3 MAKE UTILITY

3.1 The UNIX make Utility

The following chapter is designed to provide the MM5 user with both an overview of the UNIX make command and an understanding of how make is used within the MM5 (V2) system. UNIX supplies the user with a broad range of tools for maintaining and developing programs. Naturally, the user who is unaware of their existence, doesn't know how to use them, or thinks them unnecessary will probably not benefit from them. In the course of reading this chapter it is hoped that you not only become aware of make but that you will come to understand why you need it.

In the same way you use dbx when you want to "debug" a program or sort when you need to sort a file, so you use make when you want to "make" a program. While make is so general that it can be used in a variety of ways, its primary purpose is the generation and maintenance of programs. But why the bother of a separate make utility in the first place? When you wrote your first program it probably consisted of one file which you compiled with a command such as "f77 hello.f". As long as the number of files is minimal you could easily track the modified files and recompile any programs that depend on those files. If the number of files becomes larger you may have written a script that contains the compiler commands and reduces the amount of repetitive typing. But as the number of files increases further your script becomes complicated. Every time you run the script every file is recompiled even though you have only modified a single file. If you modify an include file it is your responsibility to make sure that the appropriate files are recompiled. Wouldn't it be nice to have something that would be smart enough to only recompile the files that need to be recompiled? To have something that would automatically recognize that a buried include file have been changed and recompile as necessary? To have something that would optimize the regeneration procedure by executing only the build steps that are required. Well, you do have that something. That something is make.

When you begin to work on larger projects, make ceases to be a nicety and becomes a necessity.

3.2 make Functionality

The most basic notion underlying make is that of *dependencies* between files. Consider the following command:

```
f77 -o average mainprog.o readit.o meanit.o printit.o
```

Consider the object file mainprog.o and its associated source code file mainprog.f. Since a change in mainprog.f necessitates recompiling mainprog.o, we say that mainprog.o is *dependent* upon mainprog.f. Using the same reasoning we see that the final program average is dependent upon mainprog.o. average in this context is the *target* program (the program we wish to build). In this fashion we can build up a tree of dependencies for the target, with each node having a subtree of its own dependencies (See Figure 3.1). Thus the target average is dependent upon both mainprog.o and mainprog.f, while mainprog.o is dependent only upon mainprog.f. Whenever mainprog.f is newer than mainprog.o, average will be recompiled.

make uses the date and time of last modification of a file to determine if the dependent file is newer than the target file. This is the time you see when using the UNIX ls command. By recognizing this time relationship between target and dependent, make can keep track of which files are *up-to-date* with one another. This is a reasonable approach since compilers produce files sequentially. The creation of the object file necessitates the pre-existence of the source code file. Whenever make finds that the proper time relationship between the files does not hold, make attempts to regenerate the target files by executing a user-specified list of commands, on the assumption that the commands will restore the proper time relationships between the source files and the files dependent upon them.

The make command allows the specification of a heirarchical tree of dependency relationships. Such relationships are a natural part of the structure of programs. Consider our program average (See Figure 3.1). This application consists of 6 include files and four source code files. Each of the four source code files must be recompiled to create the four object files, which in turn are used to create the final program average. There is a natural dependency relationship existing between each of the four types of files: include files, FORTRAN sources, object, and executables. make uses this relationship and a specification of the dependency rules between files to determine when a procedure (such a recompilation) is required. make relieves people who are constantly recompiling the same code of the tedium of keeping track of all the complexies of their project, while avoiding inefficiency by minimizing the number of steps required to rebuild the executable.

3.3 The Makefile

Even with a small number of files the dependency relationships between files in a programming project can be confusing to follow. In the case of mm5 (V2) with hundreds of files, an English description would be unuseable. Since make requires some definition of the dependencies, it requires that you prepare an auxillary file -- the makefile -- that describes the dependencies between files in the project.

There are two kinds of information that must be placed in a makefile: dependency relations and generation commands. The dependency relations are utilized to determine when a file must be regenerated from its supporting source files. The generation commands tell make how to build out-of-date files from the supporting source files. The makefile therefore contains two distinct line formats: one called *rules*, the other *commands*.

3.4 Sample make Syntax

targetfile :	**dependencies**
< tab >	**command1**
< tab >	**command2**
myprog.exe :	**mysource1.f mysource2.f**
< tab >	**f77 -o myprog.exe mysource1.f mysource2.f**

A rule begins in the first position of the line and has the following format:
targetfile: dependencies.

The name or names to the **left** of the colon are the names of target files. The names to the **right** of the colon are the files upon which our target is dependent. That is, if the files to the right are newer than the files to the left, the target file must be rebuilt.

A dependency rule may be followed by one or more command lines. A command line must begin with at least one tab character; otherwise, it will not be recognized by make and will probably cause make to fail. This is a common cause of problems for new users. Other than this, make places no restrictions are command lines - when make uses command lines to rebuild a target, it passes them to the shell to be executed. Thus any command acceptable to the shell is acceptable to make.

3.5 Macros

make Macro definitions and usage look very similar to UNIX environment variables and serve much the same purpose. If the Macro *STRING1* has been defined to have the value *STRING2*, then each occurrence of *$(STRING1)* is replaced with *STRING2*. The () are optional if *STRING1* is a single character.

MyFlags = -a -b -c -d

In this example every usage of $(MyFlags) would be replaced by make with the string "-a -b -c -d" before executing any shell command.

3.6 Internal Macros

$@	name of the current target
$<	The name of a dependency file, derived as if selected for use with an implicit rule.
$?	The list of dependencies that are newer than the target

$* The basename of the current target, derived as if selected for use with an implicit rule.

D directory path, $(@D), $(<D)

F file name, $(@F), $(<F)

3.7 Default Suffixes and Rules

.f.o:

< tab > $(FC) $(FFLAGS) -c $<

.f:

< tab > $(FC) $(FFLAGS) $(LDFLAGS) $< -o $@

.SUFFIXES:

< tab > .o .c .f

In the *configure.make/user* file you may notice a line beginning with .SUFFIXES near the top of the file and a number of targets defined at the bottom of the file (e.g., .f.o). In addition, you may notice that the MAKE macro is commonly defined using the -r option. These definitions are all designed to deal with what are known as make's implicit suffix rules.

An implicit suffix rule defines the relationship between files based on their suffixes. If no explicit rule exists and the suffix of the target is one recognized by make, it will use the command associated with the implicit suffix rule. So if there is no explicit rule in the Makefile which deals with the target mainprog.o, make will recognize the suffix .o to indicate that this is an object file and will look in its list of implicit suffix rules to decide how to update the target. If there is a file named mainprog.f in the directory, make will compile mainprog.o using the .f.o rule. If instead there is a file named mainprog.c, make will compile mainprog.o using the .c.o implicit suffix rule. If both source files are in the directory, the rule used is dependent on the particular implementation of make.

The -r option to make turns off the implicit suffix rules. So on most platforms we do not use the implicit suffix rules, preferring to define our own suffix rules. We do this by specifying which files with suffixes use suffix rules - this is done with the .SUFFIXES macro. We then define what these rules are at the bottom of the *configure.make/user* file. For example, one of the suffix rule we specify is

```
.F.o:
<tab>    $(RM) $@
<tab>    $(FC) -c $(FCFLAGS) $*.F
```

The reason we have this suffix rule is that all our Fortran files are named *.F, which will be subject to *cpp* (c pre-processor) before being compiled.

3.8 Sample Program Dependency Chart

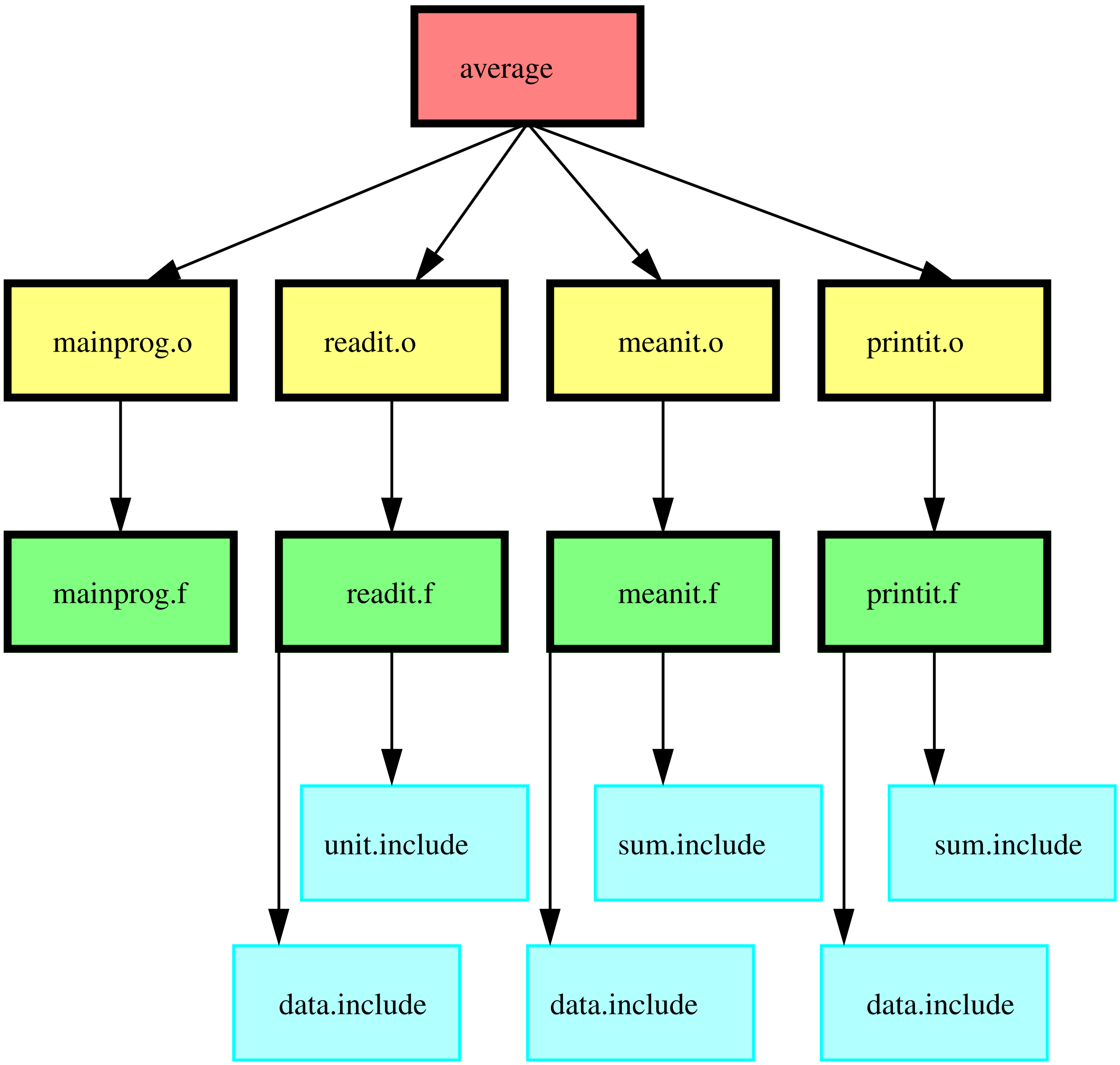

Fig. 10.1 Sample program dependency chart.

3.9 Sample Program Components for make Example

```
mainprog.f
-------------------------
program mainprog
call readit
call meanit
call printit
stop 99999
end
```

```
readit.f
-------------------------
 subroutine readit
 include 'unit.include'
 include 'data.include'
open (iunit,file='input.data',
*       access='sequential',
*       form='formatted')
 read (iunit,100) data
100    format(f10.4)
 close (iunit)
 return
 end
```

```
meanit.f
-------------------------
      subroutine meanit
      include 'data.include'
      include 'sum.include'
      do 100 l = 1, length
         sum = sum + data (l)
100   continue
      sum = sum / float(length)
      return
      end
```

```
printit.f
-------------------------
 subroutine printit
 include 'data.include'
 include 'sum.include'
print *,(l,data(l),l=1,length)
 print *,'average = ',sum
 return
 end
```

```
unit.include
-------------------------
parameter (iunit=7)
```

```
sum.include
-------------------------
common /avg/ sum
```

```
data.include
-------------------------
parameter (length=10)
common /space/ data(length)
```

3.10 makefile Examples for the Sample Program

```
#
#          first makefile example
#
average : mainprog.o readit.o meanit.o printit.o
          f77 -o average mainprog.o readit.o meanit.o printit.o

mainprog.o : mainprog.f
          f77 -c mainprog.f

readit.o : readit.f unit.include data.include
          f77 -c readit.f

meanit.o : meanit.f data.include sum.include
          f77 -c meanit.f

printit.o : printit.f data.include sum.include
          f77 -c printit.f
```

```
#
#          second makefile example
#
average : mainprog.o readit.o meanit.o printit.o
          f77 -o $@ mainprog.o readit.o meanit.o printit.o

mainprog.o : mainprog.f
          f77 -c $<

readit.o : readit.f unit.include data.include
          f77 -c $<

meanit.o : meanit.f data.include sum.include
          f77 -c $*.f

printit.o : printit.f data.include sum.include
          f77 -c $*.f
```

```
#
#          third makefile example
#
OBJS = mainprog.o readit.o meanit.o printit.o
average : $(OBJS)
          f77 -o $@ $(OBJS)

readit.o : readit.f unit.include data.include
          f77 -c $<

meanit.o : meanit.f data.include sum.include
          f77 -c $<

printit.o : printit.f data.include sum.include
          f77 -c $<
```

```
#
#          fourth makefile example
#
.f.o:
          rm -f $@
          f77 -c $*.f

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

average : $(OBJS)
          f77 -o $@ $(OBJS)

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

3.11 Configure.make File

The make rules, defined dependencies (sometimes not the default ones), and compiler/loader options are provided in a file called *configure.make/user* file. This section explains the rules and dependencies as defined in the *configure.make/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.make/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:
<tab>   $(RM) $@
<tab>   $(CPP) $(CPPFLAGS) $*.F > $@
<tab>   mv $*.i $(DEVTOP)/pick/$*.f
<tab>   cp $*.F $(DEVTOP)/pick
.F.o:
<tab>   $(RM) $@
<tab>   $(FC) -c $(FCFLAGS) $*.F
.F.f:
<tab>   $(RM) $@
<tab>   $(CPP) $(CPPFLAGS) $*.F > $@
```

3.12 An Example of configure.make File

```
# Sections
# 1. System Variables
# 2. User Variables
# 3. Fortran options
#    3a. Cray (YMP, J90)
# 4. General commands
#
#-----------------------------------------------------------------------------
# 1. System Variables
#-----------------------------------------------------------------------------
SHELL =         /bin/sh
.SUFFIXES: .F .i .o .f
#-----------------------------------------------------------------------------
# 2. User Variables
#-----------------------------------------------------------------------------
# RUNTIME_SYSTEM                  - Currently supported systems.
```

```
#                                   DEC
RUNTIME_SYSTEM = "DEC"
#
#---------------------------------------------------------------------------
# 3. Fortran options
#---------------------------------------------------------------------------
LIBINCLUDE = .
#---------------------------------------------------------------------------
#    3e. DEC
#---------------------------------------------------------------------------
FC = f77
FCFLAGS = -I$(LIBINCLUDE) -D$(NCARGRAPHICS) -convert big_endian
CFLAGS =
CPP = /opt/lib/cpp
CPPFLAGS = -I$(LIBINCLUDE) -D$(NCARGRAPHICS) -C -P
LDOPTIONS =
LOCAL_LIBRARIES = -L/usr/local/ncarg/lib -L/usr/local/lib  \
-lncarg -lncarg_gks -lncarg_c -lX11 -lm
MAKE = make -i -r
#---------------------------------------------------------------------------
# 4. General commands
#---------------------------------------------------------------------------
AR = ar ru
RM = rm -f
RM_CMD = $(RM) *.CKP *.ln *.BAK *.bak *.o *.i core errs ,* *~ *.a \
.emacs_* tags TAGS make.log MakeOut *.f
GREP = grep -s
CC = cc
#---------------------------------------------------------------------------
# Don't touch anything below this line
#---------------------------------------------------------------------------
.F.i:
        $(RM) $@
        $(CPP) $(CPPFLAGS) $*.F > $@
        mv $*.i $*.f
.F.o:
        $(RM) $@
        $(FC) -c $(FCFLAGS) $*.F
.F.f:
        $(RM) $@
       $(CPP) $(CPPFLAGS) $*.F > $@
```

3.13 An Example of Top-level Makefile

```
# Makefile for top directory ("all")
DEVTOP = .
include ./configure.make

all:
        (cd src; $(MAKE));
code:
        find . -name \*.i -exec rm {} \; ; \
        (cd src; $(MAKE) code);
little_f:
        (cd src; $(MAKE) little_f);
clean:
        (cd src; $(MAKE) clean);
```

3.14 An Example of Low-level Makefile

```
# Program Rawins
```

```
# Makefile for directory src
#
DEVTOP = ..
include ../configure.make

OBJS=   rawins.o adpblk.o analmn.o barb.o barnes.o blend.o bogpts.o buddy.o \
            decomdat.o dischk.o dots.o elim.o filslb.o flt2int.o geterr.o \
            getraw.o getuniob.o hedrin.o hedrot.o hrzfil.o idraw.o inacct.o \
            inhrz1.o inhrz2.o int2fl.o intr.o ipint.o lltoxy.o logcmp.o \
            m12n12.o manadp.o mqd.o mxmnll.o nestll.o newplv.o ntens.o \
            outfda.o outpt.o outtap.o pad.o plgrid.o plotab.o plotdom.o \
            plsond.o probgs.o probgu.o prosfc.o proupr.o psfc.o putslab.o \
            rdadp.o rdisk.o rtape.o savfil.o savstn.o seaprs.o setana.o \
            setup.o setups.o sfcadp.o sfcbln.o sfclup.o sfmerg.o sfscvt.o \
            sfuaob.o sigadp.o sigdat.o slbfil.o smther.o sort.o unisf.o \
            uniup.o vapres.o vtran.o wdisk.o windspd.o wndbarb.o wppadp.o \
            wtape.o wzzadp.o skewtsubs.o geth_newdate.o plots.o dadlib.o \
            sgemv.o dgeco.o swap.o cio.o gbytesys.o

SRC =   rawins.i adpblk.i analmn.i barb.i barnes.i blend.i bogpts.i buddy.i \
            decomdat.i dischk.i dots.i elim.i filslb.i flt2int.i geterr.i \
            getraw.i getuniob.i hedrin.i hedrot.i hrzfil.i idraw.i inacct.i \
            inhrz1.i inhrz2.i int2fl.i intr.i ipint.i lltoxy.i logcmp.i \
            m12n12.i manadp.i mqd.i mxmnll.i nestll.i newplv.i ntens.i \
            outfda.i outpt.i outtap.i pad.i plgrid.i plotab.i plotdom.i \
            plsond.i probgs.i probgu.i prosfc.i proupr.i psfc.i putslab.i \
            rdadp.i rdisk.i rtape.i savfil.i savstn.i seaprs.i setana.i \
            setup.i setups.i sfcadp.i sfcbln.i sfclup.i sfmerg.i sfscvt.i \
            sfuaob.i sigadp.i sigdat.i slbfil.i smther.i sort.i unisf.i \
            uniup.i vapres.i vtran.i wdisk.i windspd.i wndbarb.i wppadp.i \
            wtape.i wzzadp.i skewtsubs.i geth_newdate.i plots.i dadlib.i \
            sgemv.i dgeco.i swap.i cio.i gbytesys.i

rawins.exe:: $(OBJS)
            $(RM) $@
            $(FC) -o $@ $(OBJS) $(LDOPTIONS) $(LOCAL_LIBRARIES) $(LDLIBS)

code:: $(SRC)
# ----------------------------------------------------------------------
# common rules for all Makefiles - do not edit
#
clean::
            $(RM_CMD)
# ----------------------------------------------------------------------
# DO NOT DELETE THIS LINE -- make depend depends on it.
#
adpblk.o:    comadp.incl
analmn.o: paramirb.incl paramdim.incl paramirs.incl comd.incl comwt.incl
barb.o: compts.incl
blend.o: paramirb.incl paramirs.incl coma.incl comc.incl comd.incl comwt.incl
bogpts.o: paramirb.incl paramirs.incl coma.incl comb.incl
buddy.o: sbm.incl paramirs.incl
filslb.o: coma.incl
geterr.o: paramirb.incl paramirs.incl sbm.incl comobs.incl comksca.incl
geterr.o: comksfc.incl comd.incl comwt.incl
getraw.o: paramirb.incl paramirs.incl coma.incl comb.incl comc.incl comd.incl
```

```
getraw.o: comadp.incl comksfc.incl
getuniob.o: comscv.incl comuni.incl comrbt.incl
hedrin.o: hedmif.incl
hedrot.o: hedmif.incl
hrzfil.o: paramirb.incl paramirs.incl coma.incl comc.incl
idraw.o: compts.incl
inacct.o: paramirb.incl paramirs.incl comc.incl
inhrz1.o: hedmif.incl coma.incl comd.incl
inhrz2.o: coma.incl hedmif.incl paramdim.incl
ipint.o: compts.incl
lltoxy.o: comllxy.incl
logcmp.o: comadp.incl
manadp.o: comadp.incl
newplv.o: coma.incl comd.incl
ntens.o: comadp.incl
outfda.o: paramirb.incl paramirs.incl coma.incl comc.incl comd.incl
outfda.o: hedmif.incl
outtap.o: coma.incl comd.incl hedmif.incl
plgrid.o: comllxy.incl commap.incl
plotab.o: paramdim.incl coma.incl comb.incl hedmif.incl commap.incl
probgs.o: paramirb.incl paramirs.incl coma.incl comb.incl comc.incl
probgu.o: paramirb.incl paramirs.incl coma.incl comb.incl comc.incl
prosfc.o: paramirb.incl paramirs.incl coma.incl comb.incl comc.incl
proupr.o: paramirb.incl paramirs.incl coma.incl comb.incl comc.incl
psfc.o: paramirb.incl paramirs.incl coma.incl comc.incl comd.incl
rawins.o: paramdim.incl paramirb.incl paramirs.incl paramcr.incl comdata.incl
rawins.o: comobs.incl comksfc.incl coma.incl comb.incl comc.incl comd.incl skwdrw.incl
rawins.o: memcor.incl hedmif.incl comwt.incl sbm.incl
rawins.f: hedmif.incl comadp.incl
rdadp.o: comb.incl comadp.incl
rdisk.o: paramirb.incl paramdim.incl memcor.incl
rtape.o: paramirb.incl paramirs.incl coma.incl comc.incl
savfil.o: paramirb.incl paramirs.incl coma.incl comc.incl
savstn.o: paramirb.incl paramirs.incl coma.incl comb.incl comc.incl
savstn.o: comllxy.incl
seaprs.o: paramirb.incl paramirs.incl coma.incl comc.incl comwt.incl
setana.o: paramirb.incl paramirs.incl comksca.incl comd.incl comwt.incl
setup.o: paramirb.incl paramirs.incl paramdim.incl coma.incl comb.incl
setup.o: comc.incl comd.incl comllxy.incl memcor.incl comwt.incl hedmif.incl
setups.o: paramirb.incl paramirs.incl coma.incl comb.incl comc.incl
sfcadp.o: comadp.incl
sfcbln.o: paramirb.incl paramirs.incl coma.incl comc.incl comd.incl
sfcbln.o: comwt.incl
sfclup.o: comuni.incl
sfmerg.o: comuni.incl
sfscvt.o: netcdf.incl comscv.incl
sfuaob.o: netcdf.incl comuni.incl comrbt.incl
sigadp.o: comadp.incl
sigdat.o: paramirb.incl paramirs.incl coma.incl comc.incl comd.incl comb.incl
skwdrw.incl
slbfil.o: coma.incl
wdisk.o: paramdim.incl paramirb.incl paramcr.incl memcor.incl
wppadp.o: comadp.incl mulent.incl
wtape.o: paramirb.incl paramirs.incl coma.incl comc.incl
wzzadp.o: mulent.incl
skewtsubs.o:  skwdrw.incl
dadlib.o:     comadp.incl
```