10 MAKE AND MM5

10.1 make and MM5 (V2)

The use of make in the MM5 (V2) project is necessitated for a number of reasons.

10.1.1 Logical Subdivision of code

MM5 (V2) is written in FORTRAN and organized with the goal of providing a logical structure for code development and to encourage modular development of new options and subroutines. In addition, it is desired to supply the user/developer with some “pointers” as to the location of routines of particular interest. So the hope is to create something that appeared simple to the casual user but allowed more convenient access for the power user.

This structure is implemented implicitly by taking advantage of the Unix file system structure. Since directories are arranged as trees, the subroutines are subdivided into conceptual groups. The include directory contains the include files for the various subroutines. The domain, dynamics, fdda, physics, and memory directories contain the subroutines divided by function. The Run directory holds the main program source code. make is the glue that holds this complicated structure together. As you have seen, make executes commands by spawning shells. These shells can in turn run make in subdirectories. This ability to nest makefiles is very powerful, since it allows you to recursively build an entire directory.

10.1.2 Minimize Portability Concerns

Writing portable code involves not only following language standards, but creating a development structure that is equally standard. Every time code moves to a new machine you not only need to worry about your code but compilers, the operating system and the system environment, available libraries, and options for all of the above.

The answer to this problem is two-fold, namely, use only standard tools and minimize use of esoteric options. make is such a standard tool - you will find make on every working UNIX machine you encounter. While each vendor’s make may differ in significant way, they all support a core subset of functionality. This means that a basic makefile with no bells and whistles will work on the vast majority of machines available on the market.

"All makes are equal, but some makes are more equal than others."

Every decent Unix box will have make and cpp. However, they may throw in non-standard options. The quotation above reminds us to keep to standard constructions whenver possible.

10.1.3 Conditional Compilation

One of the stated goals is conditional compilation. This is done in two different ways. make keys off the user's options to skip compilation of those directories not required. When a source file is compiled, cpp is used to avoid including code that is not required. So make skips unnecessary compilation while cpp is used to modify compilation.

10.2 Configure.user File

Since make needs rules and defined dependencies (sometimes not the default ones), and there are more than 50 makefiles in the MM5 V2 directory structure, it would be an enomous task to make any change to these makefiles. A simple solution to solve this problem is to define all the rules and dependencies in one file and pass this file to all makefiles when make is executed. These definitions constitute part of the *configure.user* file. This section explains the rules and dependencies as defined in the *configure.user* file.

SHELL	Defines the shell under which the make is run.
.SUFFIXES	Defines the suffixes the makefiles use.
FC	Macro to define fortran compiler.
FCFLAGS	Macro to define any FORTRAN compiler options.
CFLAGS	Macro to define any c compiler options.
CPP	Macro to define where to locate c pre-processor on the machine.
CPPFLAGS	Macro to define any cpp options.
LDOPTIONS	Macro to define any loader options.
LOCAL_LIBRARIES	Macro to define any local libraries that the compiler may access.
MAKE	Macro to define the make command.
-I$(LIBINCLUDE)	to search for include files when compiling
-C	cpp option: all comments (except those found on cpp directive lines) are passed along.
-P	cpp option: preprocess the input without producing the line control information used by the next pass of the C compiler.
-i	make option: ignore error codes returned by invoked commands.
-r	make option: to remove any default suffix rules.
AR	Macro to define archive options.
RM	Macro to define remove options.

RM_CMD	Macro to define what to remove when RM is executed.
GREP	Macro similar to grep.
CC	Macro to define c compiler.

The following, which appears at the end of the *configure.user* file, defines the suffix rules a makefile uses. For example, .F.o: defines the rules to go from .F to .o files. In this case, the make will first remove any existing out-of-date .o file, and compile the .F files.

```
.F.i:
        $(RM) $@
        $(CPP) $(CPPFLAGS) $*.F > $@
        mv $*.i $(DEVTOP)/pick/$*.f
        cp $*.F $(DEVTOP)/pick
.c.o:
        $(RM) $@ && \
        $(CC) -c $(CFLAGS) $*.c
.F.o:
        $(RM) $@
        $(FC) -c $(FCFLAGS) $*.F
.F.f:
        $(RM) $@
        $(CPP) $(CPPFLAGS) $*.F > $@
.f.o:
        $(RM) $@
        $(FC) -c $(FCFLAGS) $*.f
```

10.3 Makefiles

make is a tool that executes "makefiles". Makefiles contain "targets" and "dependencies". A target is what you want to compile. A dependency is what needs to be done to compile the target. We use a 3-tiered makefile structure following the directory struture.

- Top-Level
- Middle (branching) Level
- Lowest (compilation) Level

Examples of each makefile follow.

- Top Level hides everything. The casual user edits the parameters and then just types "make". We take care of the rest.
- Middle Level is where branching occurs. These would be modified for something like the addition of a new moist physics scheme.
- Lowest Level is where object files are made. Change this when adding files. In addition, the power user will make in these lower directories to avoid remaking the whole structure.

10.3.1 Example: Top-Level Makefile

```
# Makefile for top directory

DEVTOP = .
include ./configure.user

all:
```

```
    (cd Util; $(MAKE)); \
    ./parseconfig; \
    (cd include; $(MAKE)); \
    (cd memory; $(MAKE)); \
    (cd fdda; $(MAKE)); \
    (cd domain; $(MAKE));\
    (cd physics; $(MAKE));\
    (cd dynamics; $(MAKE));\
    (cd Run; $(MAKE));
code:
    find . -name \*.i -exec rm {} \; ; \
    (cd Util; $(MAKE)); \
    ./parseconfig; \
    (cd include; $(MAKE)); \
    (cd include; $(MAKE) code); \
    (cd memory; $(MAKE) code); \
    (cd fdda; $(MAKE) code); \
    (cd domain; $(MAKE) code);\
    (cd physics; $(MAKE) code);\
    (cd dynamics; $(MAKE) code);\
    (cd Run; $(MAKE) code);
little_f:
    (cd Util; $(MAKE)); \
    ./parseconfig; \
    (cd include; $(MAKE)); \
    (cd memory; $(MAKE) little_f); \
    (cd fdda; $(MAKE) little_f); \
    (cd domain; $(MAKE) little_f); \
    (cd physics; $(MAKE) little_f); \
    (cd dynamics; $(MAKE) little_f); \
    (cd Run; $(MAKE) little_f);


mm5.deck:
    ./Util/makedeck.csh $(RUNTIME_SYSTEM);

clean:
    (cd Util; $(MAKE) clean); \
    (cd include; $(MAKE) clean); \
    (cd memory; $(MAKE) clean); \
    (cd fdda; $(MAKE) clean); \
    (cd physics; $(MAKE) clean);\
    (cd domain; $(MAKE) clean);\
    (cd dynamics; $(MAKE) clean);\
    (cd Run; $(MAKE) clean); \
    if [ -f libutil.a ]; then $(RM) libutil.a; fi;

rm_obj:
    (cd Util; $(MAKE) clean); \
    (cd include; $(MAKE) clean); \
    (cd memory; $(MAKE) clean); \
    (cd fdda; $(MAKE) clean); \
    (cd physics; $(MAKE) clean);\
    (cd domain; $(MAKE) clean);\
    (cd dynamics; $(MAKE) clean);\
    (cd Run; $(MAKE) rm_obj); \
    if [ -f libutil.a ]; then $(RM) libutil.a; fi;

LineNumberer:
    $(CC) -o ./LineNumberer Util/LineNumberer.c;

mmlif:
    (cd Run; $(MAKE) mmlif);

### Additions for MPP
#
```

```
#   To clean after changes to configure.user, type 'make mpclean'.
#   To uninstall everything relating to MPP option, 'make uninstall'.
#   To partially remake installation, remove MPP/mpp_install and 'make mpp'.
#

mpclean: clean
	(cd MPP/build ; /bin/rm -fr *.o *.f *.dm *.b)

mpp: MPP/mpp_install
	(cd Util; $(MAKE))
	./parseconfig
	(cd include; $(MAKE))
	(cd include; $(MAKE) code)
	(sed '/t touch anything below this line/,$$d' configure.user \
	            > ./MPP/conf.mpp)
	(cd MPP; $(MAKE) col_cutter)
	(cd MPP/build;                \
	/bin/rm -f .tmpobjs ; \
	            $(CPP) -I../../pick ../mpp_objects_all > .tmpobjs ; \
	            $(MAKE) -f Makefile.$(MPP_LAYER) )

MPP/mpp_install:
	(cd include; $(MAKE) code )
	(cd MPP/RSL/RSL ; $(MAKE) $(MPP_TARGET) )
	(cd MPP/FLIC ; $(MAKE) ; $(MAKE) clean )
	(cd MPP/FLIC/FLIC ; $(MAKE) ; \
	  $(MAKE) clean ; \
	  /bin/rm -f flic ; \
	      sed s+INSTALL_STRING_FLICDIR+`pwd`+ flic.csh > flic ; \
	  chmod +x flic )
	(csh MPP/Makelinks $(MPP_LAYER) $(MPP_TARGET) )
	touch MPP/mpp_install

uninstall:
	(cd include; $(MAKE) clean)
	(cd memory; $(MAKE) clean)
	(cd fdda; $(MAKE) clean)
	(cd physics; $(MAKE) clean)
	(cd domain; $(MAKE) clean)
	(cd dynamics; $(MAKE) clean)
	(cd Run; $(MAKE) clean)
	if [ -f libutil.a ]; then $(RM) libutil.a; fi
	(cd MPP/FLIC/FLIC; /bin/rm -f dm ; $(MAKE) clean )
	(cd MPP/FLIC; $(MAKE) clean ; /bin/rm -fr bin )
	(cd MPP/RSL/RSL; $(MAKE) clean ; /bin/rm -f librsl.a )
	/bin/rm -f MPP/FLIC/h/*.h
	/bin/rm -fr MPP/build
	/bin/rm -f parseconfig
	/bin/rm -f MPP/col_cutter
	/bin/rm -f Run/mm5.exe
	/bin/rm -f Run/mm5.mpp
	/bin/rm -f pick/*.incl *.h
	/bin/rm -f MPP/mpp_install
```

Note: there are several targets in the top-level makefile: *all, code, little_f* (for IBM xlf compiler, or any fortran compiler that does not allow the use of cpp), *mm5.deck, clean, LineNumberer,* and *mpclean, mpp, etc.* for MPP extension. If a user does not specify a target, the makefile will use the first one it sees. In this case, it is the '*all*' target. For any target that is not placed first, a user must explicitly specify the target. For example, we use 'make mm5.deck' to make a job deck. The command for the target '*all*' is to cd to a particular directory and execute make (the macro $(MAKE) is defined in *configure.user* file).

10.3.2 Example: Mid-Level Makefile

```
# Makefile for directory physics/pbl_sfc

DEVTOP = ../..
include ../../configure.user

# Makefile for directory physics/pbl_sfc

lib:
        @tmpfile='.tmpfile'; \
        echo $(IBLTYP) > $$tmpfile; \
        $(GREP) "0" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 0"; \
        (cd dry; $(MAKE) all); \
        else \
        echo "IBLTYP != 0"; \
        fi; \
        $(GREP) "1" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 1"; \
        (cd bulk; $(MAKE) all); \
        (cd dry; $(MAKE) all); \
        else \
        echo "IBLTYP != 1"; \
        fi; \
        $(GREP) "2" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 2"; \
        (cd hirpbl; $(MAKE) all); \
        else \
        echo "IBLTYP != 2"; \
        fi; \
        $(GREP) "3" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 3"; \
        (cd btpbl; $(MAKE) all); \
        else \
        echo "IBLTYP != 3"; \
        fi; \
        $(GREP) "5" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 5"; \
        (cd mrfpbl; $(MAKE) all); \
        else \
        echo "IBLTYP != 5"; \
        fi; \
        (cd util; $(MAKE) all);

code:
        @tmpfile='.tmpfile'; \
        echo $(IBLTYP) > $$tmpfile; \
        $(GREP) "1" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 1"; \
        (cd bulk; $(MAKE) code); \
        (cd dry; $(MAKE) code); \
        else \
        echo "IBLTYP != 1"; \
        fi; \
        $(GREP) "0" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 0"; \
        (cd dry; $(MAKE) code); \
```

```
        else \
        echo "IBLTYP != 0"; \
        fi; \
        $(GREP) "2" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 2"; \
        (cd hirpbl; $(MAKE) code); \
        else \
        echo "IBLTYP != 2"; \
        fi; \
        $(GREP) "3" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 3"; \
        (cd btpbl; $(MAKE) code); \
        else \
        echo "IBLTYP != 3"; \
        fi; \
        $(GREP) "5" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 5"; \
        (cd mrfpbl; $(MAKE) code); \
        else \
        echo "IBLTYP != 5"; \
        fi; \
        (cd util; $(MAKE) code);

little_f:
        @tmpfile='.tmpfile'; \
        echo $(IBLTYP) > $$tmpfile; \
        $(GREP) "0" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 0"; \
        (cd dry; $(MAKE) little_f); \
        else \
        echo "IBLTYP != 0"; \
        fi; \
        $(GREP) "1" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 1"; \
        (cd bulk; $(MAKE) little_f); \
        (cd dry; $(MAKE) little_f); \
        else \
        echo "IBLTYP != 1"; \
        fi; \
        $(GREP) "2" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 2"; \
        (cd hirpbl; $(MAKE) little_f); \
        else \
        echo "IBLTYP != 2"; \
        fi; \
        $(GREP) "3" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 3"; \
        (cd btpbl; $(MAKE) little_f); \
        else \
        echo "IBLTYP != 3"; \
        fi; \
        $(GREP) "5" $$tmpfile; \
        if [ $$? = 0 ]; then \
        echo "IBLTYP = 5"; \
        (cd mrfpbl; $(MAKE) little_f); \
        else \
        echo "IBLTYP != 5"; \
        fi; \
        (cd util; $(MAKE) little_f);
```

```
clean:
        (cd btpbl; $(MAKE) clean); \
        (cd bulk; $(MAKE) clean); \
        (cd dry; $(MAKE) clean); \
        (cd hirpbl; $(MAKE) clean); \
        (cd mrfpbl; $(MAKE) clean); \
        (cd util; $(MAKE) clean);
```

Note: This example shows how the branching is done with the mid-level makefile. The makefile first echos the string IBLTYP defined in *configure.user* file to a temporary file, .tmpfile. It then checks, using grep, to see if any of the options exist (in this case, IBLTYP may be 0,1,2,3, or 5). If any of them is defined, it will go to the directory that contains the subroutines for that option and execute the make command there. Again there are several targets in this mid-level makefile: *lib, code, little_f,* and *clean*. The default is the target *lib.*

10.3.3 Example: Low-Level Makefile

```
# Makefile for directory physics/pbl_sfc/mrfpbl

DEVTOP = ../../..
include ../../../configure.user

CURRENT_DIR = $(DEVTOP)/physics/pbl_sfc/mrfpbl

OBJS =\
        mrfpbl.o \
        tridi2.o

SRC =\
        mrfpbl.i \
        tridi2.i

SRCF =\
        mrfpbl.f \
        tridi2.f

LIBTARGET = util
TARGETDIR = ../../../

all:: $(OBJS)
        $(AR) $(TARGETDIR)lib$(LIBTARGET).a $(OBJS)

code:: $(SRC)

little_f:: $(SRCF) $(OBJS)
        $(AR) $(TARGETDIR)lib$(LIBTARGET).a $(OBJS)

# ----------------------------------------------------------------
# common rules for all Makefiles - do not edit

emptyrule::

clean::
        $(RM_CMD) "#"*

# ----------------------------------------------------------------
# DO NOT DELETE THIS LINE -- make depend depends on it.

mrfpbl.o: ../../../include/parame.inc ../../../include/rpstar.incl
mrfpbl.o: ../../../include/varia.incl ../../../include/dusolve1.incl
mrfpbl.o: ../../../include/param2.incl ../../../include/param3.incl
```

```
mrfpbl.o: ../../../include/pmoist.incl ../../../include/point3d.incl
mrfpbl.o: ../../../include/point2d.incl ../../../include/various.incl
mrfpbl.o: ../../../include/nonhyd.incl ../../../include/nhcnst.incl
mrfpbl.o: ../../../include/soil.incl ../../../include/soilcnst.incl
mrfpbl.o: ../../../include/addrcu.incl ../../../include/pbltb.incl
tridi2.o: ../../../include/parame.incl
```

Note: In this example, when make is executed (‘*make -i -r*’), it first looks for the target *all,* for example. It finds that the target *‘all’* depends on a group of object files (defined by the macro OBJS). The rules for making the object files are defined in *configure.user* file, i.e. the .F.o: rule. The makefile checks whether any .o files are out-of-date w.r.t. the .F files, or w.r.t. any of the include files used in the .F files. The dependencies on include files are at the end of the makefile. After the .o files are made, the command on the following line specifies how to archive them into *libutil.a* using macro AR defined in *configure.user.*

10.4 CPP

The cpp pre-processor is about as old as Unix itself. A pre-processor scans a file and make modifications according to user-supplied definitions. Typically this facility is used for global substitutions, conditional code inclusion, including files, and function templating. We only use the cpp "conditional code inclusion" and "including files" features. Because we use cpp, our Fortran codes are named .F, in contrast to .f. Many machines recognize .F files as the ones that need to be run through cpp first before being compiled.

10.4.1 CPP “inclusion”

One cpp directive is "#include <filename>". This directive indicates that filename should be included in the source prior to compilation.
Example:

```
      SUBROUTINE SOLVE3(IEXEC,INEST,NN)

# include <parame.incl>
```

turns into

```
      SUBROUTINE SOLVE3(IEXEC,INEST,NN)
C PARAME
C
C--- ADDITIONAL MEMORY REQUIREMENTS FOR FDDA RUNS (IFDDA=1),
C--- NONHYDROSTATIC RUNS(INHYD=1),HIGHER ORDER PBL RUNS (INAV=1),
C--- EXPLICIT MOISTURE SCHEME (IEXMS=1), ARAKAWA-SCHUBERT
C--- CONVECTIVE PARAMETERIZATION (IARASC=1), ATMOSPHERIC
C--- RADIATION (IRDDIM=1), MIXED-PHASE ICE SCHEME (IICE=1).
C--- GRAUPEL SCHEME (IICEG=1).
C
C
      PARAMETER (IFDDAG=0,IFDDAO=0,INHYD=1,INAV=0,INAV2=0,INAV3=0,
     1 IICE=0,IICEG=0,IEXMS=1,IKFFC=1,IARASC=0,IRDDIM=0,ISLDIM=1)
```

10.4.2 CPP “conditionals”

cpp also recognizes conditional directives. You define a macro in your source code using the "define" directive - you can then use the "#ifdef" test on this macro to selectively include code. Example: In *defines.incl*, there are statements such as:

```
#define IMPHYS4 1
#define IMPHYS1 1
#define ICUPA3 1
#define IBLT2 1
```

In the SOLVE3, the .F file has

```
C-  WHAT CUMULUS PARAMETERIZATION? 1. NONE
      IF(ICUPA(INEST).EQ.1)THEN
           CALL VADV(KZZ,QVTEN,QVA,QDOT,MSFX,PSA,SCR3,J,2,INEST)
C
C--- 2. ANTHES KUO
C
#ifdef ICUPA2
      ELSE IF(ICUPA(INEST).EQ.2)THEN
           CALL CUPARA2(J,INEST,QVTEN,TTEN)
#endif
#ifdef ICUPA3
C
C--- GRELL
C
       ELSEIF(ICUPA(INEST).EQ.3)THEN
           CALL VADV(KZZ,QVTEN,QVA,QDOT,MSFX,PSA,SCR3,J,INDQ,INEST
           CALL CUPARA3(TB,QVB,PSB,TTEN,QVTEN,RAINC,CLDFRA,J,HT,UB,VB,
     +           INEST)
#endif
```
.... and so on.

In this example ICUPA2 is not defined, so the CUPARA2 call is not included in the final source. Because ICUPA3 is defined (#define ICUPA3 1 in defines.incl), the call to CUPARA3 will be kept in the final source code.