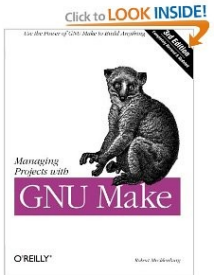
Technical University of Denmark

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

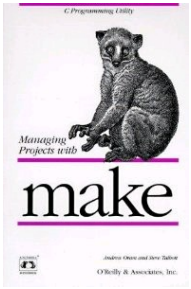
High Performance Computing

FORTRAN, OpenMP and MPI

41391



Makefile

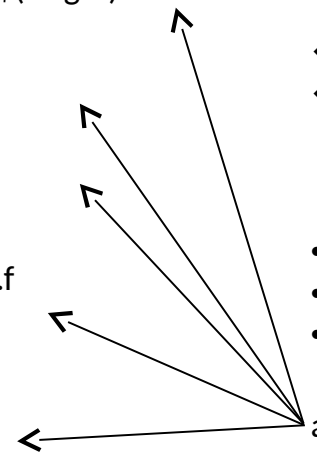


- Modification of source code (main program, subroutines, functions, modules) requires recompilation to ensure up-to-date object files and executables.
- *Make* is a tool that allows automatic compilation of source code:
  - *Make* monitors the time stamp of source code and re-compiles according to user defined dependencies.
  - *Make* is available on all platforms (incl. Windows).

Makefile

```
# macro defining the variable ‘target’
target = main.exe
# below a macro defining the list of object files in the project;
# the objects are recompiled in the order in which they appear in the objs list
# newline is “\” and must be THE last character on the line
# Tabs are not required/used in the definition of macros
objs = module.o \
       main.o \
       sub.o
# linking: the target depends on the objects
$(target): $(objs)
       f90 –free $(objs) –o $(target)
# dependencies:
main.o: main.f module.o
       f90 –free –c main.f
sub.o: sub.f
       f90 –free –c sub.f
module.o: module.f
       f90 –free –c module.f

new: clean $(target)
clean:
       rm –fr $(objs)
```



← Target depends on the objects; first target will be the default
← If objects are newer than the target – then re-link

← The object depends on the source and module
← If the source is newer than the object then recompile the source.

- Make is executed using the command: make/nmake/gmake
- Make will automatically use the file: ‘Makefile’ or ‘makefile’
- Use ‘make –f MyMakefile’ to choose a specific file.

all commands start with a tabulation!)

Makefile macro

- A macro definition is a line containing an equal sign.
  Example:
  ```
  E1 = XYZ
  E2 = “XYZ”
  ```
  E2 will contain the quotes.
- Tabs are not allowed before the macro name.
- Refer to a macro by: `${E1}` or `$(E1)`.
  Example:
  ```
  E3 = text.${E1}
  ```
  will produce: `text.XYZ`.
- If a macro is repeated, then the last will be selected.

Makefile macro

- Macros are insensitive to the order of which they are defined, thus:

```
SOURCE = ${MY_SRC} ${SHARED_SRC}
MY_SRC = parse.f search.f
SHARED_DIR = /home/jhw/src
SHARED_SRC = ${SHARED_DIR}/depend.f
```

  - Will always (irrespective of the order above) result in:

```
SOURCE = parse.f search.f /home/jhw/src/depend.f
```

- Macros must be defined before dependencies in which they occur.

Makefile macro

- Make has build in (standard) macros. Fx. `${CC}, ${FC}, ${LD}.`
- Common macros are:
  - LDFLAGS: options to the linker.
  - FFLAGS: FORTRAN compiler options.
  - CFLAGS: C compiler options.

Example:

```
FFLAGS = -O3 -free
${FC} ${FFLAGS} –c main.f
```

- The compiler options (fx. –O3) are NOT standard and depend on the specific compiler (fx. –free, -Mfreeform, -ffree-form).

Makefile macro

- Environment variables are available within Make. Example:

```
export DIR=/usr/proj # in the shell
make test            # in the shell
SRC = ${DIR}/src     # in Makefile
test:
   cd ${DIR}; ….
```

- Priority from least to greatest (`make –e` will change 2 and 3):

1. Internal (default) definition of Make.
2. Current shell variables.
3. Description file macro definitions.
4. Macros that you enter on the make command line.

   Example:

   `make “FFLAGS=–O3 -free” test`

Makefile macro string substitution

- Some/most Make support string substitution.

  Example:

  ```
  SRC = defs.f redraw.f calc.f
  ```

  Then the command (here replace all “.f” with “.o”)

  ```
  ls ${SRC:.f=.o}
  ```

  will produce the output:

  ```
  calc.o defs.o redraw.o
  ```

  assuming these files exist (otherwise `ls` will complain).

- String substitution is limited to taking place only at the end of the macro, or immediately before white space:

  ```
  LETTERS = xyz xyzabc abcxyz
  echo ${LETTERS:xyz=DEF}
  ```

  produces:

  ```
  DEF xyzabc abcDEF
  ```

Makefile internal macros

- The @ macro evaluates to the current target.

  Example:

```
plot_prompt: basic.o prompt.o
    f90 –free –o $@ basic.o prompt.o
```

  Here `$@` will produce: `plot_prompt.`

- The `$?` macro evaluates to a list of prerequisites that are newer that the current target.

  Example:

```
libops: interact.o sched.o gen.o
    ar rv $@ $?
```

  the `$?` will contain the `.o` files that are newer than the target `libops`.

Makefile suffix rules

- Make has build in (default) suffix rules for compiling FORTRAN, C, etc.:

```
.SUFFIXES : .o .f .c
.f.o :
    ${FC} ${FFLAGS} –c $<
.c.o :
    ${CC} ${CCFLAGS} –c $<
```

- Make checks the `.o` files to see if it can find a matching `.f` or `.c` file (the result is stored in $<) and if this source file is newer than the `.o` file, it recompiles the file using the above commands.

  Example:

```
OBJS   = main.o sub1.o sub2.o
FC     = f90 # overwrite default ${FC}
FFLAGS = -O3 –free # overwrite default ${FFLAGS}
program: ${OBJS}
    ${FC} –o $@ ${OBJS}
```

  Make will automatically compile the `main.f`, `sub1.f` and `sub2.f` using the `f90` compiler with the options `–free –O3` and links the executable. program also using the `f90` compiler.

Makefile dependencies

- However, Make cannot see from the object/source list, if one source/object depends on another source/object.
- To account for this, explicit dependencies should be added.

  Example:

  sub.o: sub.f module.o precision.inc

  ${FC} ${FFLAGS} –c ${@:.o=.f}

  here the object sub.o depends on sub.f (default) but also on the include file and on the object file from the module.f.

Compiling and using a library

- Creating a library:
  - `f90 –O3 –free –c src1.f src2.f`
  - `f90 –O3 –free –c src3.f`
  - `ar rv libsrc.a src1.o src2.o src3.o`
    - `rv: inserts/replaces members verbosely`
    - `ar tv libsrc.a lists the objects in the library`
    - `nm libsrc.a lists the routines in a library`
  - `mkdir –p ~/lib` (or `mkdir –p $HOME/lib`)
  - `cp libsrc.a ~/lib/.` (or `cp libsrc.a $HOME/lib/.`)
- Using a library:
  - `f90 –free –c tmp1.f tmp2.f`
  - `f90 –o myexe tmp1.o tmp2.o –L$(HOME)/lib -lsrc`
- The order of which libs are listed is significant.

Compiling with includes and modules

- Include and module files are searched in the path given by –I<path>
- The –I<path> is (normally) APPENDED to the default '.'

  Example: search '.' and './inc1/', and './inc2/'; the first occurrence is used!

```
f90 –free –Iinc1 –Iinc2 –c tmp1.f tmp2.f
```

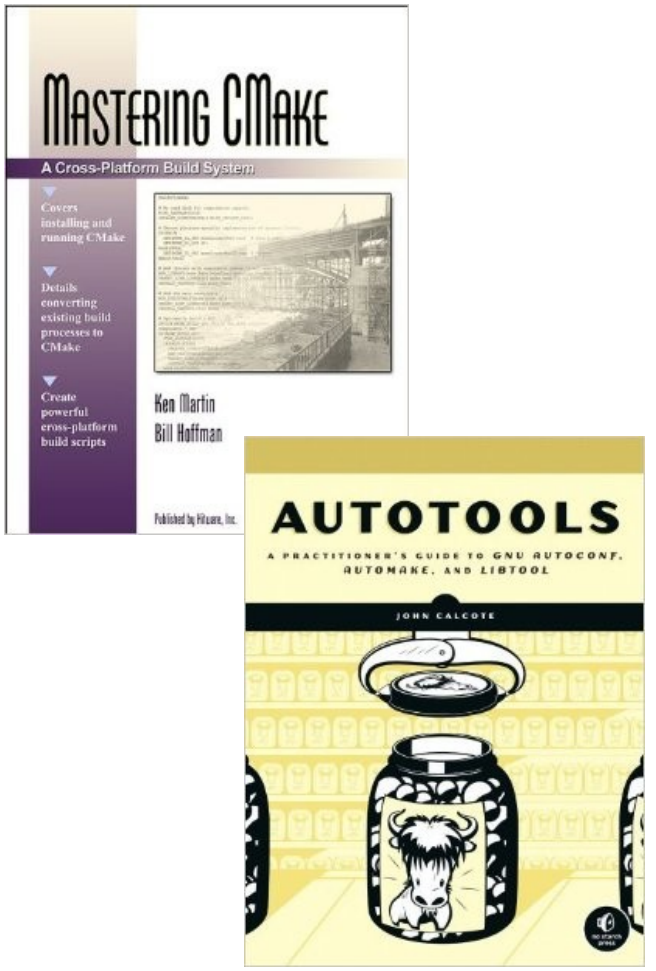
- Module files are produced by compilation:

  `f90 –free –c module.f`

  result in two (or more) files:
  - `module.o` : the object file.
  - `module_a.mod` (or `module_a.MOD`) a module file.
  - `module_b.mod(or module_b.MOD)` a module file.
  - …

Other tools

- Cmake:
  - Produces Makefiles.
- GNU build system/autotools:
  - GNU autoconf
  - GNU automake
  - Based on bourne shell scripts and m4
  - Produces Makefiles.
- Dependencies in FORTRAN can be difficult to resolve.



Compiling with includes and modules

- Modules should be compiled before they can be used.

  Example: if a routine in `tmp1.f` USEs a module contained in `tmp2.f`: then (1) will fail, whereas (2) and (3) will work:

```
1.f90 –free –c tmp1.f tmp2.f
2.f90 –free –c tmp2.f tmp1.f
3.f90 –free –c tmp2.f; f90 –free –c tmp1.f
```