Stanford Exploration Project, Report 77, November 17, 1997, pages 451–??

How to use cake with interactive documents

Jon F. Claerbout & Martin Karrenbach [1]

keywords: *graphics, algorithm*

ABSTRACT

At SEP we file our research in electronic documents so that any of us can rebuild a research result of any other (on a wide variety of computers) by opening an electronic report, going to an illustration and pressing a button found in the figure caption. The caption button exposes a menu of choices, often including interaction and movies. Researchers achieve this reproducibility and interactivity by filing their work using systematic naming conventions, explained here and precisely defined by a four page document `SEP.idoc.rules` written in "cake" language. A typical research directory builds 1-10 illustrations. Each directory has a `cakefile` containing a list (`FIGLIST`) of buildable figure names. All figures can be built with the command `cake figures` and removed with `cake burn`. Precious figure files (ones not considered replaceable) are named `NAME.save`. The LaTeX macro, `\activeplot`, launches the UNIX command `cake NAME.menu` or `cake NAME.idoc`. The default cake rule is to come up with an XWINDOW menu `xtpanel` or to show a plot with `tube` or show a movie with `X11movie`. For unusual actions, you can override the default by making a rule for the target `NAME.action`.

INTRODUCTION

At SEP we have UNIX machines from many vendors including HP, IBM, Sun, TMC, DEC, and Convex, yet most of our programs run with *no changes* when we switch from one machine to another. The reason we can do this so easily is that Dave Nichols imported `cake` (a variant of the UNIX utility `make`) and prepared compiling and linking rules for our needs. Then also using `cake`, Jon Claerbout tackled the problem of organizing the management of his textbooks so that students could easily repeat all the calculations that generated all his illustrations. After organizing the first 300 or so figures, he learned what is common to all figure making and abstracted it into two pages of cake rules that are linked (included) into all figure making directories. By extracting what is common, what remains is the small nucleus that varies from one figure to the next. Thus, after learning what is in the common part, the

[1] **email:** not available

researcher-author needs to supply a minimum of information to give the reader a consistent and powerful interface. As Dave Nichols supplied SEP with `SEP.obj.rules` and `SEP.prog.rules` and some accompanying documentation, (at SEP this is found in `/usr/local/lib/cake`) Claerbout supplied you with `SEP.idoc.rules`, and early versions of this document describing it. Claerbout's original goal was research reproducibility. Since then, Martin Karrenbach enlarged the scope of `SEP.idoc.rules` to use Steve Cole's new `xtpanel` to give much greater reader interaction.

We assume our readers are familiar with the UNIX `make` utility which is widely used for software maintenance. We use the dialect of the `make` language called `cake` because it better expresses the needs of document development.

The main strategy is to isolate the reusable code in two files, `SEP.defs` that we "include" at the beginning of each figure-making cakefile and `SEP.idoc.rules` that we include at the end. Each of the many figure-building cakefiles starts with

```
#define USRLIBS SEPCLASSLIB
#include <SEP.defs>
```

The (optional) first line says that linking will include the library used in all Claerbout's classes. If you need additional libraries, add them to the end of this line. `SEP.defs` also sets up a default location for our figures because it contains the lines

```
#ifndef FIGDIR
#define FIGDIR ./../Fig
```

which means a sister directory to any of our figure-building directories is named `Fig` (which is generally a symbolic link to a nonprecious area). Some people prefer to keep their figures in a daughter directory so they start off with

```
#define FIGDIR ./Fig
#include <SEP.defs>
```

Often people write cakefiles so that if you come into a directory and type a simple `cake`, you will view a succession of figures emerging from the `tube` program. In the skeleton cakefile below, typing `cake` will show two figures named `data` and `spectrum`, and it will build them first if they are out-of-date.

```
#define USRLIBS SEPCLASSLIB
#include <SEP.defs>
#define FIGLIST data spectrum
default& : tube
$         comment
$ Application dependent stuff described later belongs here.
```

```
$
clean& : jclean
#include <SEP.idoc.rules>
#include <SEP.obj.rules>
#include <SEP.prog.rules>
```

The line containing `jclean` defines the clean-up procedure and will be explained later. At the bottom of this skeleton cakefile we included the compilation and linking rules of Dave Nichols, and we include `SEP.idoc.rules`, which has grown to about four pages of cake rules that are common to all our figure building and interactive documentation and whose content is our main subject here.

Directory layout and clean up

Typically we put in one directory all files needed to create a family of figures except for the input data (which maybe somewhere else), and the final plot files which go in `FIGDIR`. Software developement people often go to great lengths to keep separate directories for binary files of different machine types, but most researchers pile everything in the same directory, and whenever they switch machine types, they issue a `cake clean` to remove everything rebuildable (which includes all machine dependent code). Thus, you might expect every figure-building cakefile to include a rule like

```
clean& :
        rm junk* *.o *.x etc etc
```

Instead, most people prefer to use a cleaning rule named `jclean` which silently removes files named `core, a.out, *.o, *.x, *.H, *.A, *.M, *.ps, *.v, *.v3, *.V, junk*` and a few others produced by compilers and debuggers on various machines. It also removes fortran files that can be rebuilt from other languages. To invoke `jclean`, a figure-building cakefile includes the rule

```
clean& : jclean
```

This line means that when someone types `cake clean` in the directory containing a cakefile with this rule that `jclean` must be built. Neither `clean` nor `jclean` are files; they are abstract and the & is a signal for cake not to complain when no file named `clean` appears after the completion of `cake clean`. You probably agree with our standard cleaning rule, but if you want to check, you can type `cake -n clean` which does nothing but tell you what would be done were you to type `cake clean`. Another useful cleaning rule is `texclean` which cleans the directory of intermediate TEXrelated files. Experience shows that when you begin regular use of a cleaning rule like these, the number of files cluttering your life will drop immensely and you will have a much easier time finding things.

File naming conventions

Before we can explain any more, it will be helpful to have a list of our file naming conventions:

```
NAME.rst        source code needing saw,sat, ratfor and fortran passes
NAME.rs         source code needing saw, ratfor and fortran passes
NAME.rt         source code needing sat, ratfor and fortran passes
NAME.r          source code needing ratfor and fortran passes
NAME.f          fortran
NAME.rscm       source code needing saw, ratfor and CM Fortran 90 passes
NAME.rtcm       source code needing sat, ratfor and CM Fortran 90 qpasses
NAME.rcm        source code needing ratfor and CM Fortran 90 passes
NAME.fcm        source code in CM Fortran 90
NAME.o          object file, ready to be linked
NAME.x          executable (sometimes without .x)
NAME.ps         postscript
NAME.ps.save    precious postscript, not to be burned.
NAME.v          vplot (intermediate plot language)
NAME.v.save     precious vplot, not to be burned.
NAME.v3         vplot movie
NAME.v3.save    vplot movie, precious
NAME.H          data history file.
NAME.p          process parameter file
NAME.orig       "restore" evokes "cp NAME.orig NAME"
NAME.A          history file pointing to movie bytes.
NAME.warning    a file for miscellaneous explanations used in dialogs.
```

`SEP.obj.rules` and `SEP.prog.rules` are smart enough that simple linking jobs are done for you with no assistance. Thanks to Dave Nichols, these rules know about the different compiler flags and library names on different machines. Often several files must be compiled separately and linked together. Below is how we create the executable `myprog.x` which links a fortran main with some `.o` files.

```
#define MYPROG mymainprog.o sub.o fun.o
myprog.x : MYPROG
         FLINK(MYPROG, myprog.x)
```

First figure building example

For example, if the rule for building the figure named `spectrum` is

```
#define DATA /q2/wz/wz.26.H
```

```
FIGDIR/spectrum.v : DATA  myspectrum.x
        <DATA myspectrum.x | Graph > junk.H out=FIGDIR/spectrum.v
```

and your directory contains a fortran source file named `myspectrum.f`, then the included rules will compile `myspectrum.f` and link it to the libraries. In the above rule you can use `myspectrum` in the place of `myspectrum.x`. We prefer `myspectrum.x`, however, because our `jclean` rule knows about `.x` files whereas `jclean` does not know how to identify `myspectrum` as something that needs to be removed at cleaning time.

Each of our figure-building cakefiles contains the definition of a list of figures that it can build and manipulate. Above, that definition is

```
#define FIGLIST data spectrum
```

The application-specific rules in the cakefile should make `FIGDIR/data.v` and `FIGDIR/spectrum.v` (or possibly corresponding postscript files, but for now nearly all our figures are made in the `vplot` intermediate plot language and `SEP.idoc.rules` builds the postscript for us.) Because we always include `SEP.idoc.rules` in any of our figure-building directories we can issue the following commands:

```
cake burn               # Remove vplot and postscript figures
cake burnps             # Remove only postscript figures
cake figures            # build postscript for all figs
cake vplot              # build vplot for all figs
cake tube               # view all the vplot files
cake otube              # view vplots without rebuilding if out of date
cake idoc               # run thru all push button actions
cake NAME.tube          # view with tube the file FIGDIR/NAME.v
cake NAME.idoc          # do the electronic book action of figure NAME
cake NAME.menu          # pop up a menu of actions for figure NAME
cake install            # install programs in official place
```

Duplicate targets

To make each target, there must be one rule. Your rules are appended to those in `SEP.idoc.rules`. You cannot override a rule in `SEP.idoc.rules` by making another rule by the same name. (What does happen if you try is beyond the scope of this paper.) If you don't like one of the rules in `SEP.idoc.rules` you can make your own substitute, but your target must have a different name. If you dislike some feature of `SEP.idoc.rules` the best thing to do is convince us to change it. (You might be offering an improvement that would benefit us all, or you might simply need a better explanation than this document.) Less desirable alternatives are (1) jetison `SEP.idoc.rules` (and lose the functionality above) or (2) copy `SEP.idoc.rules` and revise it to your own liking (please explain your difficulty before trying this.).

ESSENTIAL CONTENTS OF SEP.IDOC.RULES

You don't need to know how `SEP.idoc.rules` works, but it might help you to know. Burning figures regularly leads to the proof that you can rebuild them. This is done by the following line found in `SEP.idoc.rules`

```
burn&:      [[ sub -i X X.burn  FIGLIST ]]
```

Reading `man cake` you see that this line, along with our earlier definition of `FIGLIST` will expand to

```
burn&:      data.burn  spectrum.burn
```

which means that `cake burn` done in this directory is equivalent to `cake data.burn spectrum.burn` which are the "targets" that can be built by the following idoc rule

```
%.burn&:
        -/bin/rm  -f FIGDIR/%.ps
        -/bin/rm  -f FIGDIR/%.v
```

To understand this rule, you need to know that in cakefiles, "`%`" is a wildcard like "`*`" in command shells.

We have a special category of figure types that cannot be easily rebuilt so they should not be burned. This is because they are scanned, prepared manually, with exotic hardware, excessive computer time, or because of any other difficulty. (Martin Karrenbach's latex environment marks such figures NR denoting "non reproducible" (Karrenbach and Nichols, 1990).) To save such files from burning, we give them names of the form NAME`.v.save` and NAME`.ps.save`. From time-to-time we redesign the vplot to postscript program (paper printing is usually black and white, but electronic documents use color) and then we want to burn all the postscript files that can be rebuilt from vplot. Here are the rules we are using now:

```
%.burnps& : if not exist FIGDIR/%.v.save
        -/bin/rm  -f FIGDIR/%.ps

%.burnps& : if     exist FIGDIR/%.v.save
        -/bin/rm  -f FIGDIR/%.ps.save
```

Thus `cake burnps` will not burn your precious plot files if you have no way to rebuild them. (While we are writing this, we are wondering if we should change these rules to something with the `cando` construction.)

Now let us do something constructive. To build all the figures in any of our books or documents, we have a shell that passes over all the cakefiles in the document tree and executes `cake figures`. That evokes the rules below from `SEP.idoc.rules`

```
figures&:  [[ sub -i X X.figure        FIGLIST ]]

%.figure&:  FIGDIR/%.ps.save        if     exist FIGDIR/%.v.save

%.figure&:  FIGDIR/%.ps             if not exist FIGDIR/%.v.save
```

These rules say that a postscript file must be created for every name in `FIGLIST`, either `FIGDIR/%.ps` or `FIGDIR/%.ps.save`. How the figures are created is determined by the following rules:

```
#ifndef PSARGS
#    ifdef HARDCOPY
#       define PSARGS color=n fat=1 fatmult=1.5
#    else
#       define PSARGS color=y
#    endif
#endif

#define PSTEXPEN(A,B) pstexpen FIGDIR/A FIGDIR/B  PSARGS
FIGDIR/%.ps.save : FIGDIR/%.v.save
        PSTEXPEN(%.v.save,%.ps.save)

FIGDIR/%.ps : FIGDIR/%.v
        PSTEXPEN(%.v,%.ps)
```

A few weeks before report generation, we insert in `SEP.defs` a line `#define HARDCOPY` (which you can do anytime in any of your cakefiles), burn all the postscript files, build the paper document, send it to the printer, then remove the line `#define HARDCOPY`, burn all the postscript files and rebuild them again for our electronic documents.

Recall, we said that `cake tube` will take the user through a `tube` view of all the plot files. The tube program will interpret single frame pictures (`NAME.v`) or multiple frame movies (`NAME.v3`). Paper documents can only cope with `NAME.v` files.

```
#define TUBEMOVIE ( cando FIGDIR/%.v3 or exist FIGDIR/%.v3.save )
%.tube&: %.tube1        if not TUBEMOVIE

%.tube&: %.tube3        if     TUBEMOVIE
```

These rules choose the movie file if a movie exists or can be made. If a movie cannot be made, the target becomes `%.tube1` and is interpreted by

```
%.tube1&:    FIGDIR/%.v        if     cando FIGDIR/%.v
        TUBE FIGDIR/%.v

%.tube1&:    FIGDIR/%.v.save  if exist FIGDIR/%.v.save and  \
                              not cando FIGDIR/%.v
        TUBE FIGDIR/%.v.save
```

These rules say to show the file `NAME.v` if it can be made, and if it cannot, then show the file `NAME.v.save`, if it exists. (Technically, an error condition arises for a few handmade postscript files for which `tube` cannot be used. We should eliminate this error message by including a do-nothing rule, or by calling a postscript previewer where possible.) The rules for `tube3` are like the rules for `tube1`.

INTRODUCTION TO INTERACTION

Martin Karrenbach's figure macro `activeplot` converts a figure-caption-button push to an invocation of `cake NAME.menu` or `cake NAME.idoc`. The choice is made in the TEXfile by either setting `\IdocMenu` (which will issue the command `cake NAME.menu`) or setting `\noIdocMenu` (which will issue the command `cake NAME.idoc`). By default a menu will come up with several possible buttons on it. The appearance of the menu is determined by the existence of certain files and the existence and feasibility of certain cake rules. You will see those buttons:

```
INTERACT
SHOW
WARNING/README
BURN
REBUILD
CLEAN
```

Pressing the push button `INTERACT` starts some kind of action most likely a movie or an interactive program. The `SHOW` button displays the figure as it is on the book page, but you can resize it, zoom to full screen or make it smaller. This is useful if you want to compare several figures side by side. The `WARNING/README` pushbutton displays some explanation usually related to burning and rebuilding. The pushbutton `BURN` removes a figure and the `REBUILD` pushbutton regenerates it. The `CLEAN` pushbutton removes all temporary files.

ADVANCED INTERACTION

The menu command is captured by lines similar to the following

```
%.menu&:
        xtpanel -cpp -var mechanism cake -var name % -DINTERACT \
                -file builder.xt panel >&/dev/null &
```

xtpanel is a graphical user interface scripting language which was created by Steve Cole and Dave Nichols (1992). We rely heavily on it for interacting with the readers of interactive documents. The menu will have `REBUILD` and `BURN` buttons if the figure is rebuildable. A figure is judged rebuildable if the cakefile can generate the file `FIGDIR/%.ps`. The command `cake NAME.burn` is issued by pressing the `BURN` button and pressing the `REBUILD` button issues the command `cake NAME.ps` If the figure is not rebuildable those buttons are inactive on the screen menu.

The menu will have an inactive `INTERACTION` button in case the author does not provide some interaction. An default interaction is possible if there exists one or more of the following:

- vplot movie files `FIGDIR/%.v3` or `FIGDIR/%.v3.save`.
- X11 movie files `FIGDIR/%.A` or `FIGDIR/%.A.save`.
- xtpanel files `%.panel`
- editable parameter files `%.p` and menu file `%.panel` and a backup parameter file `%.p.orig`

The command `cake NAME.idoc` is issued by pressing the `INTERACTION` button.

Pseudo reproducible research

Some figures are "hazardous" or impossible to rebuild, although the mechanism for rebuilding is working perfectly. One reason could simply be that it takes too long to rebuild the figure or that you need specific hardware to run the programs or that a data set is missing. The author can put messages or explanations into the file `NAME.warning`. This will trigger the existence of a button which lets the reader display the warning message before burning figures or attempting to build them. If the warning is ignored and the button is pressed anyway, a menu will prompt him to really destroy that figure.

Defining your own actions

If you do not want a menu, you should define your own `%.action`. The following two rules from `SEP.idoc.rules` allow you to capture what happens when `%.idoc` gets involved.

```
%.idoc & :  %.action       if     cando %.action

%.idoc & :  %.defaction    if not cando %.action
```

For example, if you want the figure caption pushbutton to launch an unusual action (whistle Dixie?) then your cakefile should have a target `NAME.action` that will whistle Dixie. Normally, you will prefer the default actions defined below. The default is to show a movie with `X11movie` if there is a movie or if one can be made. Otherwise, invoke `tube` on a vplot file.

```
%.defaction& : FIGDIR/%.A       if     cando FIGDIR/%.A
             < FIGDIR/%.A       MOVIE &

%.defaction& : FIGDIR/%.A.save if not cando FIGDIR/%.A      \
                                   and exist FIGDIR/%.A.save
             < FIGDIR/%.A.save MOVIE

%.defaction&: %.tube if  not exist FIGDIR/%.A.save        \
                     and not cando FIGDIR/%.A
```

This means if your cake files build a file called `FIGDIR/%.A`, then your readers get movies when they press your caption pushbuttons. You need write no explicit rules to show movies.

Finally, if the directory contains a file called `paper.tex` you should be able to type `cake read` to bring up `xtex` with included figures or `cake print` to print it. This is because of the rules below.

```
dvi&       :   PAPER.dvi

PAPER.dvi :   PAPER.tex
        texpr TEXPROPTS -v -2 -d PAPER.tex

view& read&:  figures          PAPER.dvi
        xtex                   PAPER.dvi
```

That's all! You have seen all the common part of our 100 cakefiles. If we add any more, it can only be philosophy, generalities, and maybe a few recalled pitfalls.

OTHER DIRECTORIES

Both `makefiles` and `cakefiles` tend to be self-contained entities, controlling the directory in which they reside but being ignorant of relationships within other directories that are managed by other cakefiles. Sometimes, we can merge the wisdom of

two cakefiles by having one invoke the other. There are subtle pitfalls, however, that Dave Nichols taught us how to avoid. If you depend on a file named `foo` in a sister directory `../Data` (which might be created and maintained by someone else) and you need `../Data/foo` to be built and up-to-date, the simplest thing you can do is

```
../Data/foo :
        cd ../Data;  cake foo
```

which you probably don't want to do for the subtle reason that the descendants of `../Data/foo` will always be rebuilt regardless whether `../Data/foo` rebuilds or not (notice that `cake` does not analyze both `cakefiles` simultaneously). Instead, you could do

```
../Data/foo:  if not exist ../Data/foo
        cd ../Data;  cake foo
```

which would build `../Data/foo` the first time (when it didn't exist) but then never again, even if something had changed in the sister directory that made `../Data/foo` out of date, so you probably don't want this either. Most likely, below is what you want:

```
../Data/foo:  if not {{ cd ../Data ; cake -q foo}}
        cd ../Data;  cake foo
```

which ensures that both `../Data/foo` and its descendants are rebuilt if only if they are out-of-date. The way this works is that the construction {{ `cd ../Data ; cake -q foo` }} is replaced by the return code of running `cake -q` in the directory `../Data`. (We learn from `man cake` that `cake -q` returns exit status 0 if targets are up-to-date and 1 otherwise.) Thus the obtuse rule above is an idiom that enables one cakefile to incorporate knowledge of another.

PHILOSOPHY AND PITFALLS

Two philosophical approaches to programming are "procedural" and "rule based". Fortran and shell language are examples of procedural languages and `cake` and `make` are rule-based languages. Many of the rules in a cakefile each invoke a small shell-like program. Some rules might contain a long string of shell-like commands with many intermediate results. It is a question of philosophy if you want to break up such a long string into separate rules. We generally prefer one long rule which is like a shell. The long string of commands is easier to understand and debug than a bunch of small rules. However, many circumstances suggest breaking up a long rule into

parts. The best reason to break a long rule into separate short ones is if some of the short ones are reusable, either in this directory, or in all directories. We never like to see repeated code in any language because it makes maintenance susceptible to inconsistency. Further, coagulating nifty rules into include files makes those nifty rules available everywhere. Another reason to break a long rule into parts is if some of the parts are time consuming and we plan frequent restarts from an intermediate place.

Editing the cakefile is changing the rules. What are the rules when the rules are changing? This is especially confusing for people switching from make to cake. (make requires you to keep all the intermediate files, so to force a rebuild you just remove any file that is defective because of the changing rules). Suppose you just used the editor to change `sign1=0` to `sign1=1` in the rule below.

```
FIGDIR/spectrum.v : DATA
        <DATA Ft3D sign1=1 | Graph > junk.H out=FIGDIR/spectrum.v
```

If you type `cake figures` the figure may not build because no files have changed their dates. Technically, you should have put `sign1` in a parameter file and then the date of the parameter file would have changed when you changed the value. Often you can use the system program `touch` to change the date of the data or the program, but that is a poor idea if these are system wide objects that don't belong to you. In this case (which we have chosen to be particularly pesky) you could type `cake spectrum.burn`, or you could temporarily change your rule to

```
FIGDIR/spectrum.v : DATA cakefile
        <DATA Ft3D sign1=1 | Graph > junk.H out=FIGDIR/spectrum.v
```

because the date of the `cakefile` itself did change when you changed the value of `sign1`. Later you must remember to remove this figure dependence on the cakefile, or this figure will rebuild every time you change anything in the cakefile.

Finally there are some hazards to coding in the cake language. We might think that rules with no dependencies would never need to run, but the inventor of cake decided (probably by experience) that such rules should always run.

Most hazards arise because there can be many rules and they can interact in unexpected ways. Sometimes cake will attempt to use a rule recursively and will run for a half minute and then give up. We learned this when we tried working with file names like NAME.`save.v`. It is particularly confusing if two rules can make the same target. Which should be used? We think cake should tell you that two rules can make the same target and you should get rid of one of them. Instead it seems that we cannot tell which will be used, or maybe it tries to use both. This hazard is especially likely with the wildcard `%`. Recently we had trouble when we had two similar rules, one for building plot files and one for merging them. These rules were

```
skmod%.v : mod%.v dat%.v
        vp_SideBySideAniso   mod%.v  dat%.v  > skmod%.v

%.A %.V %.v : %.H
        <%.H Byte > %.A
        <%.A Ta2vplot title=% labelsz=12 > %.V out=%.v
```

Notice that the second rule creates the same target as the first if "%" takes the value `skmod%`.

RELATED TOPICS

`SEP.idoc.rules` constitutes only a part of the essential wisdom of interactive documents. Other things you need to worry about are establishing the overall `seplib` environment and X environment. For this we refer you to (Claerbout, 1991) and (Dellinger and Tálas, 1992). Also, on any sizable document, there will be libraries and executables that figure-building cakefiles assume to be present but are not actually built by those figure-building cakefiles. In my textbook (Claerbout, 1992) the building process requires a preliminary tour of all the figure-building cakefiles with a `cake install` which builds a handful of executable main programs needed in the book that are not generally needed by the whole of SEP. Recently, we have been experimenting with another arrangement where such programs are kept in a `Src` directory, and cakefiles that need them use a symbolic link to the `Src` directory.

We also suggest a way to organize your paper directory; this is described in a companion paper by Karrenbach (1993). It can be found on CDROM in the directory `/sepcd5/sep77/martin2`. The example paper follows exactly the recommended layout.

LISTING

Below is the current copy of `SEP.idoc.rules`. You might also like to look at `SEP.defs` but it is not very interesting to read, and certainly not worth printing here.

```
/*
keywords: idoc idocrules idocrule idoc.rules idoc.rules rules rule
*/
#ifndef FIGLIST
#define FIGLIST
#endif
#ifndef FIGDIR
#define FIGDIR .
#endif

#include <SEP.class.defs>
```

```
$                                                   make .ps and/or .ps.save
figures&:  [[ sub -i X X.figure    FIGLIST ]]
$                                                   remove .v .ps .A
burn&:     [[ sub -i X X.burn      FIGLIST ]]
$                                                   remove .ps
burnps&:   [[ sub -i X X.burnps    FIGLIST ]]
$                                                   show .v or .v3 (or .save)
tube&:     [[ sub -i X X.tube      FIGLIST ]]
$                                                   show .v or .v3 even out-of-date
otube&:    [[ sub -i X X.otube     FIGLIST ]]
$                                                   make .v and .v3
vplots&:   [[ sub -i X X.vplot     FIGLIST ]]
$                                                   caption interaction pushbutton.
idoc&:     [[ sub -i X X.idoc      FIGLIST ]]
$
$ the next three rules are unannounced
$                                                   if .v.saved exists, remove .v
unvplot&:  [[ sub -i X X.unvpl     FIGLIST ]]
$                                                   mv %.orig  %
restore&:  [[ sub -i X X.restore   FIGLIST ]]
$                                                   if .p.orig exists, remove .p
makecd&:   [[ sub -i X X.makecd    FIGLIST ]]

#define REDUNDANT ( exist FIGDIR/%.v.save and exist FIGDIR/%.v )
%.unvpl& : if REDUNDANT
        /bin/rm FIGDIR/%.v

%.unvpl& : if not REDUNDANT

%.burn&: %.burnA
        -/bin/rm  -f FIGDIR/%.ps
        -/bin/rm  -f FIGDIR/%.v

%.burnA&: if exist FIGDIR/%.A
        -Rm FIGDIR/%.A

%.burnA&: if not exist FIGDIR/%.A
$       echo "nothing to do"

%.burnps& : if not exist FIGDIR/%.v.save
        -/bin/rm  -f FIGDIR/%.ps

%.burnps& : if exist     FIGDIR/%.v.save
        -/bin/rm  -f FIGDIR/%.ps.save

%.figure&:  FIGDIR/%.ps.save    if     exist FIGDIR/%.v.save

%.figure&:  FIGDIR/%.ps         if not exist FIGDIR/%.v.save

%.vplot&:  FIGDIR/%.v       if     cando FIGDIR/%.v

%.vplot&:                   if not cando FIGDIR/%.v
```

```
#ifndef PSARGS
#    ifdef HARDCOPY
#       define PSARGS color=n fat=1 fatmult=1.5
$#      define PSARGS color=n fat=1 fatmult=1.5 invras=n
#    else
#       define PSARGS color=y
#    endif
#endif

#define PSTEXPEN(A,B) pstexpen FIGDIR/A FIGDIR/B  PSARGS
FIGDIR/%.ps.save : FIGDIR/%.v.save
        PSTEXPEN(%.v.save,%.ps.save)

FIGDIR/%.ps : FIGDIR/%.v
        PSTEXPEN(%.v,%.ps)

#define TUBEMOVIE ( cando FIGDIR/%.v3 or exist FIGDIR/%.v3.save )
%.tube&: %.tube1        if not TUBEMOVIE

%.tube&: %.tube3        if     TUBEMOVIE

/* if all else fails bring up ghostview     Martin  */
%.tube&: %.tubeg        if    not cando %.tube1  and not cando %.tube3

%.tube1&:    FIGDIR/%.v       if     cando FIGDIR/%.v
        TUBE FIGDIR/%.v

%.tube1&:    FIGDIR/%.v.save  if exist FIGDIR/%.v.save and not cando FIGDIR/%.v
        TUBE FIGDIR/%.v.save

%.tube3&:    FIGDIR/%.v3      if     cando FIGDIR/%.v3
        TUBE FIGDIR/%.v3

%.tube3&:    FIGDIR/%.v3.save if exist FIGDIR/%.v3.save and not cando FIGDIR/%.v3
        TUBE FIGDIR/%.v3.save

%.tubeg&:                     if exist FIGDIR/%.ps
        ghostview FIGDIR/%.ps

%.tubeg&:                     if exist FIGDIR/%.ps.save
        ghostview FIGDIR/%.ps.save

%.otube&: %.otube1      if not TUBEMOVIE

%.otube&: %.otube3      if     TUBEMOVIE

%.otube1&:
        TUBE FIGDIR/%.v

%.otube3&:
        TUBE FIGDIR/%.v3

/*****************************************/
```

```
%.idoc & :  %.action       if     cando %.action

%.idoc & :  %.defaction    if not cando %.action

%.idoc & :  %.tube         if ( not cando %.action and not cando %.defaction )

%.defaction& : FIGDIR/%.A      if     cando FIGDIR/%.A
        <      FIGDIR/%.A      MOVIE

%.defaction& : FIGDIR/%.A.save if not cando FIGDIR/%.A and exist FIGDIR/%.A.save
        <      FIGDIR/%.A.save MOVIE

/*********************************************************/
/* latest and greatest  menu and xtpanel interaction  from Martin and Dave*/


#ifdef NOXTPANEL
#define XTPANEL exist fred_the_fireman
#define XTPANELCPP exist bob_the_butcher
#define EDPAR exist joe_the_baker
#else
#define XTPANEL ( exist %.p.orig or exist %.panel )
#define XTPANELCPP ( exist %.panel.pp )
#define EDPAR (  ( exist %.p.orig and not exist %.panel )  and ok %.vplot )
$#define EDPAR ( exist %.p.orig and not exist %.panel )
#endif

#define ALL_P [[ sub X.p X.p.orig [[ echo -n *.p.orig |sed 's,p.orig,,g' ]] ]]

#define ALL_ORIG [[ sub X X.restore [[ echo -n *.orig |sed 's,.orig,,g' ]] ]]

/* we have no real interaction provided */
#define CANT_INTERACT  ( (not cando %.action )  and (not cando %.defaction) )

/* truely non reproducable:            we do not want destroy/rebuild buttons */
#define NONREP ( not cando FIGDIR/%.ps  )

#define FIGWARN ( exist %.warning )


/* those *.xtpanel files live in /usr/local/lib/X11/xtpanel */
/* source is in interact/chooser */

%.defaction&:   if EDPAR
        xtpanel -var name % -file chooser.xtpanel

%.defaction&: %.panel   if XTPANEL
        xtpanel -file %.panel

%.defaction&: %.panel.pp   if XTPANELCPP
        xtpanel -cpp -file %.panel.pp

/*        cake -DNOXTPANEL %.idoc*/
```

```
/*****************************************/

/* alternatively to cake %.idoc issued from the document, */
/* we can issue cake %.menu, then we get a menu that to work interactively   */


%.menu&:         if  ( CANT_INTERACT and NONREP and FIGWARN )
        xtpanel -cpp -var mechanism cake -var name %  -DNOREBUILD \
                -DFIGWARN -file builder.xtpanel >&/dev/null &

%.menu&:         if  ( CANT_INTERACT and NONREP )
        xtpanel -cpp -var mechanism cake -var name %  -DNOREBUILD \
                 -file builder.xtpanel >&/dev/null &

%.menu&:         if  ( CANT_INTERACT            and FIGWARN )
        xtpanel -cpp -var mechanism cake -var name %  -DFIGWARN \
                 -file builder.xtpanel >&/dev/null &

%.menu&:         if                    ( NONREP and FIGWARN )
        xtpanel -cpp -var mechanism cake -var name %  -DINTERACT  \
                -DNOREBUILD -DFIGWARN -file builder.xtpanel >&/dev/null &

%.menu&:         if  ( CANT_INTERACT )
        xtpanel -cpp -var mechanism cake -var name %  \
                -file builder.xtpanel >&/dev/null &

%.menu&:         if                    ( NONREP )
        xtpanel -cpp -var mechanism cake -var name %  -DINTERACT \
                -DNOREBUILD -file builder.xtpanel >&/dev/null &

%.menu&:         if                               ( FIGWARN )
        xtpanel -cpp -var mechanism cake -var name % -DINTERACT \
                -DFIGWARN -file builder.xtpanel >&/dev/null &

/* this is default  */

%.menu&:
        xtpanel -cpp -var mechanism cake -var name % -DINTERACT \
                -file builder.xtpanel >&/dev/null &


/*****************************************/
/* restoration of crucial files */

/* called from cleaner.xtpanel */
all.restore&: ALL_ORIG

%.saverestore&:     if exist %.orig and not exist %
        /bin/cp  %.orig %

%.saverestore&:     if not (exist %.orig and not exist %)
        @echo % not restored since % exists already
```

```
%.restore& :     if exist %.orig
        /bin/cp  %.orig %

%.makecd& :     if exist %.p.orig
        /bin/rm  %.p

/*****************************************/

dvi&      :   PAPER.dvi


PAPER.dvi :   figures         PAPER.tex
        @-/bin/rm -f \*.aux \*.bbl
        texpr TEXPROPTS -v -bib -d PAPER.tex

view& read&:  figures         PAPER.dvi
        texpr TEXPROPTS -v    PAPER.dvi

ghost&:  figures         PAPER.dvi
        xtex -useGhostScript true PAPER.dvi

print&:       figures         PAPER.tex
        @-/bin/rm -f \*.aux \*.bbl
        texpr TEXPROPTS -bib  PAPER.tex

/*****************************************/
/* cleaning rules */

$               martin's tex cleanup
texclean&:
        -RM_CMD *.dvi *.aux *.bbl *.blg *.log TEX*

jclean&:  klean.usual klean.fort klean.cmfort

klean.usual& :
        @-/bin/rm -f \*.o \*.x \*.H \*.A \*.M \*.ps \*.v \*.v3 \*.V \*.trace
        @-/bin/rm -f junk\*
        @-/bin/rm -f core
        @-/bin/rm -f a.out
        @-/bin/rm -f mon.out
        @-/bin/rm -f paper.log

klean.fort& : [[ sub -i X.f X.klean  * ]]

#define CAN_MAKE_FORT exist %.r or exist %.rs or exist %.rt or exist %.rst or exist %.fs
%.klean&:  if CAN_MAKE_FORT
        @-/bin/rm %.f

%.klean&:  if not ( CAN_MAKE_FORT )


klean.cmfort& : [[ sub -i X.fcm X.klean  * ]]

#define CAN_MAKE_CMFORT exist %.rcm
```

```
%.klean&:  if CAN_MAKE_CMFORT
        @-/bin/rm %.fcm

%.klean&:  if not ( CAN_MAKE_CMFORT )


$       Make a dummy install target for every cakefile.
install&:
$       This is done sequentially with nontrivial install targets, if any.
$       There should be some way to turn off the message "install is up to date"
```

REFERENCES

Claerbout, J. F., 1991, SEPlib and SEP software-computer: SEP–**71**, 283–286.

Claerbout, J. F., 1992, Earth Soundings Analysis: Processing Versus Inversion: Blackwell Scientific Publications.

Cole, S., and Nichols, D., 1992, Xtpanel: An interactive panel builder: SEP–**75**, 497–520.

Dellinger, J., and Tálas, S., 1992, A tour of SEPlib for new users: SEP–**73**, 461–502.

Karrenbach, M., and Nichols, D., 1990, Progress towards the interactive book: SEP–**67**, 149–160.

Karrenbach, M., 1993, How to organize an interactive document: SEP–**77**, 417–426.