MPSYS4

Version 4

Command Reference Manual

MARCO
Microanalytical Research Centre Commercial Organisation

Microanalytical Research Centre
School of Physics
University of Melbourne
VICTORIA 3010
AUSTRALIA
Fax: + 61 (0)3 9347 4783
Ph: + 61 (0)3 8344 5376
Email: marc@physics.unimelb.edu.au
Web: http://www.ph.unimelb.edu.au/marco

February 2002

Preface

This manual is a reference for the MpSys data collection and manipulation package. It contains:

an alphabetical listing of all the commands

information about command structures and syntax

a guide to the Skip programming language

The companion volume, the MpSys User's Manual, provides a guide to the use of MpSys to collect and analyse data using the commands listed in this manual.

MpSys version 3 provides a full graphical user interface to the functions of the program. The operation of the graphical user interface is described in the User's Manual. The purpose of this manual is to serve as a reference on the command line interface to MpSys.

The command line interface to MpSys provides and alternative method of accessing the functions of MpSys. Most of the commands available from the command line interface are more conveniently accessed from the graphical user interface. The command line interface exploits the full ability of MpSys to run user defined macros or other automatic procedures.

For further technical assistance please contact MARC via the following email address:

Limitation of Liability

Manual Version: 4.0
Manual Date: February 2002

Table of Contents

1. Starting MpSys

MpSys is started from the command line prompt of a Unix X-window. On some systems an icon on the desktop may be used to invoke MpSys. Several arguments may be added to the command to modify the characteristics of MpSys when it starts.

```
praxis:dnj/ldata/dnj/run1$ mpsys
```

2. On-line Help

The help files are also available on-line while using MpSys and are accessed by typing:

```
MpSys> help <topic>
```

Where `<topic>` is the capitalised topic name in the following listing. Please remember, however, to type the topic name in lower case letters to be compatible with MpSys and the UNIX operating system under which MpSys runs.

3. The Skip Language

Skip stands for Simple Kommand Interface Package.

Skip is a powerful command processing and programming language. It is written to be used as an interface between the user's keyboard and interactive programs.

Beginning users will find the command aliasing and the calculator useful. More advanced users and programmers may use the full programming language available. Skip is programmed using a syntax quite similar to 'C' and 'awk'.

Skip adds a great deal of power and flexibility to an interactive programs like MpSys with the Skip "front end" to the command processor.

3.1 Skip Features

Command aliasing

You may reduce the amount of typing you do, by defining "aliases" for common command sequences.

Example:

alias pr="print"

alias ssave='sort; save; print "Done"'

See ALIAS for more information.

Calculator

Skip has a "calculator" built in. The syntax is very similar to the 'C' programming language. You may define variables, assign values to them, and calculate mathematical expressions. Skip provides double precision numeric variables, string variables, and arrays of numbers and strings. You can use the Skip programming language to just do some sums for you, or you can feed values of variables and expressions to your commands. Numeric variables are defined with the "real" command.

Example:

```
real x, y, r
x = 3
```

```
y = 4
r = sqrt( x*x + y*y )
print “Plotting circle of radius “, r
plot circle radius=r
```

Input and output redirection

You may “redirect” the input or output of any command, or sequence of commands, to a file. The syntax is very similar to the UNIX shell input/output redirection. Unfortunately, you must place the input/output redirection at the start of the command. The UNIX shells allow you to put the redirection at the start or the end.

Example:

“logfile” print “Loading file “, file[i]

```
>>”logfile” load file[i]
>>”logfile” print “Done”
```

this could be more easily written by grouping the commands:

“logfile” {

```
      print “Loading file “, file[i]
      load file[i]print “Done”
              }
```

Online help for Skip

You may obtain help on any of the Skip features or commands by using the “help” command (UNIX users may want to “**`alias man=help`**”). Just typing “help” will print some introductory information, then walk you through a series of more detailed topics.

Example:

```
help
help print
help loops
help functions
```

You can obtain help on macros. “**`help macroname`**” will print any comments that Skip finds at the start of the macro. So, if you write macros, take the time to put a couple of explanatory comments at the start of your macro files. Comments are started by a “#”.

Example:

```
# loadfiles: loads files into the data buffers.
         #
         #     Usage: “loadfiles filenames”


         for (i = 0; i < $1; i++) {
                    load buffer=i file=\$I
               }
```

A programming language

Skip is a full programming language for interactive programs. The Skip programming language is similar to ‘C’ or ‘awk’. Skip provides conditional loops (while, for, do), conditional selection (if else), compound commands (command list inside curly braces “{ }”), and functions and procedures.

Example:

```
if (x < y) print “x < y”
else print “x >= y”
for (i = 0; i < 10; i++) {
  print “Loading “, file[i]
  load file[i]
          }


func sinc( x ) {
return sin( x ) / x
          }


plot sinc(x)
```

It may be most useful to use the Skip programming features in macro files and aliases. In this way you may reduce long, repetitive and complex command sequences to a single macro call.See FOR, WHILE, DO and IF in the Command Reference section below for more information.

There is more help available; take the time to have a browse.

There are a number of example macro files distributed with Skip. These are an excellent way to find your way around Skip. The macros demonstrate only Skip features and are not specific to any particular application. The application you are using may also have a “library” of macros.

4. Macros of MpSys commands

You may also place long sequences of commands in a file. Calling this “macro” file will execute the commands as if you had typed them at the command line.

Example:

```
MpSys> @loadfiles
MpSys> @printall
```

Macros may be supplied with arguments on the command line. The macros can use these arguments when they are called.

Example:

```
MpSys> @loadfile “red.data”
```

See the command MACRO for more information.

5. MpSys Command Reference

5.1 Command syntax

Command entries are in the form:

COMMAND (ABBREVIATION)

Some commands will have a ‘Usage’ section which shows the full syntax of the command and how it may be used at the prompt. Below is a list of syntax characters showing how the arguments to a command may be used.

[ ]	optional
A | B	use A or B
<xyz>	fill in name
<nn>	fill in numeric value
[window id]	the name of the window or if missing the window currently pointed to where appropriate in commands the name can include the wildcard character ‘*’ and thereby refer to multiple windows to operate on
[station id]	refers to one of the forms <snum> <snum> ... - list of stations <snum> - <snum> - range of stations or blank - all stations

5.2 The Commands

In this list, the most frequently used commands appear in a box for ready reference.

Some commands in this manual have been superceded by the functions of the graphical user interface.

These superceded commands appear here “greyed out”.

The graphical user interface should be used in preference to the superceded commands.

alias

Creates short-hand versions of longer commands or command sequences.

Examples:

```
alias pr=”print”
alias exit=”save; exit”
alias man=”help” readx=”read( x )”
```

Skip looks for an alias as the first word on each command line. If an alias is found the value of the alias is substituted on the command line. The ALIAS command works similarly to the “alias” command in the Unix Bourne shell.

Use the UNALIAS command to remove alias definitions.

break

Break out of a FOR or WHILE loop. The BREAK command can be used any where within the body of a Skip loop and will terminate the innermost WHILE or FOR loop.

Example:

```
while (i < 100) {
     if (name[i] == “”) {
       print “Error: no string found at name[”,i,”]”
       print “Stopping...”
       break
                    }
       print “name[”,i,”] = “, name[i]
                 }
```

See Also: FOR, WHILE

buffers (buf)

Displays details on the current data objects:

raw data; sorted data; spectra station buffers; maps; extracted spectra

calibrate (cal)

This command has been superceded by the gui.

```
calibrate ?
calibrate [window id] -m  val1  val2  <label>
calibrate [window id] -el el1   el2   <label>
calibrate [window id]     val1  val2  <label>
```

Set a calibration for a spectrum currently in the specified window.

Options:

m	the markers X0 and X1 are set to values val1 and val2 given and the conversion factors a and b are calculated from this.
el	search the xray database for the energy (in keV) of the elements named and assumed to be marked by X0 and X1 and the conversion factors calculated from this otherwise the values given are taken to be the conversion factors

label (or units) must be supplied - it will be displayed in the window

The -el option will only be accepted for units of keV

Using the ? argument will display the calibration status and conversion factors for each spectra when set

Note: X(calibrated) = a * X(uncalibrated) + b

cd

`cd <directory name>`

The CD command is used to change the current working directory.

clear

clear <window spec>

Clear the specified windows.

clmarkers (clm)

`clrmarkers [window spec]`

Clear all markers for spectra in the window specified if no windows named clears all station spectra markers.

It will set X0 and X1 to default minimum and maximum settings.

close

Stop this run and close down data collection.

See also:

`close, run, stop`

cls

`cls <colour scale filename>`

Remaps the current colour map according to the ‘‘cls” file (RGB values) specified. It applies immediately to all maps.

continue

The CONTINUE command may be used to short-circuit the body of a WHILE or FOR loop. When a continue command is encountered within the body of a Skip loop, the current iteration of the loop is halted, and if the condition is still satisfied, the next iteration of the loop is started.

Example:

```
for (i = 0; i < 100; i++) {
if (name[i] == “”) {
            print “No name found at record number “, i
            print “Continuing to next record...”
            continue
            }
print “name[”, i, “] = “, name[i]
      }
```

contract

`contract <window spec> <multiplier>`

Contract viewport of spectra window specified by exp_contract ratio [* multiplier]

ctool

The colour tool is used to modify the intensity scale of a map. It allows non-linear mapping of the false colour intensity scale into the actual number of counts. The command pops up a new window containing a line representing the linear mapping of the colour scale to the number of counts per pixel in the map. Right clicking the mouse button adds a handle any where along this line and left clicking on this handle allows it to be dragged to a new position. More than one button may be added. This tool is particularly useful for bringing out faint detail in maps.

```
ctool       [on|off] [-r] [-v] [-g] [-l] [-s filename]
            [-a|-x|-m x=<val> y=<val> ox=<val> oy=<val>]
```

Many option control the creation of the colour tool

on | off	turn on or off respectively - default is on

-r	restore colour tool to initial state
v	reverse colour tool
g	turn on/off grid showing colour coordinates
l	turn on/off colour coordinate labels
s <filename>	save the current colour table as a colour scale file
a | -x x=<val> y=<val>	add/delete the control point closest to coordinates given
m ox=<oldval> oy=<oldval> x=<newval> y=<newval>	move the control point closest to coordinates given to the new coordinates given

Note: coordinates are to be in colour coordinates [127,0] first and last point exempted from deletion or horizontal movement

While the pointer is in the colour tool window the mouse buttons have the following functions:

Button 1:	move nearest control point - display coords (in colours) won't move first or last off the margins and wont move any other point to left or right of point either side
Button 2:	Add a control point (control handle)
Button 3:	Delete nearest control point (exempting first and last)

The histogram is applied to the colour table directly mapping old colour (horizontal scale) to new colour (vertical scale) with a range [127,0] in both cases.

Note that the control points are in pixels and the window geometry is arbitrary but when calculating colours the nearest colour grid point [0-127] is used - this is displayed in the top right hand corner as you move a point.

disable

```
disable <station number> [station number ...]
```

Disable the specified stations from collecting data this is the default station state on startup.

do

The DO WHILE command follows the syntax of the "C" programming language (except that a newline is a valid statement delimiter). DO is used to generate command loops in Skip (see also WHILE and FOR). The program will keep re-executing "statement" until the condition becomes false. See "condition" for more information. The "do" command differs from the "while" command only in that the condition is tested at the end of the loop (not at the start. Thus the body of the loop is guaranteed to be executed at least once.

Example:

The following example will read values into the array "x" until a value less than zero is entered:

```
do read( x[i++] ) while (x[i] > 0)
```

The BREAK command may be used to break out of a do loop from within the loop. The CONTINUE command may be used to short-circuit a do loop. When a continue statement is encountered inside a loop, SKIP jumps out and starts the next iteration of the loop.

See also: **`break, continue`**

down

```
down [window spec] [multiplier]
```

Move viewport of spectra window down by up_down ratio [*multiplier]

enable

`enable <station number> [station number ...]`

Enable the specified stations to collect data.

erase

`erase       <-u|-s|-m> [window spec]`

Erase the buffer specified.

u	unsorted events buffer
s	sorted events buffer
m	map in window specified

exit

Exit the MpSys program and return to the unix shell.

expand

`expand      <window spec> [multiplier]`

Expand viewport of spectra window specified by exp_contract ratio [* multiplier]

extract

`extract     [?] [window=<name>] [-a] [-s] [stn=<n>] <buffername>`

Extract energy spectra from shape in map and place in extract buffer with name given, wrapping around if number of buffers created exceeds MAX_EXTRACTION_BUFFERS.

Save the spectra buffer created to buffer_name.img and save the shape used to buffer_name.shp

Display in plot window with the same name (creating one if necessary)

window=<name>	Extract spectrum from window <name>. Default is the current window.
a	Colour shape as we extract data
s	Colour each pixel from which an event was extracted
stn=nn	Extract data from the named station. (default is the current station in map window)
?	Display list of current extraction spectra buffers

Report on the total number of events extracted and the area of the shape both in channels and as a percentage of the current map area saving this information with the window (see WINFO)

fnkeys

`fnkeys [<function key> <command|macro>]`

Remaps function key to new command or macro.

With no arguments displays macros/commands currently mapped to function keys.

Example:

```
fnkeys F12 menu
```

Remaps function key F12 to issue the ‘menu’ command

for

The FOR statement follows the syntax of the “C” programming language (except that a newline is a valid statement delimiter. FOR is used to generate command loops in Skip (see also WHILE).

Usage:

```
for ( initialise; condition; increment ) statement
```

where “initialise” and “increment” are any valid SKIP statements. A FOR statement is equivalent to:

```
initialize;
while (condition) {
  statement
  increment
        }
```

See WHILE for more information.

Examples:

```
define x[100]
for (i = 0; i < 100; i = i + 1) print x[i]


for (i = 0; i < 20; i++) {
print “Next value > “
read( x[i] )
        }
```

Note: the newline character is a valid statement delimiter in SKIP (unlike “C”), so you must include the statement on the same line as the FOR statement, or use braces. i.e.. the following statement is not valid...

```
for (i = 0; i < 100; i++)
sum += x[i];
```

func

The FUNC statement is used to define SKIP procedures. Functions are equivalent to functions in FORTRAN or “C”. SKIP functions take an optional argument list.

Usage:

```
func name ( arg1, arg2,... ) statement
```

or:

```
string func name ( arg1, arg2,... ) statement
```

where the optional arguments are of the form:

name	defines a numeric argument
string name	defines a string argument

name[]	defines a numeric array argument
string name[]	defines a string array argument

SKIP reads the function definition, and converts the “statement” into an internally executable form. You can then call the function “name” by entering the command:

```
name( args,..)
```

Functions can return a value to be used in expressions. You can pass back a value using the “return” command. NOTE: functions can call themselves recursively (see examples).

Examples:

the following code calculates factorials:

```
          func fac( i ) {
          if (i > 1) {
          return i * fac( i - 1 )
     } else {
       return 1
               }
               }


       n = 6
       print n, “! = “, fac( n )
```

and produces the following output:

```
6! = 720
```

NOTE: the “undefine” command can be used to delete function and procedure definitions. After “undefin”ing a function or procedure you can re-define it.

identify

Give a list of those elements within en_width (see ‘set’) of where the pointer is ,in a window containing a calibrated energy spectrum display the most likely element on the marker drawn on the spectra CURRENTLY the ‘most likely’ is simply the closest the graphics drawn are temporary so use the ‘redraw’ command to erase

if

The “if” statement follows the syntax of the C programming language (except that a newline is considered a statement delimiter). “if” is used to produce conditional execution of some command sequence.

Usage:

```
if (condition) statement
```

or:

```
if (condition) statement
else statement
```

In the first form, the statement will be executed if the condition (in brackets) is true. In the second form, the first statement will be executed if the condition is true,else the second statement is executed.

Example:

```
if (x > 45) x = 45
if (s == “hello”) {
printf “s = %s\n”, s
s = “goodbye”
} else {
printf “Still going\\n”
                }
```

Note: the newline character is a valid statement delimiter in SKIP (unlike “C”), so you must include the statement on the same line as the “if” statement, or use braces. i.e.. the following statement is invalid...

```
if (y < 0)
print “y is negative\\n”
```

kill (k)

`kill <window id>`

Close the specified windows.

left

`left [window id] [factor]`

Move viewport of spectra window specified by left_right ratio multiplied by factor

list

The “list” command is used to list the names of commands, variables, functions, etc.

e.g..

`list functions`

will print a list of all the defined Skip functions.

You can also use list to get more information about a particular command, variable or function name.

e.g..

`list path`

will tell you that “path” is a pre-defined string variable and print it’s current value.

The “help” command is used to get more detailed information on a particular topic. The output from the list command is intended to be brief and informative.

load

`load [-switch] <filename> [window spec]`

Load buffer(s) specified by switch from specified file possible switches are : window=”name” For use when loading maps. Use this option to load a map into window “name”. Default is the current window

u	unsorted events buffer

s	sorted events buffer
m	map into window specified
e[i]	energy spectra for station i (i.e. e1, e2, ...)
x[i]	x spectra for station i
y[i]	y spectra for station i

[i] is an optional station number - defaults to 1

i	e x y spectra into station i
a	e x y spectra for all stations
ext	extract spectra (into next extract buffer) giving it the name of the file, creating plot window and displaying spectra in it (default) e x y spectra into station 1

Note on file sizes (expected)

e spectra	32K
x or y spectra	16K
e x y spectra for single station	64K
e x y for all 4 stations	256K

login

`login [macro name]`

Executed on login if no login macro supplied by user in command line or available in local macros directory

map

`map {arguments}`

Generate a two dimensional map from event data into window specified with switches and parameters:

m	load E X and Y windows from X0 and X1 markers
set	set default parameters only - don't draw a map
?	print current map parameter settings
window=name	draw map in the named window
el="name"	set energy window from name's characteristic X-ray peak (assumes the E spectrum has been calibrated)
stn=nn	set station
elo=nn	set lower limit of energy window
ehi=nn	set higher limit of energy window
xlo=nn	set lower limit of X window
xhi=nn	set higher limit of X window
ylo=nn	set lower limit of Y window
yhi=nn	set higher limit of Y window
col=nn	set number of colours used in map
bin=nn	set psuedo pixel size for map
bg=nn	set background number of counts per pixel (subtracted)

type="<name>"	name of mapping function to use, current choices:	
	intu	intensity maps from unsorted data
	int	intensity maps from sorted data
	av	for average energy maps from sorted
	med	median maps from sorted data
	mom	moment maps from sorted data
order=nn	order for moment maps	
min=nn	map min count given to lowest colour	
max=nn	map max count given to highest colour	
auto="on/off"	if on set scale colour scale to min \& max values in map	

If the corresponding spectrum is calibrated, the window limits should be specified in calibrated units - otherwise in channels.

NOTE: the order of arguments is significant. e.g.. "map -m stn=2" will load the xlo, ylo, etc values from the markers of the current station, but generate the map from station 2. In most cases it is more appropriate to type "map stn=2 -m".

mapwindows (mapw)

`mapwindows [window id]`

Make any windows that have become invisible visible.

markers (mark)

`marker [e|x|y|extr_name] [station id] [X|Y]`

Display the marker positions in uncalibrated (and where possible calibrated) units and show ranges

if spectra type is omitted it will use type of current window

if marker type is omitted it will display the X markers

if station number is omitted it will display station one

modify

`modify [parameters] <window name>`

where "windowname" is the name of the window to modify (default is the current window), and parameters may be any of the following:

?	Display the current default parameter settings
name="wname"	The new window name
xo=xx	Position of left side of window on screen (pixels)
yo=xx	Position of top of window on screen (pixels)
sz=xx	Height of window on screen (pixels)
as=1.35	Aspect ratio (Height / Width) of window
stn=n	New station number for the window
vd={0,1}	This switch toggles the foreground and background pixels
tune	Set the window type to "Tuned"
map	Set the window type to "Map"

plot	Set the window type to “Spectrum”
m	Use spectrum markers to set the window size ratio. Set the channel/pixel ratio when using “-m” option.

The last two options are for ensuring that maps do not suffer from aliasing when they are rendered on the screen.

mpsort

`mpsort [-n filesize] <filename>`

External program which process event files into sorted/indexed form allowing (easier) shape extractions and average energy mappings takes expt.evt file and produces expt.sd and expt.sp if sorting more than 1Mbyte of events user must specify size with -n switch

Example:

```
mpsort -n 3000000 data_file
```

new

`new [parameters] {arguments}`

Create new window(s) with the specified characteristics a name that would be a duplicate is ‘incremented’ by one see also the modify command

Arguments:

plot, tuned, map	Specifies type of window to create
parameters :	
n = nn	number of windows
name= “name”	window name
xo= nn	X position of top left hand corner in screen pixels
yo= nn	Y position ,, ,, ,, ,,
sz= nn	height of the window in screen pixels
as= nn	aspect ratio (width/height)
vd= nn	video type forward - positive negative - reverse

pallette (pal)

`pallette [window spec] [-h] [-r] [num label]`

Create a pallette showing the ranges for the colours in the map window specified

by default vertical - but horizontal if set ‘-h’ switch

by default the max value is drawn at the top (or on the right if horizontal)

reverses sense if set the ‘-r’ switch

by default 2 rectangles are labelled - changeable by setting num_label

The pallette can be resized and moved as required but the colour information must be a multiple of 128 lines (ie num colours)

`pallette [window spec] num_label`

will just change the number of labels used in this case need to point to or name the pallette window itself

pointer

Displays coordinates of mouse pointer (X, Y) in pixels from top left hand corner of screen mostly useful in developing macros and for fine control in wcopy in combination with ‘warp’ command.

print

The “print” command may be used to print value of variables, expression, and strings. The print command takes a list of arguments, separated by commas, and prints the value of each. The print command produces loosely formatted output. Use the “printf” command to format your own output.

Usage:

```
print arg1, arg2,...
```

Examples:

```
print 34 * 45
print “x = “, x
print “data[”, i, “] = “, data[i]
print “name = “, name
```

NOTE: typing an expression on a line by itself will cause it’s value to be printed. e.g.. the line:

```
6 * 6
```

will produce the output:

```
36
```

This provides a short cut to using the print command.

You can “redirect” the output from the print command (or any other command) with the special SKIP output redirection symbol, “>”.

Example:

```
>”file1” print x
```

will write the value of x to a disk file called file1. This redirection is similar to input/output redirection used by the UNIX operating system. See “input” or “output” for more information.

printf

The “printf” command is used to print output in a precisely formatted manner. The formatting specifications are similar to the printf function in the “C” programming language. Any book on “C” will give you a complete description of the formatting style.

Usage:

```
printf format_string, args, ...
```

The format_string is a string expression which specifies the manner in which the arguments are to be printed.

Examples:

```
printf “x = %g\\n”, x
printf “Loading data from file: %s\\n”, filename
```

NOTE: the PRINT command may be used to produce loosely formatted output. The print command does not require a format string and will be adequate for most uses.

proc

The “proc” statement is used to define SKIP procedures. procedures are equivalent to procedures in FORTRAN, or void functions in “C”. SKIP procedures take an optional argument list, and executes a sequence of statements.

Usage:

```
proc name ( [arg1, arg2,...] ) statement
```

where the optional arguments are of the form:

name	defines a numeric argument
string name	defines a string argument
name[]	defines a numeric array argument
string name[]	defines a string array argument

SKIP reads the procedure definition, and converts the “statement” into an internally executable form. You can then call the procedure “name” by entering the command:

```
name( args,..)
```

Examples:

The following commands:

```
proc printxy ( x, y ) printf ”x = %g, y = %g\\n”, x, y
printxy( 3, 4 )
printxy( PI, 1/3 )
```

will produce the following output:

```
x = 3, y = 4
x = 3.14159, y = 0.333333
```

The following command define a procedure to read an array of data, then calls that procedure to read the data:

```
proc readdata( x[], n ) {
define i
i = 0
while (i < n)
read( x[i++] )
          }


define data[100]
readdata( data, 100 )
```

SKIP “func”tions are like procedures, except they return a value (numeric or string), and may be used in numeric or string expressions. See “func” for more information.

quit (q)

Exit the MpSys program and return to the unix shell.

read

The “read” command reads the value of a variable or string from the standard input (usually the keyboard). It returns a status value which may be tested to see if an error occurred during the read. “read” can be used to read a value into a string or numeric variable.

Usage:

```
read( x )
```

The values returned are:

1	the read worked ok
0	an error occurred while trying to read the value.

Examples:

```
define string s
read( s )


define x
if (read( x )) print “x = “, x
else print “Error reading x”
```

NOTE: you can “redirect” the input of read, using the input redirection symbol, “<”. eg. <”file1” read(x) will read a value from the disk file file1, into the variable x. Input/output redirection is similar to the UNIX operating system. See “input” or “output” for more information on input/output redirection.

real (string)

The “real” and “string” commands are used to create new variables to be used by SKIP. You can create numeric, string, numeric array, and string array variables.

Usage:

```
real name, name, ...
string name, name, ...
```

Examples:

```
real x
string name
real x, y, z
string s, d


real data[100]
string files[30], x, y
```

You may use the real and string commands any where outside a set of braces (“{” and “}”), but you may only use them as the first commands inside a pair of braces. If used inside a set of braces, any variables created are “local” to that set of braces, ie. they can only be used by commands within that set of braces.

Example:

```
real x
x = 23
         {
real x
x = 5
print x
         }
print x
```

This example will print two numbers. The first print command will produce a “5”, but the second will produce “23”. The new “x” created within the braces is only within the braces, and is quite distinct from the “x” defined outside the braces. Note this means that once we define “x” inside the braces, there is no way to access the “x” defined outside the braces.

All real variables are initialised to 0.0 when they are created, and string variables are initialised to the empty string.

You may create static variables within braces using the “static” modifier. Like local variables, static variables are only accessed within the set of braces in which they are defined. However, the value of static variables is retained between loops and repeated function calls.

Example:

```
proc dostart () {
static real init
  if (!init) {
                  .        \# initialise
                  .
                  .
                       init = 1
                  }
                  .
                  .
                  .
                  return
            }
```

help static for more information.

redraw

redraw [window id]

Redraws the specified window, thus erasing any temporary features ie. shape shadings, partial shapes and zoom boxes.

return

The “return” command is used to escape from a “proc”edure or “func”tion.

Usage:

return

or:

return argument

The first form is used in procedures. The second form is used by functions. The argument supplied is the return value for the function. This value can be used in calculations by SKIP.

See FUNC and PROC for more information about functions and procedures.

reverse (rev)

reverse [window id]

Reverses the video type of the specified windows.

right

right [window id] [multiplier]

Move viewport of spectra in display right by left_right ratio [* multiplier]

run

run [{options}] <experiment name>

Sets up for a new experimental run. This creates spectrum and unsorted data files with names derived from the given experiment name.

Options:

Q	Perform a charge transient experiment.
l	Specifies “local” mode. In this mode mpsys will read an event-by-event file from the local disk.

See also:

close, start, stop

save

save [options] <filename> [window id]

save -ext <buffer name> <filename>

Save the data buffer specified by switch into the file specified.

Possible switches are:

window=”name”	For use when saving maps (-m switch). By default, mpsys will save the map in the current window. Use this option to save a map in window “name”
m	Map buffer from window named to file named if no window name use current window if no file named , name file using window name
e[i]	energy spectra for station i (ie. e1, e2 ...)
x[i]	x spectra for station i
y[i]	y spectra for station i
i	e x and y spectra for station i
a	e x and y spectra for all stations
ext	extract buffer named into file named if not given file name equals buffer name if no names uses extract buffer corresponding to

window currently pointed to (default) e x y spectra into station 1

Note on file sizes:

e spectra	32K
x or y spectra	16K
e x y spectra for single station	64K
e x y for all 4 stations	256K

set

Lists the available control variables and their current values first string variables , then integer then floating point variable for a list of the control variables and their meaning see ‘set_list’

`set <variable name> ?`

reports the current value of the control variable

`set <variable name> <new value>`

sets the control variable to the new value specified

`set [window spec] MARKER_ID new value`

reset the marker specified (eg X1 , Y9) for the spectra in the specified window to new value a value below 0 or above MAX means set it undefined except for X0 , X1 where it means reset to min and max values if the spectra is calibrated the value is in calibrated units otherwise it is in channels

setcolor (setcol)

`setcolor [window spec] <fg|bg> <colour name>`

Reset the background (bg) or foreground (fg) colour of the windows specified to the colour named update the global foreground or background colour name to be used in subsequent window creations

setopt

Set clear various MpSys global options.

The options are:

echo	enable/disable echoing of input when running macros
log	enable/disable logging of typed in commands to a “log” file
alias	enable/disable the use of command aliases.
logfile	set the name of the “log” file

Examples:

`setopt echo`	enable echoing
`setopt noecho`	disable echoing
`setopt echo=on`	enable echoing
`setopt echo=off`	disable echoing
`setopt alias=on`	enable aliases
`setopt noalias`	disable aliases

setopt log	enable logging
setopt nolog	disable logging
setopt log=on	enable logging
setopt log=off	disable logging
setopt logfile=”filename”	set the “log” file name

The echo and log options are disabled by default, and the alias option is enabled.

The echo option is useful if you want to see what your macros are doing when you run them. help macros for more information.

The log option saves all the commands you type in a disk file. This is useful just as a record of your work, and also to create macro files ‘on the fly’. So you can save a sequence of commands as you type them in. You can then re-execute those commands by calling your “log” file as a macro.

Examples:

```
setopt logfile=”macro1” log=on
                                        .
                                        .
sequence of commands....
                                        .
                                        .
setopt log=off
@macro1                                              this will re-run
  your commands
```

The logfile option may be set to any valid Skip string.

Example:

string name	creates a string variable called name
name=”log1”	
setopt logfile=name	logfile == “log1”
setopt logfile=”log2”	logfile == “log2”

Type help strings for more information.

shape

shape [options] [filename]

Manipulate shapes (multiple polygons) drawn using B2 and B3.

Options:

window=”name”	Manipulate shape in window “name”. Default is current window.
x	erase shape in map window (no file name)
s	save shape in map window to file
l	load a new shape. Deletes any existing shape in the window
a	add a new shape to the current shape in the window

see ‘buttons’ for how to draw the shape

show

`show [window spec] [-x] el_name`

Show location of elements characteristic xray lines on E spectra only if spectra is calibrated.

Note: input is case insensitive but need use '_' for space

the -x switch is used to erase an existing element picture 'show -x *' to erase them all

relevant settable variables are:

min_show_size	minimum line length as a fraction of window height
max_show_size	maximum line length as a fraction of window height
show_gap	height of most intense line above curve in pixels (pos or neg)
show_dir	direction of lines (>=0 - upwards < 0 - downwards)

showrange (showr)

`showrange [window id] el_name`

Show width of elements first xray line (only if spectra is calibrated) (e first in standard data base) in E spectra using control variable 'en_width'. The line is displayed as a line above and below the value with total width en_width.

These are also shown as part of map or sum cmd with the 'el' option.

The display is not permanent to erase simply type 'redraw'

spectrum (spec)

`spectrum [options] <buffer name>`

Load a spectrum into the plot window.

Options:

window="name"	Load spectrum into window "name"
lin	Plot with a vertical linear scale
log	Plot with a logarithmic vertical scale
a	Add the named buffer to this window

buffer name specifies the spectrum buffer to load into the window.

Examples:

`spectrum y2`	load the Y spectrum from station 2
`spectrum -a e1`	add the E spectrum from station 2

Adding spectra to a window (-a option) cause multiple spectra to be displayed in different colours in the window.

stall

`stall [public macro]`

Sets up a small map and E,X,Y spectra windows for each of the stations.

start

Start or restart data collection on all enabled stations.

See also:

`close, run, stop`

static

You may create static variables in loops and functions using the “static” modifier. Normally local variables are re-created and re-initialised each time execution enters a set of braces within a loop or function. Static variables retain their values between different invocations of a function.

status

Display status of data acquisition framework.

stop

Stops (suspends) data acquisition. Data collection may be re-started with the “start” command.

See also:

`close, run, start`

string (real)

See REAL.

sum

Sums the counts between markers in a spectrum window.

`sum [window id] ch1 ch2`

`sum -m <m1> <m2>`

`sum -el <name>`

`sum -a`

Sum data in the spectra in the window specified the sum includes the first and last channels specified

ch1 ch2	between the channel numbers specified ‘:’ can stand for either first or last channel
m <m1> <m2>	between markers specified
el	within en_width of xray energy for element named
m	between X0 and X1 markers in window specified
a	between all active markers

switch (swi)

`switch [window id]`

Makes window window id the current window ie default in commands a button press in any non current window has the same effect

switch -all	switches to all windows in turn

text

`text [window id] [ X0 Y0] [-x] [-v] <text string>`

Draw text string in specified window at specified coordinates if no coordinates given use current pointer location

Options:

v	switch draw it vertically
x	delete the string - by coordinates if given or by the text string if given or the string closest to the pointer location

See also: **`font`**

top

`top [window id] [value]`

Rescale spectra in plot window specified to maximise displayed range or change top range to value given no effect in either case unless spec_auto is “off”

tune

`tune [window id] [options]`

Set up a two dimensional tuned map (ie. updated as data is collected) in the specified window with the specified argument settings

Options:

?	display current tuned parameter settings
-m	load E X and Y windows from X0 and X1 markers
el=”name”	set energy window from name’s characteristic x-ray peak assumes the E spectrum has been calibrated
stn=nn	set station to number nn
elo=nn	set lower limit of energy window
ehi=nn	set higher limit of energy window
xlo=nn	set lower limit of X window
xhi=nn	set higher limit of X window
ylo=nn	set lower limit of Y window
yhi=nn	set higher limit of Y window
col=nn	set number of colors used in tuned map
bin=nn	set psuedo pixel size for map
bg =nn	set background number of counts per pixel (subtracted)

unalias

The “unalias” command is used to remove alias definitions. “help alias” for information on aliases.

Usage:

`unalias name {name ...}`

Example:

`unalias man pr readx`

will remove any predefined aliases for “man”, “pr” and “readx”.

uncalibrate (uncal)

`uncalibrate [window id]`

Turn off calibration for the spectra in the specified window or name the spectra buffers eg. to uncalibrate all spectra (in plot windows or not) use ‘uncal *’

unmapwindows (unmapw)

```
unmapwindows [window id]
```

Unmap (make invisible - but retain) specified windows.

unzoom

```
unzoom [window id]
```

Return display to original window settings in window specified whether it contains a map or spectra that has beem zoomed.

up

```
up [window id] [multiplier]
```

Move viewport for spectra window specified up by up_down ratio [* multiplier].

while

The WHILE statement follows the syntax of the “C” programming language (except that a newline is a valid statement delimiter). WHILE is used to generate command loops in Skip (see also FOR).

Usage:

```
while (condition) statement
```

The program will keep re-executing STATEMENT while the condition remains true. See “condition” for more information.

Examples:

```
i = 0;
while (x[i] > 0) print x[i++], “\\n”


while (y > 0) {
  print “Enter number (-1 to quit) ? > “
  read( y )
  print y, “\^2 = “, y \* y
}
```

The BREAK command may be used to break out of a while loop from within the loop. The CONTINUE command may be used to short-circuit a while loop. When a continue statement is encountered inside a loop, SKIP jumps out and starts the next iteration of the loop. See BREAK and CONTINUE.

Note: the newline character is a valid statment delimiter in Skip (unlike “C”), so you must include the statement on the same line as the WHILE statement, or use braces. ie. the following statement is not valid...

```
while (s != “quit”)
  read( s );
```

windows

List all windows showing their type and visibilty status and contents - ie which map and spectra buffers are displayed in them.

winfo

`winfo [window name]`

Creates an information window underneath the data window specified and displays in it the characteristics of the map or spectra.

zero

`zero [station id]`

Clear the spectra buffers for the specified stations.

zoom

`zoom [options]`

if map window selected - reissue map command using zoom box as new X and Y window limits where the zoom box was created using B1

if plot window selected - reset window limits to X0 and X1 markers

Options:

a	Automatically detects the non-zero region of a spectrum, and zooms into this region
lo=n	Set lower window limit to n
hi=n	Set upper window limit to n

[End of Manual]