An Augmented Version of MAKE

E. G. Bradford

Bell Laboratories
Whippany, New Jersey 07981

ABSTRACT

This paper describes an augmented version of the UNIX† *make* command. With one debatable exception, this version is completely upward compatible with the old version. This paper describes and gives examples only of additional features. The reader is assumed to have read the original *make* paper by S. I. Feldman.[1] Further possible developments for *make* are also discussed.

1. INTRODUCTION

This paper describes in some detail an augmented version of the *make* program. Some justification will be given for the chosen implementation and examples will demonstrate the additional features.

2. MOTIVATION FOR THE CURRENT IMPLEMENTATION

The *make* program was originally written for personal use by S. I. Feldman. However, it became popular on the BTL Research UNIX machine and a more formal version was built and installed for general use. Further developments of *make* have not been necessary in the BTL Research environment, and thus have not been done.

Elsewhere, *make* was perceived as an excellent program administrative tool and has been used extensively in at least one project for over two years. However, *make* had many shortcomings: handling of libraries was tedious; handling of the SCCS[2] file-name format was difficult or impossible; environment variables were completely ignored by *make*; and the general lack of ability to maintain files in a remote directory. These shortcomings hindered large scale use of *make* as a program support tool.

Make has been modified to handle the problems above. The additional features are within the original syntactic framework of *make* and few if any new syntactical entities have been introduced. A notable exception is the *include* file capability. Further, most of the additions result in a "Don't know how to make ..." message from the old version of *make*.

3. THE ADDITIONAL FEATURES

The following paragraphs describe with examples the additional features of the *make* program. In general, the examples are taken from existing *makefiles*. Also, the appendices are working *makefiles*.

3.1 The Environment Variables

Environment variables are read and added to the macro definitions each time *make* executes. Precedence is a prime consideration in doing this properly. For example, if the environment variable CC is set to **occ**, does it override the command line? Does it override the definition in the makefile? The following describes *make*'s interaction with the environment.

---

† UNIX is a trademark of Bell Laboratories.

1. *MAKE—A Program for Maintaining Computer Programs* by S. I. Feldman.
2. *Source Code Control System User's Guide* by L. E. Bonanni and C. A. Salemi.

A new macro, MAKEFLAGS is maintained by *make*. It is defined as the collection of all input flag arguments into a string (without minus signs). It is exported, and thus accessible to further invocations of *make*. Command line flags and assignments in the *makefile* update MAKEFLAGS. Thus, to describe how the environment interacts with *make*, we also need to consider the MAKEFLAGS macro (environment variable).

When executed, *make* assigns macro definitions in the following order:

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal *make* variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the **−f**, **−p**, and **−r** flags.)
2. Read and set the input flags from the command line. The command line adds to the previous settings from the MAKEFLAGS environment variable.
3. Read macro definitions from the command line. These are made *not resettable*. Thus any further assignments to these names are ignored.
4. Read the internal list of macro definitions. These are found in the file *rules.c* of the source for *make*. (See Appendix A for the complete makefile which represents the internally defined macros and rules.) They give default definitions for the C compiler (CC=cc), the assembler (AS=as), etc.
5. Read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense). Note, MAKEFLAGS will be read and set again. However, because it is not an internally defined variable (in *rules.c*), this has the effect of doing the same assignment twice. The exception to this is when MAKEFLAGS is assigned on the command line. (The reason it was read previously, was to be able to turn the debug flag on before anything else was done.)
6. Read the *makefile*(s). The assignments in the *makefile(s)* will override the environment. This order was chosen so when one reads a makefile and executes *make* one knows what to expect. That is, one gets what one sees unless the **−e** flag is used. The **−e** is an additional command-line flag that tells *make* to have the environment override the *makefile* assignments. Thus if **make −e** ... is typed, the variables in the environment override the definitions in the *makefile*. (Note, there is no way to override the command line assignments.) Also note that if MAKEFLAGS is assigned it will override the environment. (This would be useful for further invocations of *make* from the current *makefile*.)

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, we have:

1. Internal definitions (from *rules.c*).
2. Environment.
3. *Makefile(s)*.
4. Command line.

The **−e** flag has the effect of changing the order to:

1. Internal definitions (from *rules.c*).
2. *Makefile(s)*.
3. Environment.
4. Command line.

This ordering is general enough to allow a programmer to define a *makefile* or set of *makefiles* whose parameters are dynamically definable.

3.2 Recursive Makefiles

Another feature was added to *make* concerning the environment and recursive invocations. If the sequence **$(MAKE)** appears anywhere in a shell command line, the line will be executed even if the **−n** flag is set. Because the **−n** flag is exported across invocations of *make*,

(through the MAKEFLAGS variable) the only thing which will actually get executed is the *make* command itself. This feature is useful when a hierarchy of *makefile(s)* describes a set of software subsystems. For testing purposes, **make −n** ... can be executed and everything that would have been done will get printed out; including output from lower level invocations of *make*.

3.3 Format of Shell Commands within Make

Make remembers embedded new-lines and tabs in shell command sequences. Thus, if the programmer puts a *for* loop in the makefile with indentation, when *make* prints it out, it retains the indentation and backslashes. The output can still be piped to the shell and is readable. This is obviously a cosmetic change; no new functionality is gained.

3.4 Archive Libraries

Make has an improved interface to archive libraries. Due to a lack of documentation, most people are probably not aware of the current syntax for addressing members of archive libraries. The previous version of *make* allows a user to name a member of a library in the following manner:

```
lib(object.o)
```

or:

```
lib((_localtime))
```

where the second method actually refers to an entry point of an object file within the library. (*Make* looks through the library, locates the entry point and translates it to the correct object file name.)

To use this procedure to maintain an archive library, the following type of *makefile* is required:

```
lib::   lib(ctime.o)
        $(CC) −c −O ctime.c
        ar rv lib ctime.o
        rm ctime.o
lib::   lib(fopen.o)
        $(CC) −c −O fopen.c
        ar rv lib fopen.o
        rm fopen.o
... and so on for each object ...
```

This is tedious and error prone. Obviously, the command sequences for adding a C file to a library are the same for each invocation, the file name being the only difference each time. (This is true in most cases.) Similarly for assembler and YACC and LEX files.

The current version gives the user access to a rule for building libraries. The *handle* for the rule is the **.a** suffix. Thus a **.c.a** rule is the rule for compiling a C source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.s.a** and the **.l.a** rules rebuild YACC, assembler, and LEX files, respectively. The current archive rules defined internally are **.c.a**, **.c~.a**, and **.s~.a**. (The tilde (~) syntax will be described shortly.) The user may define in his makefile any other rules he may need.

The above two-member library is then maintained with the following shorter makefile:

```
lib:    lib(ctime.o) lib(fopen.o)
        @echo lib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rules is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

Thus, the **$@** macro is the **.a** target (**lib**) and the **$<** and **$*** macros are set to the out-of-date C file and the file name without suffix, respectively (*ctime.c* and *ctime*). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what *make* does when it sees the construction:

```
lib:    lib(ctime.o)
        @echo lib up-to-date
```

Assume the object in the library is out-of-date with respect to *ctime.c*. Also, there is no *ctime.o* file:

1. Do *lib*.
2. To do *lib*, do each dependent of *lib*.
3. Do *lib*(*ctime.o*).
4. To do *lib*(*ctime.o*), do each dependent of *lib*(*ctime.o*). (There are none.)
5. Use internal rules to try to build *lib*(*ctime.o*). (There is no explicit rule.) Note that *lib*(*ctime.o*) has a parenthesis in the name so identify the target suffix as **.a**. (This is the key. There is no explicit **.a** at the end of the *lib* library name. The parenthesis forces the **.a** suffix.) In this sense, the **.a** is hard-wired into *make*.
6. Break the name *lib*(*ctime.o*) up into *lib* and *ctime.o*. Define two macros, **$@** (=*lib*) and **$*** (=*ctime*).
7. Look for a rule **.X.a** and a file **$*.X**. The first **.X** (in the .SUFFIXES list) which fulfills these conditions is **.c** so the rule is **.c.a** and the file is *ctime.c*. Set **$<** to be *ctime.c* and execute the rule. (In fact, *make* must then do *ctime.c*. However, the search of the current directory yields no other candidates, whence, the search ends.)
8. The library has been updated. Do the rule associated with the **lib:** dependency; namely:

```
        echo lib up-to-date
```

It should be noted that to let *ctime.o* have dependencies, the following syntax is required:

```
lib(ctime.o):   $(INCDIR)/stdio.h
```

Thus, explicit references to **.o** files are unnecessary. There is also a new macro for referencing the archive member name when this form is used. **$%** is evaluated each time **$@** is evaluated. If there is no current archive member, **$%** is null. If an archive member exists, then **$%** evaluates to the expression between the parentheses.

An example *makefile* for a larger library is given in Appendix B. The reader will note also that there are no lingering ***.o** files left around. The result is a library maintained directly from the source files (or more generally, from the SCCS files).

3.5 SCCS File Names: The Tilde

The syntax of *make* does not directly permit referencing of prefixes. For most types of files on UNIX machines this is acceptable, because nearly everyone uses a suffix to distinguish different types of files. SCCS files are the exception. Here, **s.** precedes the file-name part of the complete path name.

To allow *make* easy access to the prefix **s.** requires either a redefinition of the rule naming syntax of *make* or a trick. The trick is to use the tilde (~) as an identifier of SCCS files. Hence, **.c~.o** refers to the rule which transforms an SCCS C source file into an object. Specifically, the internal rule is:

```
.c~.o:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Thus the tilde appended to any suffix transforms the file search into an SCCS file-name search with the actual suffix named by the dot and all characters up to (but not including) the tilde. The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

```
.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:
```

Obviously, the user can define other rules and suffixes which may prove useful. The tilde gives him a handle on the SCCS file-name format so that this is possible.

3.6 The Null Suffix

In the UNIX source code, there are many commands that consist of a single source file. It seemed wasteful to maintain an object of such files for *make*'s pleasure. The current implementation supports single-suffix rules, or, if one prefers, a null suffix. Thus, to maintain the program *cat* one needs a rule in the *makefile* of the following form:

```
.c:
        $(CC) -n -O $< -o $@
```

In fact, this **.c:** rule is internally defined so no *makefile* is necessary at all! One need only type:

```
make cat dd echo date
```

(these are notable single-file programs) and all four C source files are passed through the above shell command line associated with the **.c:** rule. The internally defined single-suffix rules are:

```
.c:
.c~:
.sh:
.sh~:
```

Others may be added in the *makefile* by the user.

3.7 Include Files

Make has an include file capability. If the string **include** appears as the first seven letters of a line in a *makefile* and is followed by a blank or a tab, the following string is assumed to be a file name that the current invocation of *make* will read. The file descriptors are stacked for reading *include* files so no more that about sixteen levels of nested includes is supported.

3.8 Invisible SCCS Makefiles

SCCS *makefiles* are invisible to *make*. That is, if *make* is typed and only a file named

s.makefile exists, *make* will do a *get*(1) on it, then read it and remove it; likewise for **−f** arguments and *include* files.

3.9 Dynamic Dependency Parameters

A new dependency parameter has been defined. It has meaning only on the dependency line in a makefile. **$$@** refers to the current "thing" to the left of the colon (which is **$@**). Also the form **$$(@F)** exists which allows access to the file part of **$@**. Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string **cat.c**. This is useful for building large numbers of executable files, each of which has only one source file. For instance the UNIX command directory could have a *makefile* like:

```
CMDS = cat dd echo date cc cmp comm ar ld chown

$(CMDS):    $$@.c
        $(CC) -O $? -o $@
```

Obviously, this is a subset of all the single-file programs. For multiple-file programs, a directory is usually allocated and a separate *makefile* is made. For any particular file which has a peculiar compilation procedure, a specific entry must be made in the *makefile*.

The second useful form of the dependency parameter is **$$(@F)**. It represents the file-name part of **$$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **/usr/include** directory from a makefile in the **/usr/src/head** directory. Thus the **/usr/src/head/makefile** would look like:

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCIDR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

3.10 Extensions of $*, $@, and $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The **D** refers to the directory part of the single-letter macro. The **F** refers to the file-name part of the single-letter macro. These additions are useful when building hierarchical makefiles. They allow access to directory names for purposes of using the *cd* command of the shell. Thus, a shell command can be:

```
cd $(<D); $(MAKE) $(<F)
```

An interesting example of the use of these features can be found in the set of *makefiles* in Appendix C.

3.11 Output Translations

Macros in shell commands can now be translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning is that **$(macro)** is evaluated. For each appearance of *string1* in the evaluated macro, *string2* is substituted. The meaning of finding *string1* in **$(macro)** is that the evaluated **$(macro)** is considered as a bunch of strings each delimited by white space (blanks or tabs). Thus the occurrence of *string1* in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because *make* usually concerns itself with suffixes. A more general regular expression match could be implemented if the need arises. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script which can handle all the C programs (i.e., those file ending in **.c**). Thus the following fragment will optimize the executions of make for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv $(LIB) $?
        rm $?
```

A dependency of the preceding form would be necessary for each of the different types of source files (suffices) which define the archive library. These translations are added in an effort to make more general use of the wealth of information that *make* generates.

4. CONCLUSIONS

The augmentations described above have increased the size of *make* significantly, but it is our belief that this increase in size is a reasonable price to pay for the resulting additional features.

Appendix A: Internal Definitions

The following *makefile* will exactly reproduce the internal rules of the current version of *make*. Thus if **make** **−r** ... is typed and a *makefile* includes this *makefile* the results would be identical to excluding the **−r** option and the *include* line in the makefile. Note that this output can be reproduced by:

```
make −fp − < /dev/null 2>/dev/null
```

The output will appear on the standard output.)

```
#       LIST OF SUFFIXES
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~

#       PRESET VARIABLES
MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=−O
AS=as
ASFLAGS=
GET=get
GFLAGS=

#       SINGLE SUFFIX RULES
.c:
        $(CC) −n −O $< −o $@
.c~:
        $(GET) $(GFLAGS) −p $< > $*.c
        $(CC) −n −O $*.c −o $*
        −rm −f $*.c
.sh:
        cp $< $@
.sh~:
        $(GET) $(GFLAGS) −p $< > $*.sh
        cp $*.sh $*
        −rm −f $*.sh

#       DOUBLE SUFFIX RULES
.c.o:
        $(CC) $(CFLAGS) −c $<
.c~.o:
        $(GET) $(GFLAGS) −p $< > $*.c
        $(CC) $(CFLAGS) −c $*.c
        −rm −f $*.c
.c~.c:
        $(GET) $(GFLAGS) −p $< > $*.c
.s.o:
        $(AS) $(ASFLAGS) −o $@ $<
```

```
.s~.o:
	$(GET) $(GFLAGS) -p $< > $*.s
	$(AS) $(ASFLAGS) -o $*.o $*.s
	-rm -f $*.s
.y.o:
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) -c y.tab.c
	rm y.tab.c
	mv y.tab.o $@
.y~.o:
	$(GET) $(GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	$(CC) $(CFLAGS) -c y.tab.c
	rm -f y.tab.c $*.y
	mv y.tab.o $*.o
.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm lex.yy.c
	mv lex.yy.o $@
.l~.o:
	$(GET) $(GFLAGS) -p $< > $*.l
	$(LEX) $(LFLAGS) $*.l
	$(CC) $(CFLAGS) -c lex.yy.c
	rm -f lex.yy.c $*.l
	mv lex.yy.o $*.o
.y.c :
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@
.y~.c:
	$(GET) $(GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	mv y.tab.c $*.c
	-rm -f $*.y
.l.c :
	$(LEX) $<
	mv lex.yy.c $@
.c.a:
	$(CC) -c $(CFLAGS) $<
	ar rv $@ $*.o
	rm -f $*.o
.c~.a:
	$(GET) $(GFLAGS) -p $< > $*.c
	$(CC) -c $(CFLAGS) $*.c
	ar rv $@ $*.o
	rm -f $*.[co]
.s~.a:
	$(GET) $(GFLAGS) -p $< > $*.s
	$(AS) $(ASFLAGS) -o $*.o $*.s
	ar rv $@ $*.o
	-rm -f $*.[so]
.h~.h:
	$(GET) $(GFLAGS) -p $< > $*.h
```

Appendix B: A Library Makefile

The following library maintaining makefile is from current work on LSX. It completely maintains the LSX operating system library.

```
#       @(#)/usr/src/cmd/make/make.tm     3.2
LIB = lsxlib

PR = vpr -b LSX

INSDIR = /r1/flop0/
INS = eval

lsx::   $(LIB) low.o mch.o
        ld -x low.o mch.o $(LIB)
        mv a.out lsx
        @size lsx

#       Here, $(INS) is either : or eval.
lsx::
        $(INS) 'cp lsx $(INSDIR)lsx && \
                strip $(INSDIR)lsx && \
                ls -l $(INSDIR)lsx'

print:
        $(PR) header.s low.s mch.s *.h *.c Makefile

$(LIB): \
        $(LIB)(clock.o) \
        $(LIB)(main.o) \
        $(LIB)(tty.o) \
        $(LIB)(trap.o) \
        $(LIB)(sysent.o) \
        $(LIB)(sys2.o) \
        $(LIB)(sys3.o) \
        $(LIB)(sys4.o) \
        $(LIB)(sys1.o) \
        $(LIB)(sig.o) \
        $(LIB)(fio.o) \
        $(LIB)(kl.o) \
        $(LIB)(alloc.o) \
        $(LIB)(nami.o) \
        $(LIB)(iget.o) \
        $(LIB)(rdwri.o) \
        $(LIB)(subr.o) \
        $(LIB)(bio.o) \
        $(LIB)(decfd.o) \
        $(LIB)(slp.o) \
        $(LIB)(space.o) \
        $(LIB)(puts.o)
        @echo $(LIB) now up-to-date.

.s.o:
        as -o $*.o header.s $*.s
.o.a:
        ar rv $@ $<
        rm -f $<
```

```
.s.a:
        as -o $*.o header.s $*.s
        ar rv $@ $*.o
        rm -f $*.o

.PRECIOUS:      $(LIB)
```

Appendix C: Example of Recursive Use of Makefiles

The following set of *makefiles* maintain the operating system for Columbus UNIX. They are included here to provide realistic examples of the use of *make*. Each one is named **70.mk**. The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is **ucb**. Here, we have used some of the conventions described in A. Chellis's paper entitled *Proposed Structure for UNIX/TS and UNIX/RT Makefiles*. FRC is a convention for *FoRC*ing *make* to completely rebuild a target starting from scratch.

./ucb makefile

```
#       @(#)/usr/src/cmd/make/make.tm     3.2
#       ucb/70.mk makefile

VERSION = 70

DEPS = \
        os/low.$(VERSION).o \
        os/mch.$(VERSION).o \
        os/conf.$(VERSION).o \
        os/lib1.$(VERSION).a \
        io/lib2.$(VERSION).a

#       This makefile will re-load unix.$(VERSION) if any
#       of the $(DEPS) is out-of-date wrt unix.$(VERSION).
#       Note, it will not go out and check each member
#       of the libraries.  To do this, the FRC macro must
#       be defined.

unix.$(VERSION):      $(DEPS) $(FRC)
        load -s $(VERSION)

$(DEPS):      $(FRC)
        cd $(@D); $(MAKE) -f $(VERSION).mk $(@F)

all:    unix.$(VERSION)
        @echo unix.$(VERSION) up-to-date.

includes:
        cd head/sys; $(MAKE) -f $(VERSION).mk

FRC:    includes;
```

```
#       @(#)/usr/src/cmd/make/make.tm     3.2
#       ucb/os/70.mk makefile

VERSION = 70

LIB = lib1.$(VERSION).a
COMPOOL=

LIBOBJS = \
        $(LIB)(main.o) \
        $(LIB)(alloc.o) \
        $(LIB)(iget.o) \
        $(LIB)(prf.o) \
        $(LIB)(rdwri.o) \
        $(LIB)(slp.o) \
        $(LIB)(subr.o) \
        $(LIB)(text.o) \
        $(LIB)(trap.o) \
        $(LIB)(sig.o) \
        $(LIB)(sysent.o) \
        $(LIB)(sys1.o) \
        $(LIB)(sys2.o) \
        $(LIB)(sys3.o) \
        $(LIB)(sys4.o) \
        $(LIB)(sys5.o) \
        $(LIB)(syscb.o) \
        $(LIB)(maus.o) \
        $(LIB)(messag.o) \
        $(LIB)(nami.o) \
        $(LIB)(fio.o) \
        $(LIB)(clock.o) \
        $(LIB)(acct.o) \
        $(LIB)(errlog.o)

ALL = \
        conf.$(VERSION).o \
        low.$(VERSION).o \
        mch.$(VERSION).o \
        $(LIB)

all:    $(ALL)
        @echo `$(ALL)' now up-to-date.

$(LIB)::$(LIBOBJS)

$(LIBOBJS):     $(FRC);

FRC:
        rm -f $(LIB)

clobber:cleanup
        -rm -f $(LIB)

clean cleanup:;

install: all;

.PRECIOUS:      $(LIB)
```

```
#       @(#)/usr/src/cmd/make/make.tm     3.2
#       ucb/io/70.mk makefile

VERSION = 70

LIB = lib2.$(VERSION).a
COMPOOL=

LIB2OBJS = \
        $(LIB)(mx1.o) \
        $(LIB)(mx2.o) \
        $(LIB)(bio.o) \
        $(LIB)(tty.o) \
        $(LIB)(malloc.o) \
        $(LIB)(pipe.o) \
        $(LIB)(dhdm.o) \
        $(LIB)(dh.o) \
        $(LIB)(dhfdm.o) \
        $(LIB)(dj.o) \
        $(LIB)(dn.o) \
        $(LIB)(ds40.o) \
        $(LIB)(dz.o) \
        $(LIB)(alarm.o) \
        $(LIB)(hf.o) \
        $(LIB)(hps.o) \
        $(LIB)(hpmap.o) \
        $(LIB)(hp45.o) \
        $(LIB)(hs.o) \
        $(LIB)(ht.o) \
        $(LIB)(jy.o) \
        $(LIB)(kl.o) \
        $(LIB)(lfh.o) \
        $(LIB)(lp.o) \
        $(LIB)(mem.o) \
        $(LIB)(nmpipe.o) \
        $(LIB)(rf.o) \
        $(LIB)(rk.o) \
        $(LIB)(rp.o) \
        $(LIB)(rx.o) \
        $(LIB)(sys.o) \
        $(LIB)(trans.o) \
        $(LIB)(ttdma.o) \
        $(LIB)(tec.o) \
        $(LIB)(tex.o) \
        $(LIB)(tm.o) \
        $(LIB)(vp.o) \
        $(LIB)(vs.o) \
        $(LIB)(vtlp.o) \
        $(LIB)(vt11.o) \
        $(LIB)(fakevtlp.o) \
        $(LIB)(vt61.o) \
        $(LIB)(vt100.o) \
        $(LIB)(vtmon.o) \
        $(LIB)(vtdbg.o) \
        $(LIB)(vtutil.o) \
```

```
        $(LIB)(vtast.o) \
        $(LIB)(partab.o) \
        $(LIB)(rh.o) \
        $(LIB)(devstart.o) \
        $(LIB)(dmc11.o) \
        $(LIB)(rop.o) \
        $(LIB)(ioctl.o) \
        $(LIB)(fakemx.o)

all:    $(LIB)
        @echo $(LIB) is now up-to-date.

$(LIB)::$(LIB2OBJS)

$(LIB2OBJS):    $(FRC)

FRC:
        rm -f $(LIB)

clobber: cleanup
        -rm -f $(LIB) *.o

clean cleanup:;

install: all;

.PRECIOUS:    $(LIB)

.s.a:
        $(AS) $(ASFLAGS) -o $*.o $<
        ar rcv $@ $*.o
        rm $*.o



#       @(#)/usr/src/cmd/make/make.tm    3.2
#       ucb/head/sys/70.mk makefile

COMPOOL = /usr/include/sys

HEADERS = \
        $(COMPOOL)/buf.h \
        $(COMPOOL)/bufx.h \
        $(COMPOOL)/conf.h \
        $(COMPOOL)/confx.h \
        $(COMPOOL)/crtctl.h \
        $(COMPOOL)/dir.h \
        $(COMPOOL)/dm11.h \
        $(COMPOOL)/elog.h \
        $(COMPOOL)/file.h \
        $(COMPOOL)/filex.h \
        $(COMPOOL)/filsys.h \
        $(COMPOOL)/ino.h \
        $(COMPOOL)/inode.h \
        $(COMPOOL)/inodex.h \
        $(COMPOOL)/ioctl.h \
```

```
        $(COMPOOL)/ipcomm.h \
        $(COMPOOL)/ipcommx.h \
        $(COMPOOL)/lfsh.h \
        $(COMPOOL)/lock.h \
        $(COMPOOL)/maus.h \
        $(COMPOOL)/mx.h \
        $(COMPOOL)/param.h \
        $(COMPOOL)/proc.h \
        $(COMPOOL)/procx.h \
        $(COMPOOL)/reg.h \
        $(COMPOOL)/seg.h \
        $(COMPOOL)/sgtty.h \
        $(COMPOOL)/sigdef.h \
        $(COMPOOL)/sprof.h \
        $(COMPOOL)/sprofx.h \
        $(COMPOOL)/stat.h \
        $(COMPOOL)/syserr.h \
        $(COMPOOL)/sysmes.h \
        $(COMPOOL)/sysmesx.h \
        $(COMPOOL)/systm.h \
        $(COMPOOL)/text.h \
        $(COMPOOL)/textx.h \
        $(COMPOOL)/timeb.h \
        $(COMPOOL)/trans.h \
        $(COMPOOL)/tty.h \
        $(COMPOOL)/ttyx.h \
        $(COMPOOL)/types.h \
        $(COMPOOL)/user.h \
        $(COMPOOL)/userx.h \
        $(COMPOOL)/version.h \
        $(COMPOOL)/votrax.h \
        $(COMPOOL)/vt11.h \
        $(COMPOOL)/vtmn.h

all:    $(FRC) $(HEADERS)
        @echo Headers are now up-to-date.

$(HEADERS): s.$$(@F)
        $(GET) -s -p $(GFLAGS) $? > xtemp
        move xtemp 444 src sys $@

FRC:
        rm -f $(HEADERS)

.PRECIOUS:    $(HEADERS)

.h~.h:
        get -s $<

.DEFAULT:
        cpmv $? 444 src sys $@
```