

Building Your Code and an Introduction to Makefiles

Command line compilation

- * **Example:** program **abc.f90**, and external subroutines **a.f90**, **b.f90** and **c.f90**. Let's compile these in the usual command-line method:

```
> f90 -o abc abc.f90 a.f90 b.f90 c.f90    (one fell swoop)
```

- * The compile step alone is done by specifying the **-c** flag in the compile command:

```
> f90 -c abc.f90 a.f90 b.f90 c.f90    (compile only and keep the .o files)
> f90 -o abc abc.o a.o b.o c.o    (link to create the executable)
```

- * Suppose we later modified **b.f90** and needed to recompile:

```
> f90 -c b.f90    (compile only the file b.f90)
> f90 -o abc abc.o a.o b.o c.o    (link to create the new executable)
```

Introduction to Makefiles

- * A makefile is just a set of rules to determine which pieces of a large program need to be recompiled, and issues commands to recompile them.
- * For large programs, it's usually convenient to keep each program unit in a separate file. Keeping all program units in a single file is impractical because a change to a single subroutine requires recompilation of the entire program, which can be time consuming.
- * When changes are made to some of the source files, only the updated files need to be recompiled, although all relevant files must be linked to create the new executable.

- * With such a small example like this, recompiling everything is not that time consuming. But with more and more program units and/or when optimization is used, the time savings can be substantial.

- * Basic makefile structure: a list of rules with the following format:

```
target ... : prerequisites ...  
<TAB>      construction-commands
```

- * A “**target**” is usually the name of a file that is generated by the program (e.g, executable or object files). It can also be the name of an action to carry out, like “clean”.

- * A “**prerequisite**” is a file that is used as input to create the target.

*** Here's a simple makefile for our "abc" example:**

```
# makefile : makes the ABC program
```

```
abc : a.o b.o c.o abc.o  
f90 -o abc abc.o a.o b.o c.o
```

```
abc.o : abc.f90  
f90 -O -c abc.f90
```

```
a.o : a.f90  
f90 -O -c a.f90
```

```
b.o : b.f90  
f90 -O -c b.f90
```

```
c.o : c.f90  
f90 -O -c c.f90
```

- * By default, the first target listed in the file (the executable **abc**) is the one that will be created when the **make** command is issued.
- * Since **abc** depends on the files **a.o**, **b.o** and **c.o**, all of the **.o** files must exist and be up-to-date. **make** will take care of checking for them and recreating them if necessary. **Let's give it a try!**
- * Makefiles can include comments delimited by hash marks (**#**). A backslash (****) can be used at the end of the line to continue a command to the next physical line.

How Does Make Work?

- * The make utility compares the modification time of the target file with the modification times of the prerequisite files. Any prerequisite file that has a more recent modification time than its target file forces the target file to be recreated.
- * By default, the first target file appearing in the makefile is the one that is built. Other targets are checked only if they are prerequisites for the initial target.
- * Other than the fact that the first target in the makefile is the default, the order of the targets does not matter. The make utility will build them in the order required.

More MAKE Functionality

- * By default, if you just type `make`, then the make utility looks for a file called `makefile` or `Makefile`. Use `make -f <mymakefile>` to explicitly define what file to use.
- * Use `make -n` to display which commands will be used to build the program but it will not actually execute them.
- * Use a "phony target" to code in a clean-up section.
- * Define variables for compiler type, compiler flags, list of all objects, etc. **Note that variables names in make are case-sensitive!**
- * example: `makefile2`

* Use a general pattern rule for the compilation steps:

```
# makefile3 : let's use some makefile variables
objects = a.o b.o c.o abc.o
Comp = /Applications/Absoft10.1/bin/f90      # Absoft compiler
#Comp = /usr/bin/fort                       # Intel compiler
FFLAGS = -g
#FFLAGS = -O

abc : $(objects)
    $(Comp) -o abc $(objects)

%.o : %.f90
    $(Comp) ${FFLAGS} -c $<
```

* Here we have made use of an **automatic variable**. **\$<** is the name of the first prerequisite.

More complexity

- * Handling c-preprocessor statements.
 - * most compilers automatically process .F90 and .F files
- * External libraries like netcdf.
- * Separate compile options for certain files.
- * Placing object files into archives.

Module compile/make issues

1. The order of compilation matters! If module B uses module A, then module A must be compiled first. This means that developing and maintaining dependency lists can be a bit cumbersome.
 - * There are some free scripts that try to handle this for you. **EXAMPLE:** `fmkmf.pl`
2. Most Fortran compilers generate module files in addition to object files.
3. Make is general-purpose: works great with C, but not so great with complex Fortran codes.

Foray: an alternative to make

- * Designed specifically to handle Fortran builds.
- * Simple to install, uses just one file (buildinfo).
- * Supports multiple interdependent targets
- * Supports multiple build configurations (debug, release, parallel, serial).
- * **Understands fortran dependencies and determines them automatically.**
- * Ability to set different compile options for different groups of files or individual files.
- * Uses archives in place of object files, and stores time stamps in a separate database.