

# The COMB Cookbook

Marc W. Pound, John Bally and Robert W. Wilson

August 22, 1990 — for August 1990 version of COMB

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	The COMB Philosophy . . . . .	3
1.2	About the Cookbook . . . . .	3
<b>2</b>	<b>Getting Started</b>	<b>3</b>
<b>3</b>	<b>Data Formats: Scans and Stacks</b>	<b>4</b>
<b>4</b>	<b>Commands</b>	<b>5</b>
4.1	Argument types . . . . .	5
4.2	Command Syntax . . . . .	6
4.3	Hand-hold Mode . . . . .	6
4.4	COMB's History Mechanism: Recalling and Editing Commands . . . . .	7
<b>5</b>	<b>Getting Data Into COMB</b>	<b>7</b>
5.1	tar or dd Tapes . . . . .	8
5.2	Stand-alone Programs . . . . .	8
5.3	Spectra in FITS Format . . . . .	8
<b>6</b>	<b>Manipulating and Plotting Spectra</b>	<b>9</b>
6.1	Baseline Removal . . . . .	9
6.2	Sorting and Storing Spectra . . . . .	10
6.3	Bad Channel Removal . . . . .	11
6.4	Folding Frequency-Switched Spectra . . . . .	12
6.5	Do Loops . . . . .	12
6.6	Co-adding Spectra . . . . .	12
6.7	Redirecting Output . . . . .	13
6.8	Obtaining Hard Copies of Plots . . . . .	13
<b>7</b>	<b>Making Maps</b>	<b>14</b>
7.1	Relative Coordinates . . . . .	14
7.2	Spatial-Spatial Maps . . . . .	14
7.3	Spatial-Velocity Maps . . . . .	15
7.4	Recontouring Maps with cp . . . . .	16
7.5	Reading/Writing FITS images . . . . .	16

<b>8</b>	<b>Advanced Processing: Macros</b>	<b>16</b>
8.1	Variables . . . . .	17
8.1.1	Global Variables . . . . .	17
8.1.2	Global Strings . . . . .	17
8.1.3	Stack Header Variables . . . . .	18
8.1.4	Labelling Inside Plots . . . . .	19
8.2	Macro Arguments . . . . .	19
8.3	Some Useful Macros . . . . .	20
<b>9</b>	<b>Figures</b>	<b>22</b>
<b>A</b>	<b>How to Install COMB</b>	<b>32</b>
<b>B</b>	<b>FITS Software</b>	<b>34</b>
<b>C</b>	<b>List of COMB Commands</b>	<b>35</b>
<b>D</b>	<b>For More Info</b>	<b>37</b>

## 1 Introduction

*COMB* is a powerful analysis package for single-dish radio astronomical spectral line data reduction. It can be installed on any computer using the UNIX<sup>1</sup> operating system (see Appendix A). At present, it is most frequently used on SUN workstations, although it has been installed on VAX and other computers, including 386 machines running UNIX. The program was written mostly by R. W. Wilson at AT&T Bell Laboratories in C and FORTRAN.

### 1.1 The COMB Philosophy

*COMB* (pronounced cäm) has been designed to make spectral line data reduction easy and *automatic*. As the Bell Labs database has grown (it now exceeds 700,000 spectra), this automatic feature has been indispensable. *COMB*'s 61 simple commands can be concatenated using semi-colons (a la UNIX), nested in do-loops, or made into saveable macros (which themselves can be nested). Even for an inexperienced user, the process of reducing hundreds of spectra and making a contour map (or FITS image) can be reduced to a task requiring less than ten minutes.

### 1.2 About the Cookbook

This cookbook will acquaint neophyte users of *COMB* with the basic tools necessary to reduce their data painlessly. It is intended to supplement *COMB*'s on-line documentation by providing simple recipes for the basic reduction operations. Section 2 provides information about getting started. Section 3 introduces data format. Section 4 describes commands syntax and arguments. Section 5 tells how to get external data, such as FITS, into *COMB*. Section 6 deals with simple manipulation of spectra such as baseline removal, plotting and storing. Section 7 is all about making maps. Section 8 describes how to set-up and use macros. Throughout this cookbook, input to be typed by the user appears in this font (a carriage return is implied at the end of each user input), this font will be used for command names, command arguments and *COMB* output messages, , and *this font is used for UNIX file names*. We will use \$ to symbolize the UNIX operating system prompt and the -> symbol is your friendly *COMB* prompt.

This cookbook was written using the L<sup>A</sup>T<sub>E</sub>X Document Preparation System developed by Leslie Lamport. A copy of the cookbook L<sup>A</sup>T<sub>E</sub>X document is in the file `/usr/comb/doc/cookbook.tex`.

## 2 Getting Started

*COMB* can be run from any type of terminal. It is, of course, most useful on one with some sort of graphics capability. In order to have *COMB* plot graphics on a CRT, you need a graphics terminal or a window which emulates a graphics terminal. The graphics terminals which are supported are hp2648, hp2623, tek4010, and vt100 (with retrographics). (*COMB* has a very nice interface to tek4112, AT&T 5620 and AT&T 630 terminals, but they aren't common.) It will be most convenient if your environment variable TERM is set to one of the above if you are using one of those terminals. The most common way to get *COMB* graphics is with a terminal emulator. With a PC, PCPLOT<sup>2</sup> minimally emulates the vt100/tek4014 combination. With a Macintosh, Versaterm is a very nice emulator of vt100/tek4014. In either of these cases have TERM=vt100 in your environment. With a Sun workstation, the best emulator is package is `gterm` from NOAO/IRAF. With TERM=SUN or TERM=sun, *COMB* expects a tek4014 emulator such as `gterm` in the window. After you have installed *COMB* (see appendix A), sit down at your terminal and type `comb`. If you have `gterm` , do this in the `gterm` window.

```
$ comb                               Start COMB
Welcome to comb
```

<sup>1</sup>UNIX is a registered trademark of AT&T.

<sup>2</sup>PCPLOT is a registered trademark of IBM.

->

Most *COMB* commands consist of two-character names chosen to be suggestive of their function. There are also a few single character commands. The command *lc* (list commands) gets you a summary list. Appending the string *?!* to any command name, will give the on-line documentation about that command. The string *??* appended to the command will give the input arguments for that command.

-> <i>lc</i>	Produce a listing of <i>COMB</i> commands along with a one-line description.
-> <i>pl ?!</i>	Give detailed description of the <i>pl</i> (plot) command which displays spectra on the screen.
-> <i>e 'ls'</i>	Execute the UNIX command <i>ls</i> for the directory. <i>COMB</i> doesn't care if you use single or double quotes, as long as they are matched.
-> <i>q</i>	Exit <i>COMB</i> ...
<i>\$</i>	... and get the shell prompt from UNIX.

Typing *help* at the *COMB* prompt will give you instructions about how to get information about syntax, data formats, and commands.

### 3 Data Formats: Scans and Stacks

*COMB* assumes that data exists in two forms – raw spectra which have been untouched by human hands called *scans* and spectra which may have been processed called *stacks*. These data are located on disk in directories. At Bell Labs, scans are stored by the telescope computer in files in the directory */cdata* and can be accessed by the *gt* (get) command, *e.g.*, *gt 86a614* will get the 614th spectrum from the raw data file */cdata/c86a*. Any filename can be specified to be the raw data file with *nf* (name file), *e.g.*, *nf '/mydirectory/mydata'*.

At most institutions, raw data files will not exist, and all spectra will reside in “stacks directories”. A stacks directory contains several files called *data*, *default*, *index*, *macros*, and *search* which have the following functions:

<i>data</i>	Contains the raw headers and spectra in binary form.
<i>default</i>	ASCII file containing information on user preferences such as relative coordinates.
<i>index</i>	A table of pointers to the data location corresponding to the beginning of each stack in <i>data</i> .
<i>macros</i>	ASCII files containing the user macros applicable to these stacks only.
<i>search</i>	A lookup table linking stack numbers to locations in the sky.

Stack directories are accessed from inside *COMB* by *ns* (name stacks), *e.g.*, *ns '/usr/jb/data/Orion'*. Stacks in a directory are numbered sequentially starting at 1. The first three stacks, however, are special temporary (“core-resident”) locations for spectra, similar to the stack on a pocket calculator, and are not actually “members” of the stacks files.

Stack 1	The current spectrum. When you <i>gt</i> or <i>rt</i> (retrieve) a spectrum, it is automatically stored in stack 1. <i>pl</i> will always plot the contents of stack 1.
Stack 2	Generally used as an accumulator for co-adding spectra.

Stack 3 an additional temporary location.

Stacks 4 through 9 are by default left empty by *COMB* when automatically storing data, and hence ignored by commands like *cm* that use stacks data (you can reset these defaults.) Thus, they can be used as special storage locations. They are part of the stacks files and will be saved between *COMB* sessions.

A stacks directory can be created in several ways. The *ns* command will create a new directory if the named one doesn't yet exist. Spectra can then be stored using *st* (store). Alternatively, a stand-alone program (see §5.2) can create the stacks directory and associated files, storing the data and building the index file. (*COMB* will build the search file from the data, see documentation for *up*, [update search file]). And, of course, a stacks directory from another institution can be copied from tape. More details of the stack format are given in §8.1.3.

## 4 Commands

### 4.1 Argument types

A *COMB* command consists of 2 basic parts: a command name followed by a delimited list of arguments. The delimiters may be commas or spaces (it's okay to mix the two). The command name is a one- or two-character mnemonic for the function, e.g., *cm* is contour map. The arguments can take a variety of forms: numerical, character string, numerical range, list, flag, toggle, and expression. The command *lk* (look), which displays the location of spectra on the sky, is a good example of most of these types.

-> lk ??

Look at where stacks are ``lk''

Horizontal limits ``xl''

Left x ``lx'' - REAL, OLD(0)

Right x ``rx'' - REAL, OLD(0)

Vertical limits ``yl''

Bottom y ``by'' - REAL, OLD(0)

Top y ``ty'' - REAL, OLD(0)

Stack number limits ``sn''

Low ``l'' - INTEGER, OLD(10)

High ``h'' - INTEGER, OLD(262143)

Center Frequency (MHz) ``fr'' - REAL RANGE, OLD(0\_300000)

Plot stack numbers ``psn'' - FLAG(no)

Horizontal label ``hlab'' - STRING, AUTOMATIC

Vertical label ``vlb'' - STRING, AUTOMATIC

Main label ``mlb'' - STRING, AUTOMATIC

Check search vs stack ``chk'' - FLAG(no)

-> lk,20,-10,-10,15,10,98,110000\_111000 psn:yes mlb:'My Stacks'

The first four arguments are examples of *numerical* (in this case, REAL) input. These four specify that *lk* plot only those stacks between the relative coordinate offsets  $-10 \leq x \leq 20$ ,  $-10 \leq y \leq 15$ . The next two arguments are also numerical (INTEGER). They tell *lk* to use only stacks between stack numbers 10 and 98. If real numbers had been used here, the *COMB* parser would round them to the nearest integers. The next argument is an example of a *range*. Ranges' limits are separated by the underscore (.) character. This argument specifies

the range of frequencies (MHz) that spectra must satisfy for `lk` to plot their location. The next argument, `psn:`, is a *flag* which makes `lk` mark a location with the stack number(s) at that location rather than the default crosses (+). A flag simply tells the parser to turn on or off a particular feature of a command. Note that `psn:` is the same as `psn:yes` and leaving `psn:` out of the argument list is the same as `psn:no`. The final argument in the list is a *string*, indicating the main label on the plot should be ‘My Stacks’. Figure 1 is the result (a “look map”) of the above command. A *toggle* is similar to a flag, except that it will turn a feature off if it is on and vice versa, *e.g.*,

```
-> op ap:                               Turn on automatic plot option (so you don't have to keep typing pl
                                     after rt ).
-> op ap:                               Turn off automatic plot option.
```

Note flags and toggles delineated by a colon do not also need to be separated by commas or spaces.

```
-> op ap:ortho:yw:
```

is syntactically legal. The type *list* is a delimiter-separated list of variables all describing the same argument, *e.g.*, `cp sc:[0,20,2,39,3,49]`. We will meet the last argument type, *expression*, when we cover `do` loops (§6.5).

Notice that some of arguments in the syntax tree above are described by the words OLD, AUTOMATIC, or SET. An OLD variable starts out at a default value, shown in parentheses, that will stay changed once it is changed by the user. It’s previous value is remembered and accessible via the `&` operator (see §6.2). An AUTOMATIC argument is a number (or string) calculated by *COMB* which can have a variable default value, *e.g.*, axes lengths or labels on a spectral plot. A SET argument has a set (as opposed to a variable) default value. For an AUTOMATIC or SET arguments, the previous value is not remembered and therefore not accessible via the `&` operator.

## 4.2 Command Syntax

Command arguments are arranged in a *syntax tree*, with a unique *label* to specify each node of the tree. This syntax tree is what is printed out when you type `??` after a command name. Labels, such as `psn:`, `mlb:`, `ap:` in the above examples, *must* be delineated by a colon. If the arguments are entered in the order in which they appear in the syntax tree, then the labels are not needed. Use of labels, however, allows arguments to be entered in any order or arguments to be skipped altogether, if desired, and is also handy for remembering arguments in the list. Arguments following labels may take optional square brackets, *e.g.*, `lk fr:[110200 _110300]`. Square brackets are *required* when the arguments type is a list, *e.g.*, `cp sc:[0,20,2,39,3,49]`.

## 4.3 Hand-hold Mode

The easiest way to learn the input list of a command is to use *COMB*’s hand-hold mode. Typing `?` after a command name will cause *COMB* to prompt you for each argument using the same syntax tree that is printed out when you use `??`. For arguments with default values (OLD, AUTOMATIC, or SET), entering a carriage return at the argument prompt will input the default value. After you input all the arguments, *COMB* will tell you the most general syntactically correct form everything you just typed, *e.g.*,

```
-> lk ?
:
You could have typed:
lk xl:[10,-10]   yl:[-5,5]   sn:[10,20000]   fr:0 _300000
```

The command as you could have typed it is added to your history file (see below).

#### 4.4 COMB's History Mechanism: Recalling and Editing Commands

The *COMB* history mechanism, a way of editing and/or re-executing previous command lines, is based on that of the UNIX Korn shell, *ksh*. It is essentially a one-line version of the screen-editor program, defined by the environment variable *VISUAL* in the *.profile* (or *.login*) file in your home directory, which operates on a command string. *ksh* currently supports the editors *vi* and *emacs*. *You do not have to be running ksh as your normal UNIX shell to make use of the COMB history mechanism.* Anything typed at the *->* prompt is stored in the file *.combhistory* in your home directory. Therefore, commands issued during previous *COMB* sessions are accessible to the history mechanism.

The history mechanism is enabled by the escape ([ESC]) key for *vi* mode, or the control ([CNTL]) key for *emacs* mode. In *vi* mode, [ESC]k or [ESC]- will scroll back through the stored command lines ([ESC]j or [ESC]+ for forward scrolling), which you can edit using normal *vi* commands. In *emacs* mode, [CNTL]P scrolls backwards and [CNTL]N scrolls forward. In either mode, a carriage return will cause the currently displayed command line to be executed. For more information on the editor features, consult the *vi* or *emacs* manuals, or read the "editing mode" sections under *man ksh*.

In addition to the *ksh*-like history mechanism, *COMB* also has an "command archive" mechanism (left over from the days before *ksh*), allowing archiving, re-execution, or editing of the immediately previous command line. It is completely distinct from the *ksh*-like history. *COMB* keeps the previous command line and allows the user to access it with the following commands:

<i>.or .e</i>	Execute the previous (old) command as it is.
<i>.p</i>	Print the old command.
<i>.an</i>	Store the old command in archive n, n = 0 through 9
<i>.n</i>	Retrieve the command in archive n.
<i>.s/xxx/yy</i>	Edit the old command substituting the string <i>yy</i> for the first occurrence of <i>xxx</i> . The delimiter may be any character, not just '/

For example,

```

-> gt 88a6789
-> li 1          Remove a first-order baseline
-> .a0          Store "li 1" in archive 0
-> gt dt:       Get next data scan
-> .a1          Store "gt dt:" in in archive 1
-> .0           Retrieve command line in archive 0
-> .p           print it
li 1
-> .           and execute it
-> .1          Retrieve command line in archive 1 (gt dt: )
-> .           and execute it

```

## 5 Getting Data Into COMB

There are 3 simple ways to get data into *COMB*.

1. Move a stacks directory created by another *COMB* program from tape to disk using the UNIX utilities *tar* or *dd*.

2. Use `af` (attach FITS) to read in FITS data (spectra or images) from tape or disk.
3. Use a stand-alone program to convert data from another observatory into spectra in *COMB* format and write the spectra into a stacks directory.

### 5.1 tar or dd Tapes

Stacks files written to tape using the UNIX tape archive facility `tar` can be extracted by `tar xf tape_drive stacks_directory`. For `dd`, use `dd if=tape_drive of=stacks_file`. `tar` is preferable since it copies directories recursively. `dd` does not. See documentation under `man tar` and `man dd`.

### 5.2 Stand-alone Programs

A few C programs have been written to convert spectra from other observatories into *COMB* stacks. They are:

<code>arc2comb</code>	Convert spectra from the Boston University-Arecibo HI Survey into <i>COMB</i> stacks. Written by Marc Pound.
<code>FITS_TO_STACKS</code>	Convert spectra in IRAM FITS format into <i>COMB</i> stacks. Written by John Bally.
<code>cube</code>	Convert a FITS datacube to <i>COMB</i> stacks. It assumes the datacube is has spatial coordinates in a ‘relative coordinate’ system (see <code>cm ?!</code> and <code>rc ?!</code> ) and does not deal with sky projections at all. Written by Marc Pound.

After creating a stacks directory with one of these programs, you must run `up` on them to make a search file. These programs are available on request by their authors. If you wish to write your own program for converting data from other formats, these can be used as templates.

### 5.3 Spectra in FITS Format

Imagine you have just received a data tape from the SEST in Chile containing spectra written in IRAM’s FITS format. You can use the function `af` (attach FITS file) to read a spectrum into stack 1, `ns` to make the stacks directory and `st` to store the spectra in it. [Note: `af` will not work unfolded frequency-switched spectra, as there is no single-dish FITS standard for such data. If a standard is ever agreed upon, `af` will be re-written accordingly.]

```

-> af 'dev/rmt12' st:          Attach the tape drive /dev/rmt12 and read the
                              first spectrum from the FITS tape into in stack
                              1.

-> ns 'SESTdata'             Name the stacks directory “SESTdata” as the
                              present working directory for the spectra. Since
                              this is a new directory, ns will create it.

In command ns Creating stacks Directory SESTdata
Do you want to continue? y
* * CREATING SESTdata/macros * *
-> st a:                     Store the spectrum (st) in the 1st available
                              stack (a:) in the stacks directory. In this case,
                              there are no spectra in the directory,so the data
                              will go into stack 10. (Recall COMB leaves
                              stacks 4 through 9 empty for special use.

```



```
Stored in stack 10
```

```
-> af '/dev/rmt12' st:
-> st a:
```

Read an other spectrum into stack 1.

Store it in the next available stack, in this case, stack 11.

```
Stored in stack 11
```

```
-> do 10 {af '/dev/rmt12' st;; st a:}
```

Repeat the FITS file-reading operation 10 more times. This fills stacks 12 through 21 with data. Note the use of “;” to concatenate *COMB* commands. Also note the *do* loop, whose argument is in the curly brackets.

```
Stored in stack 12
```

```
:
```

```
Stored in stack 21
```

```
-> rt 10
```

Retrieve stack 10 and place it in stack 1. Note *rt* is used to place stacks in stack 1 and *gt* is used to place scans in stack 1.

```
-> pl
```

Plot spectrum. See Figure 2a.

## 6 Manipulating and Plotting Spectra

### 6.1 Baseline Removal

Before removing a baseline from a spectrum, you have to retrieve the spectrum and set the range of velocities to be excluded from the baseline fit because they contain a line. To do this, set the so-called “use array” with *us*.

```
-> us -30_30
```

Set use array to exclude the range  $-30$  to  $30$  in the current plot units. The default is velocity in  $\text{km s}^{-1}$ . (*pl* ?! for details on plot units.)

```
-> li 1; pl
```

Take out a 1st order baseline and plot the result. (Figure 2b) Note that the horizontal line just above the axis label has a segment missing just in the range  $-30$  to  $30 \text{ km s}^{-1}$ . This line always shows the current state of the use array.

```
-> us st:
```

Save the baseline parameters for future use. They can be reset with *us s:* to their initial state.

```
-> st 10
```

Store the spectrum in stack 10 (which is where it came from). Note that the original data will be overwritten. *COMB* catches this and asks your permission to overwrite the data and will do so only if you respond with a “y”.

```
In command st stack 10 contains data.
```

```
Do you want to continue? y
```

Baselines up to order 15 can be removed from spectra. The usual caveats apply to high-order baseline fits. For frequency-switched data, channels within a specified width of the signal and reference channels are automatically excluded from the fit in addition to the use array. *COMB* sets this width automatically, but it can also be specified as an argument to *li* (see *li* ?!). You can set the use array with the cursor using the macro *uscr* defined in the global macros file (see §8 and *cr* ?!).

The flag *xt:* will cause *li* to remove the same baseline from stack 2 as it does from stack 1. This is useful for

removing baselines from spectra where the emission line takes up too much of the band, leaving too few channels from which to compute a baseline fit, and where you have a broader band spectrum taken simultaneously.

```
-> gt n6r2655 exp;; st 2           Get scan from spectral expander backend, in this case 12.5 kHz
                                   channel width. The line takes up too much of the band to remove
                                   a reliable baseline.

-> pl hst:tk:                       Plot the spectrum histogram style with no tick grid. Figure 3a.
-> op ap:                             Turn on auto-plot option
-> gt nbe:                             Get the narrow backend of the same scan (250 kHz channel width).
                                   Figure 3b.
-> us -9_0 st;; li 1 xt:             Compute baseline parameters from stack 1 and remove the baseline
                                   from stacks 1 and 2. Figure 3c.
-> pl h:-10,0; rt 2; pl ovl:         Compare the baseline fits by overlaying the two spectra and display-
                                   ing a limited horizontal range. Figure 3d.
```

The flags `exp:` and `nbe:` are peculiar to Bell Labs since spectra taken simultaneously with different backends are given the same scan number. You will most likely use `rt` to retrieve two separate stacks, instead of `gt` for two separate backends. *COMB* will give a warning if the stacks have different scan numbers, but you can still remove the baseline.

## 6.2 Sorting and Storing Spectra

Suppose you want to take data from one stacks directory, do something with it, and store the result in a second directory. You can attach two stacks directories, called the foreground and background directories, to *COMB* simultaneously. Let's assume that you want to reduce the SEST data from the previous example without overwriting the original unreduced data. You need to open a second stacks directory (directory two). This can be done by

```
-> ns 'SESTreduced' dt:             The flag dt: stands for "directory two"
In command ns Creating stacks Directory SESTreduced.
Do you want to continue? y
* * CREATING SESTreduced/macros * *

At this point two directories are open: The foreground directory is SESTdata and the background directory is SESTreduced. Data can be easily transferred between the two directories.

-> ns p:                             To see the names of the working directories.
Foreground - SESTdata, last stk = 3, next = 22
Background - SESTreduced, last stk = 3, next = 10
-> rt 10; li 3; st a:dt:             Retrieve stack 10, remove a 3rd order base-
                                   line, and store the result in directory two.

Stored in stack 10
-> do 11 {rt &+1; li 3; st a:dt:}    Remove a 3rd order baseline for the next
                                   11 spectra in the foreground directory, stor-
                                   ing the results in the next available stack lo-
                                   cations in the background directory. See §6.5
                                   for more on do loops.

Stored in stack 11
```

```

:
Stored in stack 22
-> .                               Re-execute the last command (the do loop.)
Stored in stack 13
:
Stored in stack 34

```

In this example, a feature of *COMB* is illustrated: the idea of incrementing the values of OLD variables. *COMB* stores the values of command arguments of type OLD each time they are changed by a new issuance of the command. In the call `rt 10` the value of the stack argument, 10, has been explicitly stated. When *COMB* encounters `&` in an argument, it substitutes the previous value of this argument for the `&`. Thus `&+1` increments the previous value by 1, `&+2` increments by 2, etc., storing the new value after the operation is completed. Similarly, `&-1` decrements the value by 1. Thus, each time `rt &+1` is issued in the `do` loop, the value of the stack argument is incremented by 1, and the next stack is retrieved. Any syntactically legal expression can follow the `&`,

```

-> rt &*8^(1/3)                      Retrieve the the stack whose number is twice the current stack number
-> li &*&                             Remove the square of the previous baseline order.

```

This mechanism can be used to change any numerical argument (of type OLD) of a command by any amount.

To empty stack of its contents, use `em`.

```

-> em 22                               Empty stack 22
-> em 23_34                            Empty stacks 22 through 34 inclusive.

```

Since `em` simply marks the stacks as overwrite-able, you can get back emptied stacks, but *only if they have not been overwritten and you have not quit the session of COMB in which they were emptied*. If you quit *COMB*, the emptied stacks are *not* retrievable.

```

-> em 22_34 rev:                       Un-empty stacks 22 through 34 inclusive

```

### 6.3 Bad Channel Removal

Bad channels in spectra can be removed with `e1` (eliminate). This command accepts a range of either channel number(s), the velocity, or the frequency of the bad channel(s); the choice is set by the current plot units. The eliminated channels are set to the average of the two adjacent channels. Since `e1` only works on stack 1, you must retrieve the desired stack first.

To identify the bad channel, you can use `f1` (flag) which displays a vertical line at the specified value. For example,

```

-> rt 10; pl ch:                       Plot a spectrum using channels for horizontal plot units.
-> fl 67                                Flag channel 67.
-> el 67; st 10 dc:                    Eliminate the channel, store the spectrum back in stack 10 and don't check (dc: ) for the existence of data already there.
-> rt 11; el 67_69; st 11 dc:          Eliminate channels 67 through 69 inclusive from stack 11.

```

The command `bc` is similar to `e1` except that it saves the eliminated channel numbers and eliminates them from any scan subsequently “gotten” with `gt`. These can be cleared with `bc cl`.

## 6.4 Folding Frequency-Switched Spectra

For frequency switched data, if the reference frequency falls within the band of the backend, you can fold the spectrum to double the effective integration time using `fo`.

```
-> gt dt;; li 1; rm                                Get data, remove baseline, print rms noise level
rms = 0.5407 system noise = 566.43
-> fo; li; rm
rms = 0.3636 system noise = 380.90
```

## 6.5 Do Loops

Most likely, you'll need to perform the same operation on many spectra. `do` loops make this easy. The simplest form of such a loop consists of `do` followed by an argument which specifies the number of repetitions of the loop, and a string of commands delimited by either curly brackets or by quotation marks (single or double). Curly brackets can be nested to any reasonable depth, quotes cannot be nested.

```
-> rt 10; do 100 {do 10 {el 67; fo; li 2; st a:dt;; rt &+1}; pl}
                                Eliminate bad channel, fold spectrum, and remove baseline. Do this
                                to 1000 spectra, plotting every 10th one. Store the reduced spectra
                                in directory two.
```

The conditional loops, `do-while` and `do-if-else`, are also available. In these cases, `do` evaluates an *expression* at the beginning of the loop and only executes the loop if the expression evaluates to non-zero.

```
-> rt &+1 t;; do i: {.test} {el 67; fo; li 2; st a:dt;} el: {p "No Stack" }
```

Normally, `rt` will return an error if you try to retrieve a stack which is empty or doesn't exist. The `t:` flag following `rt` sets the variable `.test` to 1 if the stack exists and is full and 0 otherwise, and prevents `rt` from returning an error. That is, an empty stack is simply skipped. Thus, the `do` loop command string is executed only if the stack exists and is full, else the string "No Stack" is printed. Null else statements are made by eliminating the `el:` flag. Similarly, a `do-while` loop will execute the command string only if the expression is true.

```
-> rt 10 t;; do w: { .numst < 1000 } {el 67; fo; li 2; st a:dt;; rt &+1 t;}
                                Execute the loop until stack number 1000 is retrieved.
```

The string `.numst` is an example of a stack header variable. These will be explained in §8.1.3. `Do-while` loops may also take an else clause. `do` loops with local variables as part of the control expression should use parentheses to delimit the local variable, *i.e.*, `do i: { (#1)=0 } { ... }`. For more on expressions, type `docp` at the `->` prompt.

## 6.6 Co-adding Spectra

To co-add (or simply add) spectra, use `co` (combine). `co` averages the data in two stacks using the weight of each channel. The result is always stored in stack 2. The default is to use stacks 1 and 2. So to co-add stacks 10 and 11:

```
-> rt 10; st 2
-> rt 11; co
```

Alternatively,

```
-> co 10 11
```

will accomplish the same thing, storing the result in stack 2. Using the `add:` flag will cause `co` to sum the stacks instead of averaging them. To subtract two spectra

```
-> rt 10;rs -1;st 2           Rescale stack 10 by -1 and store in stack 2
-> co 11 add:                 Add stack 11 to stack 2. Note this is the same as rt 11;co add:
```

`co` automatically checks for compatibility (e.g., position, LSR velocity, process type) of the two stacks. If any differences are found, you will be asked if you want to continue. To suppress this checking, use the `dc:` (don't check) flag.

Frequently, you will want to go through a stacks directory and co-add all spectra at the same position. `xf` (transfer stacks) will do this automatically. `xf` transfers stacks from the foreground directory to the background directory, combining all matching spectra into a single one before storing it in directory two. It will not combine spectra with the same scan number. You can set the “match parameters” such as positional tolerance, stack limits and center frequency.

```
-> ns 'SESTReduced'          Move SESTReduced to the foreground directory.
-> ns 'SEST.xf, dt:          Open up a new background directory.
-> xf ptol:.5 v:1           Transfer all stacks from SESTReduced to SEST.xf, combining stacks
                           within 0.5' of each other, and giving a brief report about the process.
                           The flag v: stands for “verbosity” and is an integer between 0 and
                           3, 0 giving no information about the transfer.
```

## 6.7 Redirecting Output

You can redirect anything that *COMB* writes to the standard output using `ro`. It is useful for generating files suitable for plotting with MONGO. The general form of this command is `ro options; command-string`, where the standard output of the *command-string* will be redirected according to *options*. The default option is to write the output to the file *comb.out* in your home directory. (You can change the file name with `ro fn:'filename'`). The standard output can be piped through any UNIX system call with `ro p:`, e.g., `ro p:"grep hlb"; cm ??`. Most of *COMB*'s diagnostics and prompts are written to the standard error and won't be redirected.

For efficiency, the file is not closed between successive calls to `ro`. This can cause confusion if you remove a file (which *COMB* still has open) with some other UNIX process, and then try to write to it again with *COMB*, since the file will still exist but not have a directory entry. (That is, `ls` will not list the file). To force *COMB* to close the file, you can either quit *COMB*, or use the `fn:` flag to redirect output to some other file. `ro t:` will close an opened file, truncate it to zero length, and open it again.

Normally, the carriage return at the end of a command line containing `ro` tells *COMB* to revert to the normal standard output (i.e., the terminal). The flag `q:` is used to force *COMB* to do this. It is used if you wish to turn `ro` on and off several times on a single command line (or in a macro), e.g., `ro; ...; ro q; ...; ro; ...`. Use of `ro q:` does *not* close the output file.

You can also redirect the standard input with `ri`.

If you are new to UNIX, you should familiarize your self with standard input, standard output, and pipes before using `ro`.

## 6.8 Obtaining Hard Copies of Plots

*COMB* supports the following types of hardcopy output devices: PostScript<sup>3</sup> language laser printers, Impress language laser printers and HP7580 pen plotters using `hpgl`. Appendix A gives details on how to let *COMB*

<sup>3</sup>PostScript is a registered trademark of Adobe Systems, Inc.

know the name of your printer. A plot is generated by issuing the command `hc` (hardcopy) followed by a flag indicating the first letter in the name of your printer. For instance, at Bell Labs, our PostScript printer is named “apost”:

```
-> hc a:          Print plot on the “apost” printer.
-> hc pf:         Send plot to a PostScript file (pf: ) in /tmp.
```

## 7 Making Maps

Two types of maps are supported in *COMB*: spatial-spatial maps made using `cm` (contour map), in which both axes are angular co-ordinates on the sky, and spatial-velocity maps (`vc`, velocity contour) in which one axis is velocity. A map made by either of these two commands is displayed and automatically stored in an image location which is analogous to a stack. Up to 5 images can be retained in memory at the same time. These can be recontoured or overlaid with `cp` (contour plot). In addition to maps made with `cm` or `vc`, any FITS image can be brought into *COMB* and stored in an image location using `af` (attach FITS).

### 7.1 Relative Coordinates

The first thing that is needed to make a contour map is a relative coordinate system to make it on. This is done with `rc`. Usually the coordinate system is centered on the object of interest and is parallel to Equatorial ( $\alpha, \delta$ ) or Galactic coordinates ( $\ell, b$ ). There is full support of Equatorial, Galactic, and user-definable coordinates.

The easiest way to make a relative coordinate system is to take it from stack 1:

```
-> ns 'SESTreduced'; rt 10
-> rc 'SEST' fs:          Make a relative coordinate system named 'SEST' with the same center
                        and offset type as that now in stack 1. fs: stands for “from
                        stack.”
-> rc sl:; rc nd:        Store these coordinates locally (sl: ) and name this coordinate sys-
                        tem as the default (rd: ) for this stacks directory.
```

The name of the relative coordinate system and the coordinates themselves are now entered in the file *.LCOORDS* (“local coordinates”) in your home directory, and the file *SESTReduced/default* now contains the name of the relative coordinate system. Coordinates in *.LCOORDS* are accessible only to you. Subsequent calls to `ns` will automatically use these two files to define the relative coordinate system. You can store changes to the coordinates of the named coordinate system using the flag `rl`: (replace locally). The flags `sg`: (store globally) and `rg`: (replace globally) will write to the file */usr/comb/.GCOORDS* (“global coordinates”). Coordinates stored in this file are accessible to all users.

### 7.2 Spatial-Spatial Maps

Normally, contour maps are made in the current relative coordinate system. *This is not a true projection*. You may specify a projection with `op` (options). Possible projections are orthographic, mercator, gnomonic, and polar.

```
-> ns '/usr/you/OrionA/12co'
Current relative coordinate system:
  SYSTEM NAME | CENTER COORDINATES | OFFSET TYPE
OriA         | rd(1950.0) 5:32:47.0, - 5:24:30 | cdra(') oddec(')
-> cm,30,-30,-40,40,61,81 vl:2,16 ci: ir:1.2 st:10 mlb:'OrionA T*dV V = 2 to 16 km/s'
```

Computing array for map  
 Getting stack values  
 Constructing map  
 Displaying map

The above command string asks *COMB* to compute a 60' by 80' integrated intensity map centered on the above relative coordinates. The first four arguments are the boundaries *in units of the current map offset type*, in this case arcminutes. The next two arguments are the number of spatial points on each axis at which to compute a value. In general, one or two times the beam sampling is appropriate. In this case, the map will consist of 4941 points. The next argument, marked by the label *vl:*, indicates the velocity limits of the integration. Interpolation type is given by *ci:* (cone interpolate) with an interpolation radius (*ir:*) of 1.2 arcminutes. The map is to be contoured at a step size (*st:*) of 10 K km s<sup>-1</sup> and a main label (other than the default) for the plot is specified. The resulting map is shown in Figure 4.

```
-> im mv:1,2                                Copy image 1 to image 2 so the subsequent calls to cm don't over-
                                         write it. mv: stands for "move."
-> cm vl:2,3                                  Recompute map, integrating between 2 and 3 km s-1. All other map
                                         parameters are kept the same, except the main label which reverts to
                                         its default string.
-> do 13 {cm vl:&+1,&+1}                       Step through the data cube, computing maps at 1 km s-1 intervals.
```

You may want to compute a map of some quantity other than integrated intensity. This is done with the label *m:* (macro). The macro label tells *cm* to use the value returned by the specified macro. You must already have defined the macro in either the stacks, local, or global macro files (See §8). For instance, suppose you want to compute a map of maximum temperatures.

```
-> cm m:'Tmax'                                Compute a contour map of values returned by the macro "Tmax"
```

The macro "Tmax" may look something like this:

```
Tmax - in,2,16 dp:: v .tmax
```

The macro "Tmax" does two things. First, it integrates the current stack between 2 and 16 km s<sup>-1</sup> (*cm* takes care of retrieving all the stacks). This assigns the value of the maximum temperature to *.tmax*, as well as assigning values to other variables associated with the integration. The *dp:* flag means don't print the results. Secondly, it takes the value of the maximum temperature returned by *in* and "puts it where *cm* can find it" with *v* (value).

### 7.3 Spatial-Velocity Maps

*vc* makes a spatial-velocity image (*e.g.*, an *l-v* diagram) along a line between two given points in the current relative coordinate system.

```
-> vc 0,-20,0,20 sp:1 ir:1 ci: vl:2,16 st:20 mlb:'OrionA 12co'
```

The above example makes a spatial-velocity contour map along the line between (0,20) and (0,-20). Note that the input order of the spatial limits is different than for *cm*. The data is interpolated onto points spaced by *sp:* in arcminutes. If the line is along a line of observations, *sp:* should be picked so that the plotting points are commensurate with the data points. The map is shown in Figure 5. *vc* needs an example stack in stack 1 to determine the number of channels on the vertical (velocity) axis of the map (just retrieve any stack if *vc* complains about this). It remembers the filter width in that stack and only uses stacks with that filter width.

## 7.4 Recontouring Maps with `cp`

You can recontour a map with `cp` without recomputing it. By default, it works on image 1.

-> `cp sc:[5,50,10,54,15,59]` Recontour image 1 using the specified contours.

The flag `sc:`, which also can be used in `cm` and `vc`, allows you to specify the contour levels and line types explicitly. The general form is `sc:[ level, line type, level, line type, ... ]`. The square brackets are required. A line type is given by a value between 0 and 99 and is coded by the two digits. The ten's digit controls the thickness with 0 thin and 9 thick. The units digit controls the dottedness with 0 most dotted and 9 solid, e.g., 50 is a dotted line of medium thickness, 54 is dot-dash line of medium thickness, and 59 is a solid line of medium thickness. This coding allows 99 different line types. See Figure 6.

The flag `ovl:` and label `cl:` allow you to overlay two (or more) images and specify cutoff limits for contour levels, respectively.

-> `cp cl:10,20 st:2` Contour only values between 10 and 20 with a step size of  $2 \text{ K km s}^{-1}$ .

-> `cp im:1 st:2; cp im:2 ovl:` Overlay image 2 on image 1

-> `lk 30,-30,-40,40 mlb:' ' hlb:' ' vlb:' '`; `cp im:2 st:10 ovl:`  
Display the look map and then overlay the OrionA map stored in image 2 on it. Look maps cannot not stored in an image location, so you must invoke `lk` each time. See Figure 7.

## 7.5 Reading/Writing FITS images

Any two-axis FITS image can be read from disk or tape into any of the image locations 1 through 5 using `af`.

-> `af ff:'mbm12.13co' im:1` Read a  $^{13}\text{CO}$  map of MBM12 from disk into image 1.

-> `af ff:'mbm12.100um' im:2` Read an  $100\mu\text{m}$  map of MBM12 from disk into image 2.

-> `cp im:1 st:2 mlb:' ' hlb:' ' vlb:' '` Contour image 1, leaving the labels blank so they are not drawn over each other in the subsequent overlay.

-> `cp im:2 sc:[4E6,15,8E6,35,1.2E7,55,1.6E6,75,2E7,95] mlb:'MBM12 13CO/100 micron' ovl:`  
Overlay the  $^{13}\text{CO}$  and  $100\mu\text{m}$  maps, using dashed contours for image 2. See Figure 8.

-> `sp fn:'MBM12.13co-100.sp'` Make a scatter plot file of image 1 vs. image 2, suitable for reading into MONGO

Similarly any image can be written to tape or disk with `wf` (write FITS), e.g., `wf ff:'/dev/rmt12' im:3`.

## 8 Advanced Processing: Macros

*COMB*'s true power and flexibility lie in the ability to allow you, the user, to construct your own macros. A macro is a small program which allows you to perform arbitrarily complicated operations without having to do much typing. Macros are kept in separate ASCII files which can be edited with the default editor. There are three types of macro files:



1. The global macros file (edited with the `cm g:`). This file resides in `/usr/comb/.GMACROS` and contains macros accessible to all users.
2. A local macros file which resides in your home directory in `.LMACROS` and is accessible to only you. It is edited with `cm l:`.
3. A stacks macros file associated with each stacks directory, e.g., `SESTreduced/macros`. This file is used for macros specific to the data in that directory and is edited with `cm st:`.

When a macro is referred to, either on the command line or by a another macro, *COMB* will search in these macros files until it finds a match. The order of the search is directory 1 stacks macros (if directory 1 is open), directory 2 stacks macros (if directory 2 is open), local macros, and global macros. *COMB* will stop searching at the first match and execute the macro.

A macro definition is simply a four-character macro name followed by the string of *COMB* command(s) which the name is to replace. For instance,

```
cmap - cm,30,-30,-30,30 ir:1 st:l vl:0,10 ci: mlb:'This space for rent.'; wf ff:'map.fits'
```

is a rather simple example. There *must* be at least 3 characters between the end of the macro name and the beginning of its definition. (By convention, we use [space][minus][space]). Macros, like commands, can have input arguments. Before we discuss these, we must first digress briefly on variables in *COMB*.

## 8.1 Variables

*COMB* has four types of user-settable variables: *global variables*, *global strings*, *macro arguments*, and *header variables*. In general, you will only want to set the first three types. Usually, the user is only interested in *reading* header variables, but occasionally may want to write to them.

### 8.1.1 Global Variables

The symbols #0 through #9 can be used to hold user-defined numeric constants. These are set these with either the `p` (print) or `c` (calculate) commands. They can be used wherever you would normally use a number.

```
-> c #1=25                                This is the same as p #1=25, except that it doesn't print the value.
-> p #1
25
-> c #=30                                  Note # is the same as #0
-> cm vl:#,#1                             Make a map integrated between 25 and 30 km s-1.
```

Expressions may be used on either side of the = sign.

```
-> c #(8-1)=3*4^5
-> p #7
3072
```

### 8.1.2 Global Strings

User-definable character strings are stored in symbols \$0 through \$9 (\$ is the same as \$0).

```
-> p $1='some string'
```

some string

-> cm vl:##,#1 mlb:\$1                      Same map with new main label

The command `pr` is a link to the C standard library routine `printf` and has a very similar form, *i.e.*, `pr conversion format, variable`. It can use the `printf` format conversion characters `%s`, `%e`, `%f`, and `%g`. There is a similar link to `scanf` via the command `sc`.

-> c #1=300

-> pr 'I have %g toes.',#1                      Print the string followed by the value of #1

I have 300 toes.

With the flag `gs:` (global string), the output of `pr` can be stored in any of the 10 global string locations, which can then be passed on as an argument to other commands.

-> pr 'I have %g toes.',#1 gs:3                      Redirect the output of `pr` to global string 3

-> p \$3

I have 300 toes.

-> rt 10;pl mlb:\$3                      Plot stack 10 with a silly main label

### 8.1.3 Stack Header Variables

Each stack is divided into blocks of 256 16-bit words; the first of these contains header information such as scan number, coordinates, and center frequency. The first 128 words of the header block contain 128 short integers. The second half of the header block contains real numbers and other constants. Each of these numbers has an associated variable name to which the user may refer when a particular header value is needed. You can access the header values of any of the core-resident stacks (*i.e.*, stacks 1, 2 or 3). For instance, `.ibsln` is the baseline order of the spectrum in stack 1, `.ibsln2` is that in stack 2, `.ibsln3` is that in stack 3.

-> rt 10; p .ra                      Print the right ascension of stack 10

2.89194

-> ph .ra                      Print the RA of stack 10 in HMS format

2:53:31.0

-> in,2,16                      Integrate the stack between 2 and 16  $\text{km s}^{-1}$ . This will assign values to `.tmax`, `.area`, and other variables associated with the integration.

Stack	From(Km/s)	To (Km/s)	Tmax	Area(K *Km/s)	Centroid(Km/s)	Peak
1743	2.000	16.000	65.791	357.921+/- 1.062	8.886	9.520

-> pr 'Tmax = %g T\*dV = %g', .tmax, .area gs:1; pl mlb:\$1  
Plot stack 10 with a main label indicating it's maximum temperature and integrated area.

-> do 100 {rt &+1; do i: {.ra<3:00} {in dp;;p .tmax nl;;p .area}; }  
Go through the first 100 stacks and, if the right ascension is less than 3<sup>h</sup>, integrate the spectrum, and print out its `.tmax` and `.area` on the same line (`nl:` ).

Note some variables, like `.tmax` and `.area`, will have zero value until the command which calculates them is called. You can assign new values the header variables with `c` or `p`. To save the new values, you must rewrite the stack with `st`.

-> rt 10; c .vlsr=20; st 10 dc:                      Change the LSR Velocity of stack 10.

The actual values of each channel of the stack are kept in the array `.stak`

-> p .stak(128) Print the value of channel 128

For a list of the names and descriptions of the stack header variables type `doch` at the `->` prompt.

### 8.1.4 Labelling Inside Plots

Now that we have introduced `pr`, `ph` and header variables, we can digress yet again on plot labelling. You can place labels inside spectra or maps using `gm` (graphics manipulation). (This is in addition to normal labelling outside plots using `mlb:`, `hlb:`, and `vlb:`). `gm` uses the label `ti:` (title) followed by the text you wish to place and the coordinates in current plot units of where you wish to place it.

-> rt 220;pl d: The `d:` toggle in `pl` is used to plot the spectrum with or without the header information. In this case, without.

-> gm ti:'DR21 CO(1-0)' 25,20 The label starts at  $V = 25$ , and is centered on  $T_A = 20$ .

You can also use `pr` or `ph` to pass values to `gm`.

-> ph .ra+.dra gs:1 Store the RA plus any offset in global string 1, using HMS format.

-> ph .dec+.ddec gs:2 Store the DEC plus any offset in global string 2.

-> pr 'RA = %s' \$1 gs:3; pr 'DEC = %s' \$2 gs:4 Put the RA and DEC in labels to be used for the plot. The labels are stored in global strings 3 and 4.

-> gm ti:\$3 25,18; gm ti:\$4 25,16 Place the labels on the plot. See Figure 9.

We could have used `cr` (cursor read) to tell `gm` where we wanted the labels. `cr` will read the position from the cursor and place the coordinates in the arrays `x()` and `y()`. The array indices start from zero.

-> cr 2 Read two positions from the cursor. The results are placed in  $x(0)$ ,  $y(0)$ ,  $x(1)$  and  $y(1)$ .

-> gm ti:\$3  $x(0)$ ,  $y(0)$ ; gm ti:\$4  $x(1)$ ,  $y(1)$  Place the labels at the read positions.

`cr` is also useful for getting positions from contour maps.

-> cr;rt rc: $x(0)$ , $y(0)$  Make one cursor read (from a contour map) and then retrieve the spectrum at that position.

Both `gm` and `cr` are useful for other operations. See the examples under `gm ?!`, `cr ?!` and `da ?!` (define area).

## 8.2 Macro Arguments

The symbols `!0` through `!9` represent macro arguments, where `!0` refers to the first argument, `!1` refers to the second, etc. Macro arguments do not have to be numbers; they can also be strings, expressions, global or string variables, lists, or ranges. Consider the macro definition

```
gfit - c #1=!0; do !1 {rt #1;pl vl::gf !2,!3 see::c #1=#1+1}
```

and the call

-> gfit,10,30,-4,4

The call to the macro `gfit` fits a gaussian profile to each of the spectra in stacks 10 through 30. First the global variable `#1` is given the value of first argument `!0`, *i.e.*, the first stack number to retrieve. This is necessary

because, unlike global variables #0 through #9, macro argument values cannot be set in arithmetic expressions, *e.g.*, `c !0=!0+1` is illegal. The `do` loop is executed 30 times (!1). Inside the loop, a gaussian is fit between  $-4$  and  $4$  (!2 and !3)  $\text{km s}^{-1}$ . The flag `see:` causes `gf` to overlay the fitted profile on the emission profile. Finally, #1 is incremented in order to fetch the next stack on the next iteration of the loop.

### 8.3 Some Useful Macros

The following is a typical stacks macros file, containing some macros used frequently at Bell Labs. As purely convention, we mark comment lines with a `%`. *COMB* does not distinguish between these lines and lines which actually contain a macro definition. The macros are individually explained at the end of the listing.

```
look - lk,.75,.5,-.15,.15      mlb:'SgrB2   CS Stacks'
base - us !0 st:c #1=!1; do !2 {rt #1;li !3;st #1 dc:c #1=#1+1}

% Movie plotting package
% #1 - File extension number.
% #2 - Beginning velocity.
% #3 - End velocity.
% #4 - Velocity step size.

init - p #1=0;p #2=-50;p #3=160;p #4=5;p #5=(#3-#2)/#4
sour - pr 'SgrB2' gs:1
file - pr '/usr/you/images/% s.% g', $1, #1 gs:2
labl - pr '%s CS(2-1) V= %g to %g km/s' $1,#2,#3 gs:3
step - c #2=#2+#4; c #3=#3+#4 ;c #1=#1+1

map0 - init:sour;labl;fil e;c m,.75,-.5,-.15,.15 3,3 7 ir:1 ci: vl:#2,#3 st:5 \
      mlb:$3 o:$1 ;wf ff:$2 im:1;c #3=#2+#4 ;c #1=#1+1
map1 - sour;do #5 {labl;file;cm vl:#2,#3 st:1 mlb:$3 o:$1 ;\
      step;wf ff:$2 im:1}
movi - map0;map1

map2 - init:sour;labl;cm,.75,.4 4,-.15,.0 6,1 08, 72 ir:1 ci: vl:#2,#3 st:5 \
      mlb:$3 o:$1 ;hc a:c #3=#2+#4 ;c #1=#1+1
map3 - sour;do #5 {labl;cm vl:#2,#3 st:1 mlb:$3 o:$1 ;\
      step;hc a;}
mapr - map2;map3
```

look

We usually define a macro called “look” for each stacks directory to make a look map specific to those stacks. This is so we don’t have to remember exactly what the spatial limits for the stacks are. In this case, the offsets are in degrees.

base

Baseline or re-baseline many stacks. The use-array is set to the range specified in !0. This removes a (!3)-order baseline from !2 spectra starting at the (!1)th stack. The stack is stored in its original location without asking if it is OK to overwrite the original spectrum. *E.g.*, `base, 40_140,10,2,1000` will remove a 2nd order baseline from the first 1000 stacks.

The next group of macros are used to make slices of the datacube and output to FITS files with names of the form required by the program `movie`, *i.e.*, `filename.0`, `filename.1`, `filename.2`... (see appendix B). The comment lines describe the use of global variables #1 through #4. You will want to edit some of these to tailor them to your own requirements.

<code>init</code>	Initializes the global variables. Edit this line to change the beginning and end velocities, and the velocity resolution of the maps/images.
<code>sour</code>	Source label for FITS header. This is stored in global string 1.
<code>file</code>	Global string 2 is used to store the name of the FITS file, gotten from global string 1, and its extension number, which is gotten from #1.
<code>labl</code>	Controls plot labels, which are stored in global string 3.
<code>step</code>	Increments variables #1, which is the file extension number, and #2 and #3 which are the limits of the velocity integration used in <code>cm</code> .
<code>map0</code>	Initializes the contour map parameters and produces a contour map on screen and a FITS image on disk in the directory specified by <code>file</code> macro. The <code>\</code> is a continuation character that tells <code>COMB</code> 's parser there is more of the command on the next line. The <code>\</code> must be the last character on the line, <i>i.e.</i> , no spaces after it. The <code>\</code> can also be used on the command line (at the <code>-&gt;</code> ).
<code>map1</code>	Repeat the map and image making for #5 iterations, <i>i.e.</i> , compute #5 separate slices through the data cube. #5 is set by the <code>init</code> macro. Note that only the parameters to <code>cm</code> which change from map to map are specified. All others are kept the same as in <code>map0</code> .
<code>movi</code>	Runs <code>map0</code> followed by <code>map1</code> .
<code>map2</code>	Same as <code>map0</code> except it sends the contour plots to the laser printer ( <code>hc a:</code> ) and does not write a FITS file.
<code>map3</code>	Same as <code>map1</code> except it sends the contour plots to the laser printer ( <code>hc a:</code> ) and does not write a FITS file.
<code>mapr</code>	Runs <code>map2</code> followed by <code>map3</code> .

Thus, after defining these macros in a macros file, all you need to do to compute the images for a 'movie' is  
`-> movi`

The macros are sufficiently flexible that you need only modify them slightly for different sources (data sets).

## 9 Figures

Figure 1 — A ‘look map’ made with `lk`. The location on the sky of a spectrum is marked with it’s ‘stack number’.  
Axis units are arcminute offsets from the central coordinate.

Figure 2 — *a)* A spectrum plotted with `pl`. The ‘tick grid’ and line type are changeable (see *e.g.*, Figure 3a).  
*b)* The spectrum with a first-order baseline removed. The horizontal line segments just above the velocity axis label indicates the portion(s) of the spectrum included in the baseline fit, known as the ‘use array.’

Figure 3 — *a)* A spectrum from the 12.5 kHz/channel backend. The line takes up too much of the band to remove a good baseline. It is plotted in histogram style with no tick grid (plot style: histogram). *b)* The spectrum at the same position taken simultaneously with the 250 kHz/channel backend. It can be used to compute a baseline for removal from the 12.5 kHz/channel spectrum (plot style: line).



Figure 3 (*cont.*) — *c*) Baseline removed from 250 kHz/channel spectrum. Same baseline is automatically removed from 12.5 kHz/channel spectrum. *d*) Overlay plot of the two spectra to compare the baseline fits.

Figure 4 — Contour map of  $^{12}\text{CO}$  in Orion A made with `cm`. The string in the lower left hand corner gives information about how the map was made. The contour step size is  $10 \text{ K km s}^{-1}$  between 10 and  $380 \text{ K km s}^{-1}$ . The data are smoothed over 1.2 arcminutes (`ir:1.2`) with a cone interpolation (`ci:`). Axis units are arcminute offsets from central relative coordinate (`rc:`).

Figure 5 — An spatial-velocity ( $\delta$ - $v$ ) diagram of Orion A made with `vc`.

Figure 6 — Same as Figure 5 but recontoured using `cp_sc:` (contour plot with specified contours) to show a few of the 90 different line types.

Figure 7 — An OrionA ‘look map’ indicating the position on the sky of spectra with a integrated intensity map overlaid on it, made using `lk` (look) and `(cp overl: )` (contour plot with overlay flag).

Figure 8 — Overlay of  $^{13}\text{CO}$  and IRAS  $100\mu\text{m}$  maps of the dark cloud MBM12, made using `af` (attach FITS) and `cp ov1`: (contour plot with overlay flag).

Figure 9 — An example of labelling inside plots using `gm` (graphics manipulation).

## Appendix A: How to Install COMB

*COMB* can be installed by porting the contents of */usr/comb* on a *tar* tape from Bell Labs to the new machine. The source code is recompiled by using the shellscripts `cleancomb` and `makecomb` in */usr/comb/bin*. You can only install *COMB* on a UNIX system. *Read this entire appendix before attempting installation.*

Here are the steps necessary to install *COMB*.

1. Create a directory in which to install *COMB*. Here we will assume it is in */usr*, although it doesn't need to be.

```
$ mkdir /usr/comb
```

2. Change to *comb* directory and extract *tar* file.

```
$ cd /usr/comb
$ tar -xv /dev/rmt12
```

3. If you are using *ksh*, edit your *.profile* file so that it contains the strings:

```
PATH=$PATH:/usr/comb/bin
COMB=/usr/comb
export COMB PATH
```

For *csh* or *tcsh*, edit the *.login* file:

```
PATH=$PATH:/usr/comb/bin
setenv COMB /usr/comb
```

4. *COMB* uses an editor for editing macro files and command lines stored in *ksh*-like history file (*\$HOME/.combhistory*). At Bell Labs, we use the *vi* editor, but *emacs* is also available to *COMB* if you prefer. To specify the editor of choice, edit your *.profile* (or *.login*) to contain the string

```
VISUAL=/usr/ucb/vi    (or emacs if you prefer)
export VISUAL
```

Again, for *csh* or *tcsh*,

```
setenv VISUAL /usr/ucb/vi    (or emacs )
```

5. Next, compile *COMB* using the utilities located in */usr/comb/bin*. First, the old object code must be removed from the *obj* subdirectory.

```
$ cleancomb
```

Then the compilation is done using the *makefile* in */usr/comb/bin*.

```
$ makecomb install
```

Compilation should take about 30 minutes on a SUN-4.



6. To attach a printer for hardcopy graphics output, you need to modify the file `/usr/comb/lib/hc.lpr` to contain the name of your printer. `COMB` supports three types of hardcopy output devices: PostScript language laser printers (`laser`, `aaser`, ...), Impress language laser printers (`imagen`, `jimagen`, ...), and an HP7580 pen plotter using `hpgl`. The shell script `hc.lpr` queues the plot file for plotting using the `lpr` spooler for the laser printers. Edit the shell script and after `PRINTER=`, put in the name by which the device is recognized in UNIX. The first character of the UNIX name is used in the hard copy command *e.g.*, `hc a:`. You can remove all the other unused names. If you have an HP pen plotter, its device name should be put into the character array `devname[]` in `/usr/comb/src/graphics/hpplot_c` before compilation.

Note that since `hc.lpr` is a shell script and not a compiled routine, you can change it without recompiling `COMB`. In fact, it is useful to run `COMB` in one window while changing the shell script in another until you get what you want.

7. The history mechanism may not easily work on operating systems other than SunOS or BSD 4.X. On such systems, it would be safest to first compile `COMB` without the `ksh` history mechanism. To do this you need to make two changes:

- In `/usr/comb/src/man/C.h`, change the line

```
#define HISTORY 1
```

to

```
#define HISTORY 0
```

- In `/usr/comb/bin/makecomb`, comment out the lines

```
for i in coordsys error graphics image main misc misc/libut misc/libedit \
    parse parse/entree scan stacks
```

and remove the `#` from the two lines below it, *i.e.*, “uncomment” the lines

```
# for i in coordsys error graphics image main misc misc/libut \
#     parse parse/entree scan stacks
```

If you have already run `makecomb` before making these changes, you must run `cleancomb` again, *i.e.*, repeat step 5 above.

## Appendix B: FITS Software

Facilities for viewing FITS images on a Sun workstation running SunView are located in the file */usr/comb/src/util/FITSView*. (You don't need *COMB* to run these.) A nice feature of these programs (written by Bob Wilson and Marc Pound) is that they can provide "full color" on an 8-bit color monitor by using a dithering algorithm (with 3 bits of red, 3 bits of green, and 2 bits of blue). The programs are called

**ql** quick look, display a single monochrome or pseudo-color FITS image

**rgb** rgb display of 3 FITS images simultaneously on an 8-bit monitor

**movie** Run a time lapse monochrome, pseudo-color, or 3-color movie from a stack of FITS images on disk.

**rgb24** rgb display of 3 FITS images simultaneously on a 24-bit monitor

These programs need to be compiled separately from *COMB*. Full documentation is provided in *README* and *FITSView.doc* in the *FITSView* subdirectory. They are also available in a shell archive package (*shar*) via e-mail.

## Appendix C: List of COMB Commands

<b>ad</b>	Add scans to stack 2
<b>af</b>	Attach a FITS file to an image
<b>bc</b>	Designate bad channels
<b>c</b>	Calculate something
<b>ca</b>	Calculate values from stacks
<b>cc</b>	Change center channel
<b>cm</b>	Space-space Contour Map
<b>co</b>	Combine two stacks, result in 1 & 2
<b>cp</b>	Contour Plot an image
<b>cr</b>	Cursor read
<b>da</b>	Define an area of an image
<b>dm</b>	Define macro
<b>do</b>	Loop through a command string
<b>e</b>	Execute a shell command
<b>el</b>	Eliminate bad chans in stack 1.
<b>em</b>	Empty a stack
<b>fl</b>	Flag location on graph
<b>fo</b>	Fold freq switched data in stack 1
<b>ft</b>	Fourier transform data in stack1
<b>gf</b>	Fit a gaussian function to part of a spectrum
<b>gm</b>	Graphics Manipulation
<b>gt</b>	Put scan in st 1
<b>hc</b>	Make a hard copy of the current screen
<b>im</b>	Image Manipulate
<b>in</b>	Integrate part of a spectrum
<b>is</b>	Interpolate a spectrun for a given position
<b>jb</b>	Calculate line widths from stack 1 (J. Bally's custom command)
<b>lc</b>	List commands li - Fit and remove a polynomial baseline
<b>lk</b>	Look at where stacks are
<b>me</b>	M ap data Extraction
<b>nf</b>	Switch data files
<b>ns</b>	Name stacks directory
<b>op</b>	Set options
<b>p</b>	Print something
<b>pa</b>	Pause in execution
<b>pd</b>	Print data
<b>pf</b>	Fit a parabola to part of a spectrum
<b>ph</b>	Print in hms format
<b>pl</b>	Plot stack 1
<b>pr</b>	Printf to standard output or a global string
<b>q</b>	Exit comb
<b>rc</b>	Define relative coordinate system
<b>ri</b>	Redirect command input
<b>rm</b>	Calculate rms and ssb noise figure for stack 1
<b>ro</b>	Redirect output to a file
<b>rs</b>	Rescale and add constant to stack 1
<b>rt</b>	Retrieve stack
<b>sc</b>	Scanf from a file or global string

- sl** Make a slice through an image
- sp** Make a scatter plot comparing two images
- sq** Squish - increase or decrease chan width
- st** Store stack
- tp** Total power - average chans in stack 1 weighted by cal in stack 3
- up** Update a stacks directory
- us** Change use array
- v** Compute value for map
- vc** Velocity Space Contour Plot
- vm** Calculate Virial Mass
- wf** Write an image to a FITS file
- wr** Write scan back onto file
- xf** Transfer stacks to directory 2 after making them unique (1 per position)

## Appendix D: For More Info

The authors of this document will be happy to answer any questions you have about *COMB*. It is possible, but discouraged, to write your own routines for *COMB*, *i.e.*, make a new command. You should be able to do any reasonable operation using macros. However, if you think you really need a new *COMB* command, consult Bob Wilson. If you think you've found a bug in *COMB*, report it to Bob Wilson. If you think you've found a bug in this cookbook, report it to Marc Pound.

Marc W. Pound  
Astronomy Program  
University of Maryland  
College Park, MD 20742  
Tel: (301) 454-3001

John Bally  
AT&T Bell Laboratories  
P.O. Box 400  
HOH L245  
Holmdel, NJ 07733-1988  
Tel: (908) 888-7124

Robert W. Wilson  
AT&T Bell Laboratories  
P.O. Box 400  
HOH L239  
Holmdel, NJ 07733-1988  
Tel: (908) 888-7120