

-.
NAME

functions – built-in functions

Operating System Utility Functions

`chdir()` – Changes `spec`'s current working directory to user's home directory as obtained from the environment variable `HOME`. Returns **true** or **false** as the command was successful or not. Updates the value of the built-in variable `CWD`.

`chdir(d)` – As above but changes to the directory *d*.

`unix()` – Spawns an interactive subshell using the program obtained from the user's environment variable `SHELL` or `shell`. Uses `/bin/sh` if the environment variable is unset. Returns exit status of shell.

`unix(cmd)` – As above, but uses `/bin/sh` to execute the one-line command *cmd*

`unix(cmd, var [, len])` – As above, but the second argument *var* is the name of a variable in which to place the string output of *cmd*. The maximum length of the string is 4096 bytes (including a null byte). The optional third argument *len* can be used to specify a larger size. This feature available since `spec` Release 4.03.01.

`time()` – Returns the current epoch in seconds. The UNIX epoch is the number of seconds from January 1, 1970, 00:00:00 GMT. The value returned includes a fractional part with the resolution dependent on the host platform. Millisecond resolution is standard, although on 80X86 systems only hundredth of a second resolution is returned.

`date()` – Returns a string containing the current date as
"Mon Feb 8 21:04:04 EST 1988"

`date(fmt)` – As above, but the output string is formatted according to the specifications in the string *fmt*. The format is passed to the standard C library `strftime()` function (see the `strftime` man page) with one addition: `spec` fills in the format options "%.1" through "%.9" with the fractional seconds, where the single digit specifies the number of decimal digits. For example, `p date("%m-%d-%Y %T.%.6")` would display
01-14-2005 22:59:30.148067.

`date(t [, fmt])` – As above, but from the epoch given by *t*. See `time()` above.

`file_info(f [, cmd])` – Returns information on the file or device named *f*. With just the one *f* argument, `file_info()` returns nonzero if the file or device exists and zero otherwise. If *f* is the string "?", the possible values for *cmd* are listed. If *f* is the string ".", `spec` uses the information from the last `stat()` system call, avoiding the overhead associated with an additional system call. The values for *cmd* and the information returned follow. Note that the first set of values essentially mimic the contents of the data structure returned by the `stat()` system call, while the second set of values have the same meaning as arguments to the `test` utility invoked from the shell.

"dev" – the device number on which *f* resides.

"ino" – the inode number of *f*.

"mode" – number coding the access modes and file attributes.

"nlink" – number of hard links for *f*.

"uid" – user id of the owner.

"gid" – group id of the owner.

"rdev" – device id if *f* is a block or character device.

"size" – size in bytes of *f*.

"atime" – time when *f*'s data was last accessed.

"mtime" – time when *f*'s data was last modified.

"ctime" – time when *f*'s attributes were last modified.

"isreg" or "-f" – whether *f* is a regular file.

"isdir" or "-d" – whether *f* is a directory.

"ischr" or "-c" – whether *f* is a character device.

"isblk" or "-b" – whether *f* is a block device.

"islnk" or "-h" or "-L" – whether *f* is a symbolic link.

"isfifo" or "-p" – whether *f* is a named pipe (fifo).

"issock" or "-S" – whether *f* is a socket.
"-e" – whether *f* exists.
"-s" – whether the size of *f* is greater than zero.
"-r" – whether *f* is readable.
"-w" – whether *f* is writable.
"-x" – whether *f* is executable.
"-o" – whether *f* is owned by you.
"-G" – whether *f* is owned by your group.
"-u" – whether *f* is setuid mode.
"-g" – whether *f* is setgid mode.
"-k" – whether *f* has its sticky bit set.

`file_info(pid, "alive")` – Returns nonzero if the process associated with the process ID *pid* exists and zero otherwise.

`getenv()` – Returns the string value of the environment variable *s*.

Evaluation Function

`eval(s)` – Parses and executes the string *s*. If the string is an expression, its value is returned. If the string is a statement or statement list, **true** (nonzero) is returned if there were no errors executing the statement(s) and **false** (zero) is returned if there was an error. Note, errors encountered during execution of the `eval()` string do not reset `spec` to the command level. Instead, the errors simply cause `eval()` to return an error, allowing execution of the calling statement block to continue.

Command Files

`dofile(f [, start])` – Queues the file *f* for reading commands. Returns nonzero if the file cannot be opened. If the optional argument *start* is an integer, the number specifies at which line to start reading the file. (Currently, only positive integers are allowed.) If the argument is anything else, it is considered a search string, and text is read from the file starting at the first line containing that search string. The metacharacters *, which matches any string, and ?, which matches any single character, are allowed in the search string. Initial and trailing white space is ignored when searching for a match.

`qdofile(f [, start])` – As above, but doesn't echo the contents of the file to the screen.

Help Functions

`gethelp(f)` – Prints the paginated help file *f* on the screen. If *f* contains a /, the argument is treated as an absolute or relative path name. Otherwise, the argument refers to a file in the *help* subdirectory of the `SPECD` directory. Returns non-zero if the file couldn't be opened.

`what_is(s, ["info"])` – With one argument, returns a number that indicates what the string argument *s* is. With two arguments, returns a string containing a text description of what *s* is. The number is a two-word (32-bit) integer, with the low word containing a code for the type of object and the high word containing more information for certain objects.

Low Word	High Word	Meaning
0	0	Not a command, macro or keyword
0	1	Command or keyword
2	Length	Macro name (length is in bytes)
4	0x0001	New-style data array
4	0x0010	Number-valued
4	0x0020	String-valued
4	0x0040	Constant-valued
4	0x0100	Associative array
4	0x0200	Built-in
4	0x0400	Global
4	0x0800	Unset
4	0x2000	Immutable
4	0x4000	Local
4	0x8000	Associative array element

Most type-4 symbols have more than one of the high-word bits set.

Controlling Output Files

`open()` – Lists all open files. Returns zero.

`open(file_name)` – Makes *file_name*, which is a string constant or expression, available for output. Files are opened to append. Returns zero for success, -1 if the file can not be opened or if there are too many open files.

`close(file_name)` – Closes *file_name* and removes it from the table of files available for output. Returns zero for success, -1 if the file wasn't open. Any open file should be closed before attempting to modify the file with other UNIX utilities. Otherwise the file may be corrupted if two processes are writing to the file.

`on()` – Lists all open files and indicates which ones are currently turned on for output.

`on(file_name)` – Turns on *file_name* for output. All messages, except for some error and debugging messages, but including all `print` and `printf()` output, are sent to all turned-on devices. If *file_name* has not been made available for output with the `open()` function, it will be opened. Returns zero for success, -1 if the file can't be opened or if there are too many open files.

`off(file_name)` – Turns off output to *file_name*, but keeps it in the list of files available for output. If this was the last turned-on file or device, `tty` is turned back on automatically. Returns zero for success, -1 if the file wasn't open.

Macro-Related Functions

`cdef("name", s, [key, [flags]])` – The function `cdef()` is used to define “chained” macros. The function can be used to maintain a macro definition in pieces that can be selectively included into a complete macro definition. The argument *name* is the name of the macro. The argument *s* contains a piece to add to the macro.

The chained macro can have three parts: a front, a middle and a back. Pieces included in each of the parts of the macros are sorted lexicographically by the keys. Pieces without a key are placed in the middle, in the order in which they were added, but after any middle pieces that include a key.

With the optional *key* argument, the pieces can be selectively replaced or deleted. The *flags* argument controls whether the pieces are added to the front or to the back of the macro or whether the pieces should be selectively included in the definition based on whether *key* is a currently configured motor or counter mnemonic. The bit meanings for *flags* are as follows:

- 0x01 – only include if key is a motor mnemonic and the motor is not disabled.
- 0x02 – only include if key is a counter mnemonic and the counter is not disabled.
- 0x10 – place in the front part of the macro.
- 0x20 – place in the back part of the macro.

If *flag* is the string "delete", the piece associated with *key* is deleted from the named macro, or if the name is the null string, from all the chained macros.

If *flag* is the string "enable", the parts of the named macro associated with *key* are enabled, and if *flag* is the string "disable", the associated parts are disabled. If *name* is the null string "", then all chained macros that have parts associated with *key* will have those parts enabled or disabled.

If *key* is the null string, the *flags* have no effect.

The `cdef()` function will remove any existing macro defined using `def` or `rdef`. Likewise, `def` and `rdef` will removed an existing `cdef()` macro with the same name. However, the commands `lsdef`, `prdef` and `undef` do work with chained macros. When `spec` starts, when the `reconfig` command is run (or the `config` macro is invoked) or when individual motors or counters are enabled or disabled, all the chained macros are adjusted for the currently configured and enabled motors and counters.

- `cdef("?")` – Lists all the pieces of all the chained macros.
- `cdef(name, "?")` – Lists the pieces of the macro named *name*, as will a "?" as the third or fourth argument.
- `clone(dest, src)` – Duplicates the macro *src* as a new macro named *dest*. Currently a clone of a `cdef` chained macro becomes an ordinary macro.
- `strdef(s)` – Returns a string containing the macro definition of *s*. If *s* is not a defined macro, returns the string *s* itself. (Available as of `spec` release 5.08.02-6.)

User Input and Output

- `input()` – Reads a line of input from the keyboard. Removes leading white space and trailing newline and returns the string. Returns the null string "" if only white space was entered.
- `input(s)` – As above, but prompts with the string *s*.
- `input(n)` – This function behaves differently depending on whether the input source is the keyboard or a pipe from another program (where `spec` is invoked with the `-p fd pid` option, with nonzero *fd*.)

In the usual case, if *n* is less than or equal to zero, the tty state is set to "cbreak" mode and input echo is turned off. Then `input()` checks to see if the user has typed a character and immediately returns a null string if nothing has been typed. Otherwise, it returns a string containing the single (or first) character the user typed. If *n* is less than zero, the cbreak, no-echo mode remains in effect when `input()` returns. If *n* is greater than zero, the normal tty state is restored (as it is also if there is an error, if the user types `^C` or if the user enters the `exit` command). Also, no characters are read and the null string is returned. The normal state is also restored before the next main prompt is issued, whether due to an error, a `^C`, or through the normal flow of the program.

On the other hand, when `spec` is invoked with the `-p fd pid` option, with nonzero *fd*, `input()` reads nothing but does return the number of characters available to be read. If *n* is nonzero, `input()` simply reads and returns a line of text, as if it had been

invoked with no argument.

`yesno([s,] x)` – Prompts the user with the optional string *s*, then waits for a yes or no response. The function returns 1 if the user answers with a string beginning with Y, y or 1. The value of *x* is returned if the user simply enters return. Otherwise the function returns 0. If the prompt string *s* is present, the characters " (YES)? " or " (NO)? " are appended depending on the value of *x*.

`getval([s,] x [, u])` – Prompts the user with the string *s*, if present, then waits for a user response. If the user enters a value, that value is returned. The value of *x* is returned if the user simply enters return. If the prompt string *s* is present, the string is printed followed by the current value of *x* and the string *u*, if present, in parenthesis, a question mark and a space. The function works with both number and string values. The optional third argument is intended to be used for a unit string (available as of spec release 5.08.02-6).

`getsval([s,] x)` – Like `getval()` above, prompts the user with the string *s*, if present, then waits for a user response. The value of *x* is returned if the user simply enters return. If the prompt string *s* is present, the string is printed followed by the current value of *x* in parenthesis, a question mark and a space. Unlike `getval()`, this function does not convert hexadecimal or octal input (number strings that begin with 0, 0x or 0X) to the corresponding decimal value. Rather, the `getsval()` function returns the literal string as entered.

`getline(f [, arg])` – This function reads the ASCII file given by the string *f* a line at a time and returns the string so obtained, including the trailing newline. If *arg* is the string "open", the function returns zero if the file can be opened for reading, otherwise -1 is returned. If *arg* is "close", the file is closed and zero is returned. If *arg* is zero, the first line of the file is returned. If only the first argument is present, the next line of the file is read and returned. At the end of the file, a -1 is returned. The previous file, if any, is closed and the new file is opened automatically when the file name argument changes (at least in this preliminary implementation).

`sscanf(s, fmt, arg [, ...])` – Scans the literal string or string variable *s* for data, where *fmt* contains a format specification in the same style as the C language `scanf()` function. Each *arg* is a variable name or array element that will be assigned the values scanned for. The function returns the number of items found in the string.

`printf(fmt [, a, ...])` – Does formatted printing on the turned-on output devices. See `printf()` in a C-manual. Returns **true**.

`fprintf(file_name, fmt [, a, ...])` – Does formatted printing on *file_name*. All other devices (except log files) are turned off while the string is printed.

`eprintf(fmt [, a, ...])` – Same behavior as `printf()`, above, except that if an error-log file is open, the generated output will also be written to that file, in addition to any other files or device turned on for output. The first line of any strings written to the error-log files will be prefixed with the #E characters.

`tty_cntl(s)` – Sends terminal-specific escape sequences to the display. The sequences are only written to the "tty" device and only if it is turned on for output. The sequences are obtained from the system terminal-capability data base using the value of the environmental variable TERM. The following values for *s* are recognized:

"ho" – move the cursor to the home position (upper left corner).

"cl" – clear the screen.

"ce" – clear to the end of the line.

"cd" – clear from current position to the end of the screen.

"so" – start text stand-out mode.
 "se" – end text stand-out mode.
 "md" – start bold (intensified) mode.
 "me" – end bold mode.
 "us" – start underline mode.
 "ue" – end underline mode.
 "mb" – start blink mode. (Note, xterms don't blink.)
 "mh" – start half-bright mode.
 "mr" – start reverse video mode.
 "up" – move up one row.
 "do" – move down one row.
 "le" – move left one column.
 "nd" – move right one column (non-destructive space).

`tty_cntl("resized?")` – Updates the `ROWS` and `COLS` variables in the event the window size has changed and returns a nonzero value if the window size has changed since the last call to `tty_cntl("resized?")`.

`tty_fmt(x, y, w, s)` – Writes the string `s` to the screen starting at column `x` and row `y`, where column 0, row 0 is the upper left corner of the screen. The string is only written to the "tty" device and only if it is turned on for output. If `s` is longer than the width given by `w`, the string is split at space characters such that no line is longer than `w`. Newlines in the string are retained, however. The function will truncate words that are wider than `w` and drop lines that would go off the bottom of the screen. Negative `x` or `y` position the cursor relative to the left or bottom edges of the screen, respectively. The function returns the number of lines written.

The two-letter control sequences listed above for the `tty_cntl()` function can be included in the string `s` by using the special string "`\[xx]`", where `xx` is the two-letter sequence. Note, though, the formatting code may fail if the sequence changes the current position of the output text.

`tty_move(x, y [, s])` – Moves the cursor to column `x` and row `y`, where column 0, row 0 is the upper left corner of the screen. If the third argument `s` is present, it is written as a label at the given position. The two-letter control sequences listed above for the `tty_cntl()` function can be included in the string `s` by using the special string "`\[xx]`", where `xx` is the two-letter sequence. The sequences and string are only written to the "tty" device and only if it is turned on for output.

Negative `x` or `y` position the cursor relative to the left or bottom edges of the screen, respectively.

Relative moves are specified by adding `+/-1000` to `x` or `y`. Both arguments must specify either relative or absolute moves. If one coordinate specifies a relative move, the absolute move in the other coordinate will be ignored. Note, not all terminal types support relative moves.

Counting and Moving

`sleep(t)` – Suspends execution for `t` seconds, where `t` may be non-integral. In old versions, a negative `t` indicated sleep was for clock ticks (1/60 second each). Returns **true**.

`mcnt(t)` – Begins counting for `t` monitor counts. Returns **true**.

`tcnt(t)` – Begins counting for `t` seconds. Returns **true**.

`cnt_mne(i)` – Returns the string mnemonic of counter `i` as given in the configuration file.

`cnt_name(i)` – Returns the string name of counter `i` as given in the configuration file.

- `cnt_num(mne)` – Returns the counter number corresponding to the counter mnemonic *mne*, or `-1` if there is no such counter configured.
- `counter_par(i, s [, v])` – Returns or sets configuration parameters for counter *i*. See the *counting* help file for more information.
- `mca_sel(n)` – Selects which MCA-type device to use with subsequent `mca_get()` and `mca_par()` commands. The numbering of MCA-type devices is determined by the order in which they appear in the config file. Returns `-1` if not configured for device *n*, otherwise returns zero. It is not necessary to use `mca_sel()` if only one MCA-type device is configured.
- `mca_get(g, e)` – Gets data from the currently selected MCA-type device, and transfers it to element *e* of data group *g*.
- `mca_put(g, e)` – Sends data to the currently selected MCA-type device from element *e* of data group *g*. (Not implemented for all devices.)
- `mca_par(s [, v])` – A device-dependent function to access various features and parameters of the currently selected MCA-type device. The string *s* selects an option. The argument *v* contains an optional numeric value. See the help file for the particular device for implemented options and return values.
- `image_par(s [, v])` – Returns or sets configuration parameters for 2D acquisition devices.
- `wait()` – Wait for all asynchronous activity to complete. Returns **true**.
- `wait(w)` – Wait for moving (*w*=1), counting (*w*=2) or other data acquisition (multi-channel scaling, for example) (*w*=4) to complete. If bit 5 of *w* is set, returns **true** if activities flagged by bits 1, 2 or 3 are active. Returns **false** otherwise.
- `stop(w)` – Stop moving (*w*=1), or counting or other asynchronous data acquisition (*w*=2). If *w* is zero or missing all asynchronous activity is halted. Returns **true**.
- `set_sim(i)` – Turns simulation mode on (*i*=1), off (*i*=0) or only reports state (*i*=-1). The first two return the value of the previous state as **true** (on) or **false** (off) and do a `wait()` before changing state. `set_sim(0)` reads in the motor settings file to restore motor positions.
- `motor_mne(i)` – Returns the string mnemonic of motor *i* as given in the configuration file.
- `motor_name(i)` – Returns the string name of motor *i* as given in the configuration file.
- `motor_num(mne)` – Returns the motor number corresponding to the motor mnemonic *mne*, or `-1` if there is no such motor configured.
- `motor_par(i, s [, v])` – Returns or sets configuration parameters for motor *i*. Values for the string *s* include "acceleration", "base_rate", "step_size", "velocity" or "backlash". The values may be modified by giving a value for *v*, although modifications to "step_size" must be enabled using
- ```
spec_par("modify_step_size", 1)
```
- first. See the *motors* help file and the help file for particular motor controllers for more information on possible parameters. Rereading the *config* file resets the values of the motor parameters to the values in the *config* file. Little consistency checking is done by `spec` on the values programmed with `motor_par()`. Be sure to use values meaningful to your particular motor controller.
- `get_lim(i, w)` – Returns the dial limit of motor *i*. If *w* > 0, returns high limit. If *w* < 0, returns low limit.

`set_lim(i, u, v)` – Sets the low and high dial limits of motor *i*. It doesn't matter which order the limits, *u* and *v*, are given. Returns -1 if not configured for motor *i* or if the motor is protected, unusable or moving, else returns 0.

`dial(i, u)` – Returns the motor dial position for motor *i* corresponding to user angle *u*.

`user(i, d)` – Returns the user angle for motor *i* corresponding to dial position *u*.

`chg_dial(i, u)` – Sets the dial position of motor *i* to *u* by changing the contents of the controller registers. Returns -1 if not configured for motor *i* or if the motor is protected, unusable or moving, else returns 0.

`chg_dial(i, s [, u])` – Starts motor *i* on a home or limit search, according to the value of *s*, as follows:

"home+" – move to home switch in positive direction.

"home-" – move to home switch in negative direction.

"home" – move to home switch in positive direction if current dial position is less than zero, otherwise move to home switch in negative direction.

"lim+" – move to limit switch in positive direction.

"lim-" – move to limit switch in negative direction.

Positive and negative direction are with respect to the dial position of the motor. (Not all motor controllers implement the home or limit search feature.) If present, the value of the third argument is used to set the motor's dial position when the home or limit position is reached (as of SPEC release 4.05.10-3). Returns -1 if not configured for motor *i* or if the motor is protected, unusable or moving, else returns 0.

`chg_offset(i, u)` – Sets offset (determining user angle) of motor *i* to *u*. Returns -1 if not configured for motor *i* or if the motor is unusable or moving, else returns 0.

### Plotting and Analysis

`data_grp(g, n, w)` – Configures data group *g*. The group will have *n* points, each having *w* elements. If *n* and *w* match the previous values for the group, the data in the group is unchanged. Otherwise, the data values of the reconfigured group are set to zero. If *w* is zero, the group is eliminated.

`data_info(g, s)` – Returns a number representing a parameter of the data group *g* according to the string *s* as follows:

"elem" – number of elements (width).

"npts" – number of points.

"last" – last modified (added) point.

Returns -1 if the group or command is invalid.

`data_put(g, n, e, v)` – Assigns the value *v* to element *e* of point *n* in group *g*.

`data_get(g, n, e)` – Returns the value of element *e* of point *n* in group *g*.

`data_nput(g, n, v0 [, v1 ...])` – Assigns values to point *n* of group *g*. Element 0 is assigned *v0*, element 1 is assigned *v1*, etc. Not all elements need be given, although elements are assigned successively, starting at element 0.

`data_uop(gs, es, gd, ed, uop [, arg])` – Performs the unary operation specified by the string *uop* on element *es* for all points in group *gs*. The results are put in element *ed* of the corresponding points in group *gd*. The source and destination groups and/or elements may be the same. If the number of points in the groups differ, the operation is carried out on up to the smallest number of points among the groups. See the *data* help file for possible values for *uop*.

- `data_bop(gs0, es0, gs1, es1, gd, ed, bop)` – Performs the binary operation specified by the string *bop* on elements *es0* and *es1* for all points in the groups *gs0* and *gs1*. The results are put in element *ed* for the corresponding points of group *gd*. The source and destination groups and/or elements may be the same. If the number of points in the groups differ, the operation is carried out on up to the smallest number of points among the groups. See the *data* help file for possible values for *bop*.
- `data_anal(g, s, n, e0, e1, op)` – Performs the operations indicated by *op* on *n* points in group *g*, starting at point *s*. The operations use the values in element *e0* (if applicable) and *e1*. If *n* is zero, the operations are performed on points from *s* to the last point added using `data_nput()` or `data_put()`. See the *data* help file for possible values for *op*.
- `data_dump(g, s, n, e0 [, e1 ...] [, fmt1] [, fmt2])` – Efficiently writes elements from group *g* to turned on output devices. The starting point is *s* and the number of points is *n*. The elements specified by *e0*, *e1*, etc., are printed. If *e0* is the string "all", all the elements for each point are printed. If *n* is zero, only the points from *s* to the last point added using `data_nput()` or `data_put()` are printed. The optional argument *fmt1* is a string, having the format "%#", that specifies how many data points (specified by the number #) to be printed on each line. The optional argument *fmt2* is a string that specifies an alternate *printf()*-style format for the values. Only e, g and f formats are recognized. For example, "%15.8f" uses fixed-point format with eight digits after the decimal point and a fifteen-character-wide field. The default output format is "%g" . See *printf()* in a C manual for more information. Note that in the default installation, the internal data arrays use single-precision floating values, which contain only about 8 decimal digits of significance.
- `data_read(file_name, g, s, n)` – reads data from the ASCII file *file\_name*, and stuffs the data into group *g* starting at point *s*, reading up to *n* points. If *n* is zero, all the points in the file are read. The values on each line of the file are assigned into successive elements for each point in the group. If there are more elements on a line in the file than fit in the group, or if there are more points in the file than in the group, the extra values are ignored. Returns -1 if the file can't be opened, otherwise returns the number of points read.
- `data_plot(g, s, n, e0, e1 [, e2 ...])` – Plots the current data in group *g* starting at point *s* and plotting *n* points. Element *e0* is used for *x*. Elements given by the subsequent arguments (up to a maximum of 64) are plotted along the *y* axis. If *n* is zero, only the points from *s* to the last point added using `data_nput()` or `data_put()` are plotted. If preceded by a call of `plot_cntl("addpoint")` and the ranges have not changed, only point *s+n-1* is drawn. If proceed by a call of `plot_cntl("addline")` the current plot will not be erased, and the plot ranges will not be changed. The plotting area is not automatically erased by a call of `data_plot()`—use `plot_cntl("erase")` for that. The axis ranges are set using the `plot_range()` function. See `plot_cntl()` for other options that affect drawing the plot.
- `data_plot(g, s, n, "all")` – As above, but uses element zero for *x* and the remaining elements (up to a maximum of 64) for *y* values. The number of elements is set with the `data_grp()` function.
- `data_fit(pars, g, s, n, edata, epars [, ...])` – Performs a linear fit of the data in element *edata* to the terms in the elements specified by *epars*. The fitted parameters are returned in the array *pars* supplied by the user. The function returns the *chi-squared* value of the fit, if the fit was successful. A -1 is returned if there are insufficient arguments or the covariance matrix is singular. The fit algorithm is along the same lines as the *lfit()* routine in *Numerical Recipes* (W.H. Press, et al., Cambridge

University Press, 1986, page 512).

`plot_cntl(s)` – Selects built-in plotting features. The argument *s* is a string of comma- or space-delimited options. See the `plot_cntl` help file for descriptions of the many options.

`plot_move(x, y [, s [, c]])` – Moves the current position to column *x* and row *y*, where column 0, row 0 is the upper left corner of the screen. If the third argument *s* is present, it is written as a label at the given position. If using color high-resolution graphics, the fourth argument, if present, is the color to use to draw the label. (See the `colors` help file.) The background color for the entire label will be the background color at the starting position. If graphics mode is not on, `plot_move()` works just as `tty_move()`.

`plot_range(xmin, xmax, ymin, ymax)` – Sets the ranges of the internally generated plots. If any of the arguments is the string "auto", the corresponding range limit is determined automatically from the data at the time the plot is drawn. If any of the arguments is the string "extend", the corresponding range limit is only changed if the current data decrease the minimum or increase the maximum. Returns **true**.

### CAMAC Hardware

`ca_get(i, a)` – Returns the 24-bit value read (using *F* = 0) from the *i*-th (*i* = 0, 1, ... ) CAMAC I/O device (from the `config` file) using subaddress *a*.

`ca_put(x, i, a)` – Writes the 24-bit value *x* (using *F* = 16) to the *i*-th (*i* = 0, 1, ... ) CAMAC I/O device (from the `config` file) using subaddress *a*. Returns the value written.

`ca_fna(f, n, a [, v])` – Sends the arbitrary FNA command to the module in slot *n*. If the dataway command given by *f* is a write function, the 24-bit value to be written is contained in *v*. If the dataway command given by *f* is a read command, the function returns the 24-bit value obtained from the module. The user should avoid issuing commands that would cause a LAM and should certainly avoid issuing commands to slots that are being used for motor or counter control by SPEC's internal hardware code.

`ca_cntl(cmd [, arg])` – Performs a CAMAC crate initialize if *cmd* is "Z" or "init", performs a crate clear if *cmd* is "C" or "clear", sets crate inhibit if *cmd* is "inhibit" and *arg* is 1, and clears crate inhibit if *cmd* is "inhibit" and *arg* is 0. During normal operation, you should not need to issue these commands. You should probably issue a `reconfig` after sending a crate initialize or clear.

### User Level Access To Hardware Interfaces

`gpib_cntl()`, `gpib_get()`, `gpib_put()`, `gpib_poll()` – See the `gpib` help file for detailed usage.

`ser_get()`, `ser_put()`, `ser_par()` – See the `serial` help file for detailed usage.

`sock_get()`, `sock_put()`, `sock_par()` – See the `sockets` help file for detailed usage.

`vme_get()`, `vme_get32()`, `vme_move()`, `vme_put()`, `vme_put32()` – See the `vme` help file for detailed usage.

### PC Port Hardware

`port_get(a)` – Reads one byte from the PC I/O port with the address *a*. Ports must be selected in the `config` file.

`port_getw(a)` – As above, but reads a 16-bit word.

`port_put(a, b)` – Write the byte *b* to the PC I/O port with the address *a*. Writable ports must be selected in the `config` file.

`port_putw(a, b)` – As above, but writes a 16-bit word.

#### Hooks To User-Added C-Code Functions

`calc(i)` – Calls user-added function having code *i*. Returns user supplied value.

`calc(i, x)` – As above, but passes argument *x* to the function.

#### String Handling

`asc(s)` – Returns the ASCII value of the first character of the string value of the argument *s*.

`length(s)` – Returns length of string *s*.

`index(s1, s2)` – Returns an integer indicating the position of the first occurrence of string *s2* in string *s1*, counted from 1, or zero if *s1* does not contain *s2*.

`split(s, a)` – splits the string *s* at space characters and assigns the resulting substrings to successive elements of the array *a*, starting with element 0. The space characters are eliminated. The functions returns the number of elements assigned.

`split(s, a, t)` – splits the string *s* into the elements that are delimited by the string *t* and assigns the resulting substrings to successive elements of the array *a*, starting with element 0. The delimiting characters are eliminated. Returns the number of elements assigned.

`substr(s, m)` – Returns the portion of string *s* that begins at *m*, counted from 1.

`substr(s, m, n)` – As above, but the returned string is no longer than *n*.

`sprintf(f [, a, ...])` – Returns a string containing the formatted print. See *printf()* in a C manual.

#### Useful Conversion Functions

`int(x)` – Returns integer part of *x*.

`bcd(x)` – Returns binary-coded decimal integer of positive *x*.

`dcB(x)` – Returns decimal equivalent of 32-bit BCD *x*.

`rad(x)` – Returns  $x \times \text{PI} / 180$ .

`deg(x)` – Returns  $x \times 180 / \text{PI}$ .

#### Standard Math Functions

`fabs(x)` – Returns absolute value of *x*.

`sqrt(x)` – Returns square root of *x*.

`cos(x)` – Returns cosine of *x*.

`sin(x)` – Returns sine of *x*.

`tan(x)` – Returns tangent of *x*.

`acos(x)` – Returns arc cosine of *x*.

`asin(x)` – Returns arc sine of *x*.

`atan(x)` – Returns arc tangent of *x*.

`atan2(y, x)` – Returns the arc tangent of *y/x* using the signs of the arguments to determine the quadrant of the return value. The return value is in the range -PI to PI.

`exp(x)` – Returns exponential of *x*.

`exp10(x)` – Returns power of 10 to the *x*.

`log(x)` – Returns natural logarithm of *x*.

`log10(x)` – Returns logarithm, base 10, of  $x$ .

`pow(x, y)` – Returns power of  $y$  to the  $x$ .

`rand()` – Returns a random integer between 0 and 32767.

`rand(r)` – If  $r$  is positive, returns a random integer between 0 and  $r$ , inclusive. If  $r$  is negative, returns a random integer between  $-r$  and  $r$ , inclusive. Values of  $r$  greater than 32767 or less than  $-16383$  are set to those limits. If  $r$  is zero, zero is returned. The C library `rand()` function is used to obtain the values. The seed is set to the time of day on the first call. The randomness (or lack thereof) of the numbers obtained is due to the C library implementation.

`srand(seed)` – Sets the seed value for the random generator used by the `rand()` function to the integer value `seed`. This feature allows the same sequence of random numbers to be generated reproducibly by resetting the seed to the same value.