRAM prog2
```

```
# Calling external procedures: prog3 requires to link in not just
# prog3.o, as usual, but also aux.o, which contains a subroutine
# called from within prog3.f90; moreover prog3 has to be rebuilt if
# aux.o is newer than prog3 itself; both tasks are accomplished by
# adding an explicit dependency for the executable on aux.o; as
# before, there is no need to confirm that prog3 depends on prog3.o,
# that prog3.o depends on prog3.f90 and that aux.o depends on aux.f90
# since these dependencies are already handled by the general rules:
prog3: aux.o
```

prog3.f90

```
PROGRAM prog3

PRINT*,'Hi, I am prog3'
CALL aux('prog3')

END PROGRAM prog3
```

aux.f90

```
SUBROUTINE aux(who)
CHARACTER(len=*) :: who

PRINT*,'Called by ',who

END SUBROUTINE aux
```

```
# Using Fortran MODULES: prog4.f90 USEs a Fortran module defined
# inside mod.f90, this is similar to the include case (prog2), but,
# since there is no standard naming convention for compiled module
# files in f90, the dependency is more easily built on the object
# files, because when mod.o is generated, one is sure that
# any_module_inside_mod.mod has been newly generated as well; mod.o
# must also be linked in when building the executable, so the
# dependency on mod.o is added also for prog4, as in the external
# procedure case (prog3):
prog4.o: mod.o
prog4: mod.o
```

prog4.f90

```
PROGRAM prog4
USE mod

PRINT*,'Hi, I am prog4, i=',i

END PROGRAM prog4
```

mod.f90

```
MODULE mod
INTEGER :: i=4
END MODULE mod
```

```
# Putting it all together: when an executable is built from many
# sources, ALL the object files have to be specified in the executable
# dependencies, not just those containing SUBROUTINEs, FUNCTIONs or
# MODULEs that are directly CALLed or USEd in the main program
# file. However among the dependencies of the main program object
# file, only those that are directly CALLed or USEd inside it have to
# be specified, the other object files have to be specified as
# dependencies of the relevant object files that require them and will
# be built by a chain rule.
```

```
# ================================================================
# And now the general rules, these should not require modification
# ================================================================

# General rule for building prog from prog.o; $^ (GNU extension) is
# used in order to list additional object files on which the
# executable depends
%: %.o
	$(FC) $(FCFLAGS) -o $@ $^ $(LDFLAGS)

# General rules for building prog.o from prog.f90 or prog.F90; $< is
# used in order to list only the first prerequisite (the source file)
# and not the additional prerequisites such as module or include files
%.o: %.f90
	$(FC) $(FCFLAGS) -c $<

%.o: %.F90
	$(FC) $(FCFLAGS) -c $<

# Utility targets
.PHONY: clean veryclean

clean:
	rm -f *.o *.mod *.MOD

veryclean: clean
	rm -f *~ $(PROGRAMS)
```

Download

You can download the makefile-fortran package with the Makefile and the example source code directly from this page.

If you are really interested, this is the full maintainer's version of the package

---

Support

If you have questions, doubts or suggestions concerning the makefile-fortran package, please address them to Davide Cesari (`dcesari <at> arpa <dot> emr <dot> it`).

License

The code has been highlihted with source-highlight, thanks to Lorenzo Bettini for the package.

Last Update 19/05/2009

Back to main page.