Appendix B

Makefiles

How to compile and run programs with Fortran 90/95

Assume you have installed in your machine a **gfortran (gnu) compiler**.
[other possible compilers are **g95** and **ifort**].
In its simplest form all you need to type is:

> *gfortran mycode.f90 -o mycode.x*

In the above you ask the compiler to take your f90 code, compile it
and give the executable the name: *"mycode.x"*.

Run it with the command: > *./mycode.x*

N.B.: if you do not specify the name of the executable the compiler
will generate, by default, one called "a.out".
In this case to run you will need to type: > *./a.out*

This section discusses the following topics:

- How to compile a program that uses modules
- How to design makefiles to work with modules
- How to use the *−I* and *+moddir* options to manage *.mod* files

Although **make** is very useful, it cannot do its job without the instructions given by us, the programmer. **make** instructions is stored in a text file. This file is usually called **makefile** and contains commands that must be processed by **make**. This file is normally named **makefile or Makefile**. As a convention, GNU programs named their makefile, Makefile, because it is easy to see (if you do "ls" then this file is usually always on the top of the

list). If you give it another name, just make sure you include **option -f** to make command in order to let it know that you use it. For example, if we have a makefile named *phys381*, then the command we use to instruct make to process that file is : **make -f** *phys381*.

Makefile Structure

```
target: dependencies
                command
                command
        ...
```

First let us start with a simple example:

```
######################################################################

# MACROS:

OBJ= test.o sub.o
COMPILER= ifc
LINKER= ifc

# EXPLICIT RULES:

# TARGET : DEPENDENCIES
test.x : ${OBJ}
# <TAB> COMMANDS
        ${LINKER} ${OBJ} -o test.x

clean :
        rm test.x *o

test.o: test.f90 test.h
        ${COMPILER} -c test.f90

sub.o: sub.f90 test.h
        ${COMPILER} -c sub.f90

######################################################################
```

This Makefile assumes that two Fortran-90 files exists in the same directory as the Makefile, *test.f90* and *sub.f90*. *test.f90* contains the main program, and *sub.f90* contains some subroutine called by the main program. in addition, an include (or *header*) file called *test.h* is included in both *test.f90* and *sub.f90*.

A Makefile is read by the program make, and it contains a set of rules for what make should do when asked to make a certain target.

The first line starts with a hash (#). Lines starting with a hash is a comment line, and is ignored by make.

The next three lines are macros, they are simply pairs of "something" = "something else", as in "LINKER= ifc". This is somewhat similar to setting variables in a program.

The rest of this Makefile are targets and rules for making these, as in "test.x : ${OBJ}". "test.x" is a target, the two lines starting with "test.x" is the rule for this target. If one type "make test.x" on the command line, then make will read the Makefile and look for the target test.x. After the colon is the files that test.x depends on. In this case this is ${OBJ}. If test.x is not up to date, the command on the following line will be executed. In this case the object files test.o and sub.o will be linked into an executable test.x using the LINKER specified above. If the object files are not up to date, i.e. if test.f90, sub.f90, or test.h are newer than the object files, make will automatically create the object files.

IT IS VERY IMPORTANT TO NOTE THAT THE LINE CONTAINING THE COMMAND HAS TO START WITH A TAB-CHARACTER, NOT A SET OF SPACES.

For a simple example like this there isn't much to gain by using a Makefile, but for a project with a hundred sourcefiles using a Makefile is crucial. It will speed up compilation time by only compiling newly changed files and also ensure that all files that have to be compiled will be compiled.

Now what it happens when **MODULES** come into the picture? Specifics below.

Consider, for example, a program that consists of three files: *main.f90*, *code.f90*, and *data.f90*. The main program unit is in *main.f90*, as follows.

main.f90

```
PROGRAM keep_stats
  ! stats_code contains module procedures for operating
  !   on statistical database
  USE stats_code
  INTEGER :: n
  ! print prompt, using nonadvancing I/O
  WRITE (*, FMT='(A)', ADVANCE='NO') 'Enter an integer '// &
                '(hint:  77 is current average):  '
  READ *, n
  IF (n == 0) THEN
    PRINT *, 'But not that one.'
  ELSE
    CALL update_db(n)
    IF (n >= get_avg()) THEN  ! get_avg is in stats_code
        PRINT *, 'Average or better.'
    ELSE
        PRINT *, 'Below average.'
    END IF
```

```
  END IF
END PROGRAM keep_stats
```

The first specification statement (USE) in the main program indicates that it uses the module *stats_code*. This module is defined in *code.f90*, as follows:

code.f90

```
! stats_code:  a (partial!) package of module procedures for
!   performing statistical operations
MODULE stats_code
  ! shared data to be used by procedures declared below
  USE stats_db
  CONTAINS  ! module procedures
        ! update_db:  updates shared variables in module stats_db
        SUBROUTINE update_db (new_item)
          INTEGER :: new_item
          n_items = n_items +1
          item(n_items) = new_item
          sum = sum + new_item
        END SUBROUTINE update_db
        ! get_avg:  returns arithmetic mean
        INTEGER FUNCTION get_avg ()
          get_avg = sum / n_items
        END FUNCTION get_avg
END MODULE stats_code
```

This program unit also begins with aUSE statement, which identifies the module it uses as *stats_db*. This module is defined in *data.f90*, as follows:

data.f90

```
! stats_db:  shared data declared here
MODULE stats_db
  INTEGER, PARAMETER :: size = 100  ! max number of items in
array
  ! n_items, sum, and item hold the data for statistical analysis
  INTEGER :: n_items, sum
  INTEGER, DIMENSION(size) :: item
  ! the initializations are just to start the program going
  DATA n_items, sum, item/3, 233, 97, 22, 114, 97*0/
END MODULE stats_db
```

The use of modules in this program creates dependencies between the files because a file that uses a module that is defined in another file is dependent on that other file. These dependencies affect the order in which the program files must be compiled. The

dependencies in the example program are:

- *main.f90* is dependent upon *code.f90*.
- *code.f90* is dependent upon *data.f90*.

These dependencies require that *data.f90* be compiled before *code.f90*, and that *code.f90* be compiled before *main.f90*. This order ensures that the compiler will have created each of the *.mod* files before it needs to read them. The order of the source files listed in the following command line ensures that they will compile and link successfully:

```
$f90 -o do_stats data.f90 code.f90 main.f90
```

During compilation, *f90* will create two *.mod* files, *STATS_CODE.mod* and *STATS_DB.mod*. These will be written to the current working directory, along with the object files and the executable program, *do_stats*. Following is a sample run of the executable program:

```
$do_stats
```

Enter an integer (hint: 77 is current average): 77 Average or better.

If instead of the preceding compile line, the program had been compiled as follows:

```
$f90 -o do_stats main.f90 data.f90 code.f90
```

the compilation would fail and *f90* would print the error message:

```
Error FCE37 : Module STATS_CODE not found
```

The compilation would fail because the compiler cannot process *main.f90* without *STATS_CODE.mod*. But the order in which the program files appear on the compile line prevents the compiler from processing *code.f90* (and thereby creating *STATS_CODE.mod*) until after it has processed *main.f90*.

B.1 Compiling with make

If you use the **make** utility to compile Fortran90 programs, the description file should take into account the dependencies created by modules. For example, to compile the *do_stats* program using the **make** utility, the description file should express the dependencies as follows:

makefile

```
# description for building do_stats
do_stats :      main.o code.o data.o
        f90 -o do_stats main.o code.o data.o
# main.o is dependent on main.f90 and code.f90
main.o :        main.f90 code.o
        f90 -c main.f90
```

```
# code.o is dependent on code.f90 and data.f90
code.o :        code.f90 data.o
        f90 -c code.f90
# data.o is dependent only its source, data.f90
data.o :        data.f90
        f90 -c data.f90

Note that the dependencies correspond to the order in which the source
files are specified in the following f90 compile line:

$f90 -o do_stats data.f90 code.f90 main.f90

Assuming that you name the description file ``{\bf makefile}", the command
line to compile the program with {\bf make} is simply:

{\bf > make}
```

Makefile Commands

```
ach of the dependencies are searched through all the targets available
nd executed if found.



e above is a target  a target called  clean.
t is useful to have such target if you want to have a fast  way to get rid
f all the object files and  executables. Simply type:

 make clean
```

B.2 Managing .mod files

By default, the compiler writes *.mod* files to the current working directory and looks there when it has to read them. The *+moddir=directory* and *-I directory* options enable you to specify different directories. The *+moddir* option causes the compiler to write *.mod* files indirectory, and the *-I* option causes the compiler to search *directory* for *.mod* files to read. (The space character between *-I* and *directory* is optional.)

Using the example of the *do_stats* program, the following command line compiles (without linking) *data.f90* and writes a *.mod* file to the subdirectory *mod_files*:

```
$f90 -c +moddir=mod_files data.f90
```

The command line:

```
$f90 -c +moddir=mod_files -I mod_files code.f90
```

uses both the *+moddir* and *-I* options, as follows:

- The *+moddir* option causes *f90* to write the *.mod* file for *code.f90* in the subdirectory *mod_files*.
- The *-I* option causes *f90* to look in the same subdirectory for the *.mod* file to read when compiling *code.f90*.

The command line:

```
$f90 -odo_stats -I mod_files main.f90 code.o data.o
```

causes *f90* to compile *main.f90*, look for the *.mod* file in the subdirectory *mod_files*, and link all of the object files into an executable program named *do_stats*.

For more information on Make and Makefiles, type "**man make**" on the command prompt.

B.3 Examples

Study the evolution of the makefile as it evolves from example 1 to example 4.

Question 39 *Makefiles*

Example 1

```
#
average : mainprog.o readit.o meanit.o printit.o
        gfortran -o average mainprog.o readit.o meanit.o printit.o

mainprog.o : mainprog.f90
        gfortran -c mainprog.f90

readit.o : readit.f90 unit.include data.include
        gfortran -c readit.f90

meanit.o : meanit.f90 data.include sum.include
        gfortran -c meanit.f90

printit.o : printit.f90 data.include sum.include
        gfortran -c printit.f90
```

Example 2

```
#
average : mainprog.o readit.o meanit.o printit.o
	gfortran -o $@ mainprog.o readit.o meanit.o printit.o

mainprog.o : mainprog.f90
	gfortran -c $<

readit.o : readit.f90 unit.include data.include
	gfortran -c $<

meanit.o : meanit.f90 data.include sum.include
	gfortran -c $*.f90

printit.o : printit.f90 data.include sum.include
	gfortran -c $*.f90
```

Example 3

```
#
OBJS = mainprog.o readit.o meanit.o printit.o
average : $(OBJS)
	gfortran -o $@ $(OBJS)

readit.o : readit.f unit.include data.include
	gfortran -c $<

meanit.o : meanit.f data.include sum.include
	gfortran -c $<

printit.o : printit.f data.include sum.include
	gfortran -c $<
```

Example 4

```
#
.f90.o:
	rm -f90 $@
	gfortran -c $*.f90
```

```
OBJS = mainprog.o readit.o meanit.o printit.o

average : $(OBJS)
        gfortran -o $@ $(OBJS)

readit.o : unit.include data.include
meanit.o : data.include sum.include
printit.o : data.include sum.include
```

Question 40 *The Clean Command*

Occasionally, you will want to remove a directory of all .o files and executables (and possibly emacs backup files). The reason? If you have written your makefile incorrectly, it's possible that there is a .o file that is not being updated when it should. By removing all such files, you force the makefile to recompile all .o files, thus guaranteeing the most recent rebuild. Of course, this is not something that should be necessary if you've created a makefile correctly. It defeats the purpose of creating a makefile in the first place! Nevertheless, if you find that the code isn't working the way it should, and you suspect it's due to a buggy makefile, you can run *make clean*. This is how it usually looks:

Clean

```
clean:
          rm *.o *~ p1
```

The backslash prevents "rm" from complaining if you remove the files (especially, if it's been aliased with the -i option). Normally, you remove all .o files, all files ending in (which are emacs backup files), and the name of the executable (or executables, as the case may be). In the example above, the executable is called p1.

Add a *clean* option to the Makefiles given above and run: *make clean*.

Question 41 *The Echo command*

Add the following lines to one of your makefiles. Run your makefile. What do you get?

Echo

```
help:                                                  '
          @echo '  * To include a figure in Ltaex, you must use
```

```
                                    "usepackagegraphicx" and  '
@echo '    "includegraphicsfilename.eps".                                  '
```