spec™

X-Ray Diffraction Software

USER MANUAL and TUTORIALS

REFERENCE MANUAL

STANDARD MACROS GUIDE

FOUR-CIRCLE REFERENCE

ADMINISTRATOR'S GUIDE

HARDWARE REFERENCE

Portions derived from appendices to the Doctoral thesis of Alan Braslau,
Department of Physics, Harvard University, 1988.

This is version 3 of the spec documentation, printed July 16, 2017,
describing features of release 6 of the software.

spec

X-Ray Diffraction Software

Certified Scientific Software

PO Box 390640 □ Cambridge, Massachusetts 02139 □ (617) 576-1610
FAX: (617) 497-4242 □ spec@certif.com
http://www.certif.com

MANUAL SUMMARY

USER MANUAL and TUTORIALS

A beginner's guide to diffractometer operation and a tutorial on the basic features of the spec user interface.

REFERENCE MANUAL

An overview of the internal structure of spec and a complete description of all the built-in keywords, operators, grammar rules, commands and functions.

STANDARD MACROS GUIDE

Tips on writing macros, examples of macros from the standard library and a detailed description of the structure of the standard scan macros.

FOUR-CIRCLE REFERENCE

Describes the special functions, variables and macros used to operate the standard four-circle diffractometer. The various modes are described, along with the use of sectors, cut points and frozen angles. Also contains an explanation of how to fit the orientation matrix when the lattice parameters are unknown.

ADMINISTRATOR'S GUIDE

How to install and update the spec package. Explains the hardware configuration and motor settings file and how to use security features to protect motors from being moved by unqualified users.

HARDWARE REFERENCE

Information on the specific hardware devices and interfaces supported by spec.

TABLE OF CONTENTS

PREFACE
For the (In-Progress) Version 3 Manual

It's been eighteen years since the last significant spec manual update. In the preface for the 1999 edition, I apologized for the work-in-progress nature of the manual, and hoped to have a completely up-to-date version "someday in either this or the next century." We are now well into the next century, and although not yet completely up-to-date, the process of updating is underway.

Besides updated content, the PDF version of the manual now contains a PDF outline table of contents and active links within the document. Figuring out how to do that was a major step forward and motivated this current update project.

The updated manual intends to describe the current spec release 6 from September 2012 and beyond. References to when features were added to spec releases 3 (1993 to 1996), releases 4 (1996 to 2001) and releases 5 (2001 to 2012) will be eliminated.

Presently, effort is being made to bring the *Reference Manual* and *Administrator's Guide* up to date. The *Four-Circle Reference* has aged well and has needed minor updates only. However, the *User Manual and Tutorials* could use a major rewrite, and the *Standard Macros Guide* does not reflect the current form of many of the standard macros. The *Hardware Reference* describes hardware that may still be supported, but is rather obsolete, such as CAMAC devices and ISA PC boards. Its fate is yet to be decided.

Since the version of the manual that has been available for download for the last many years is so badly out of date, rather than wait for this rewrite to be complete, this work-in-progress version will be made available for download, with much more frequent updates. At any rate, this version of the manual should certainly give you the flavor of spec. For details on exactly how macros are constructed, you should probably consult the source code for the macros in your current distribution. The most recent on-line *changes* help files should be perused for the most up-to-date information on new features.

Also, please note that our website at http://www.certif.com contains downloadable PDF versions of this manual (both letter-size and A4!), and will surely be updated with more interim versions, before the next *official* and *complete* version is available.

Finally, if there are questions or ambiguities that cannot otherwise be resolved, by all means contact us at CSS for the final word.

Thanks for your patience.

G.S. May 10, 2017

USER MANUAL AND TUTORIALS

Introduction

spec is a UNIX-based software package for instrument control and data acquisition widely used for X-ray diffraction at synchrotrons around the world and in university, national and industrial laboratories. Developed in 1986 for X-ray diffraction experiments, spec's portability, flexibility and power are winning it increasing application as general-purpose data-acquisition software. spec is available on a wide range of UNIX platforms and supports numerous hardware configurations. Features include:

- Built-in code to regulate motor controllers and detection electronics using VME, CAMAC, GPIB, RS-232, PC-board and ethernet-socket interfaces.
- Generalized access for VME, CAMAC, GPIB, RS-232, PC I/O ports and socket I/O to read from and write to user devices.
- Sophisticated user interface with command interpreter, complete with variables, looping and flow control, allowing for creative experiment automation.
- Language uses a familiar C-like syntax.
- A command-file facility allows complicated or commonly used command sequences to be immediately called up.
- An easy-to-use macro facility, with a large library of predefined macros. Macros can be readily modified with any text editor to suit experiments.
- Scans, data-file formats, etc. are not built into the compiled program but defined as easily modified macros.
- High-resolution, real-time data plots are available on X Windows and Sunview systems.
- Macro libraries and geometry-calculation routines support two-, four-, five- and six-circle diffractometers, kappa diffractometers, many liquid surface X-ray diffractometers and other configurations. The standard four-circle diffractometer supports many advanced modes and includes features such as least-squares refinement of the lattice parameters. New geometry configurations can be easily created.
- Hardware configuration employs a spread-sheet-style interface to select device names, addresses, CAMAC slot assignments, motor parameters, etc.
- Security features let site administrators restrict access to particular motors (such as those at a synchrotron beam-line front end).
- Available for most UNIX Systems, widely used on Linux PC platforms and UNIX workstations, including SUN (both SunOS 4.x and Solaris 2.x), HP 700 series and IBM RS/6000.

Beginner's Guide to Diffractometer Operation

Starting Up

In this introduction to the basics of spec, as in most of this manual, the standard four-circle diffractometer configuration is used in the examples. Other specialized diffractometer geometries are available, but all configurations function in a similar manner.

To start up the four-circle version of the spec package from a UNIX shell, type:

```
% fourc
```

(In this manual, input from the keyboard will be indicated in **bold-faced** type.) You will see output similar to the following:

```
                Welcome to "spec" Release 6.05.02
        Copyright (c) 1987-2017 Certified Scientific Software
                     All rights reserved
                     [2017-01-29-101010]

(Portions derived from a program developed at Harvard University.)
(Linked with BSD libedit library for command line editing.)

Using "/usr/local/lib/spec.d" for auxiliary file directory (SPECD).

Getting configuration parameters from "SPECD/fourc/config".

BUsing four-circle configuration.

=
Type h changes for info on latest changes.
Browse to http://www.certif.com for complete documentation.
=

Reading file "SPECD/standard.mac".
Warning:  No open data file.  Using "/dev/null".
Type "mstartup" to initialize data file, etc.

Reading file "SPECD/four.mac".
Warning:  Using default lattice constants.
(UB recalculated from orientation reflections and lattice.)

1.FOURC>
```

The welcome message identifies the geometry configuration (`fourc`) and the release number of the program (`6.05.02`). The directory name that contains spec's auxil-

iary files is then identified (`/usr/local/lib/spec.d`). That name is assigned to spec's interval variable named `SPECD`.

A *configuration* file is then read to obtain the hardware configuration (hardware devices and types, stepper motor parameters, etc.). Various messages are printed as the specific hardware devices are initialized. During the start-up hardware configuration, spec reads the current diffractometer angle settings from the motor-controller hardware registers and verifies that they agree with the positions stored in a *settings* file associated with the diffractometer. If there is a discrepancy, you will see output similar to the following:

```
E500 at 0 steps (.1 user), spec at 24431 steps (12.3155 user)
on motor 1, slot 6, "Theta".  Modify the E500 registers?
```

The `E500` is one of many different motor controllers available. Since the controller shows 0 steps, it probably has been powered down, and the program value is probably correct. Type `yes` or `y` to modify the controller registers. If you are uncertain what to do, the safest thing is to immediately terminate the program without updating the motor *settings* files by typing the quit control character (usually a **^V** on IBM AIX platforms and a `^\` on most others), and then seek help.

A *news* file is displayed each time the program starts up. In this example, the `Spec Hot Line` message is from that file. The spec administrator can keep the news file up-to-date with messages for local users.

The first time you run spec, standard command files from the auxiliary file directory are automatically read (`SPECD/four.mac` and `SPECD/standard.mac`). These files contain the standard macro definitions used to operate the diffractometer. There are also some commands that assign default values to the variables used in the macros. The displayed *warning* message about no data file being open is produced by these standard macros, along with the message that suggests running the `startup` macro.

Finally, you are prompted for input. The prompt indicates the geometry configuration and includes a prepended command sequence number that is used with the command recall (or history) feature. You can exit the program by typing a **^D** (control-D) at the prompt:

```
1.FOURC> ^D
Bill's state is stored for /dev/console.

%
```

The closing message confirms that your spec state is saved. The spec state consists of all your current macro definitions, variables, open output files and command history. Each user has a unique state associated with a particular terminal. Your saved state is automatically restored the next time you run spec from the same terminal. (See page 59 in the *Reference Manual* to see how to start spec with a state from

another user or terminal.)

spec is built around an interpreter that has a C-like syntax and recognizes over a hundred built-in commands and function names. However, you will typically be invoking the standard macros. These macros are written to do specific jobs using the built-in commands and functions and require minimum keyboard input. If you just want to move motors, count photons and do scans, you will only have to learn the few standard macros presented in this *Beginner's Guide*.

Using the Printer and Data Files

The standard macros in spec are designed to keep records of the experiment in progress on a printer and in a data file, although neither is required. The `startup` macro will prompt for a printer and a data file, along with asking for many other parameters and options. For now, enter information just for the printer and data file and accept the current values for the other parameters.

```
1.FOURC> startup

Enter <return> for no change in the displayed parameters.
The names of start-up macros that can be invoked separately
are shown in parenthesis above a set of queries.
Type ^C to return to command level without finishing.
(Interrupting one of the specialized start-up macros will
likely undo any changes entered for its associated parameters.)

(newsample)
Title for scan headers (fourc)? cu 110

(newfile)
Data file (/dev/null)? cu110/94_01_31.a
Using "cu110/94_01_31.a".  Next scan is number 1.
Last scan # (0)? <return>

Use a printer for scan output (NO)? y
Printer device (/dev/null)? /dev/lp

(And so on ...)

2.FOURC>
```

When prompting for input, spec generally displays the default or current response in parentheses. Simply hitting <*return*> makes that selection.

You can use the `newfile` macro directly to open (or reopen) a data file. Usage is `newfile [`*`filename`* `[`*`scan_number`*`]]`. (As is the convention in this manual, the square brackets indicate optional arguments, and the *Courier Oblique* typeface denotes variable parameters you supply.) The optional argument *scan_number* is the

number of the last scan and should be specified when appending to an existing data file.

The standard spec macros allow you to use a printer to record scan data and other status information. Not all users use a printer, though, as the information is also stored in data files.

When using a printer, spec generates output for a 132-column wide format. Most spec users use 8½" wide paper with their printer set to compressed mode. The `initfx` macro sends the correct programming sequence to put an Epson printer into compressed mode. The `initdw` macro does the same for a Decwriter. (Other macros are available for other printers – type `lsdef init*` from spec for a list.) You could also use printer switches to select compressed mode.

Also, when using the printer, you should set the top-of-form position correctly. That way, each scan will begin at the top of a new page, and it will be much easier to locate scans when thumbing through the data printout later.

Use the `comment` macro (also available as `com`) to insert arbitrary comments in the data file and on the printer. For example,

```
2.FOURC> com Absorber inserted in front of detector

Mon Feb 15 01:41:52 1994.  Absorber inserted in front of detector.

3.FOURC>
```

Setting Motor Positions and Moving Motors

When spec is used to control an X-ray diffractometer, the `wh` (where) macro is available to show the positions of the most interesting angles and the diffractometer position in reciprocal space coordinates. With the four-circle diffractometer, the output is as follows:

```
3.FOURC> wh

H K L =  0  0  1
Alpha = 30  Beta = 30  Azimuth = 90
Omega = 0  Lambda = 1.54

Two Theta     Theta       Chi       Phi
  60.0000   30.0000  -90.0000    0.0000

4.FOURC>
```

The incident and scattered angles for surface diffraction (`ALPHA` and `BETA`), the `AZIMUTH` angle used in advanced modes (see the *Four-Circle Reference*) and the incident

X-ray wavelength, LAMBDA, are also listed.

The angular positions listed above are the *user* angles. You set user angles during diffractometer alignment to satisfy the premises of the geometry calculations, such as the positions of the zeroes of the angles. *Dial* angles keep track of hardware limits and prevent complete loss of angles from alignment errors or computer failure. The dial angles are generally made to agree with a physical indicator on each motor, such as a dial. User angles are related to the dial angles through the equation:

$$user = sign \times dial + offset$$

Redefining a user angle changes the internal value of *offset*. Dial angles are directly proportional to the values contained in the hardware controller registers. The *sign* of motion is set in the configuration file by the spec administrator and normally isn't changed.

The set_dial *motor position* macro is used to set the dial position of a motor. The argument *motor* is the motor number or mnemonic. All motors have short mnemonics, such as tth, th, chi, and phi.

```
4.FOURC> set_dial tth 24.526

Mon Feb 15 01:42:10 1994.  Two Theta dial reset from 0 to 24.526.

5.FOURC>
```

The set *motor position* macro is used to set the *user* position of a motor (i.e., to change *offset*). If you had a slit motor with mnemonic ts1, you might enter

```
5.FOURC> set ts1 .5

Mon Feb 15 01:43:31 1994.  Top Slit1 reset from 0 to .5.

6.FOURC>
```

The wa (where all) macro lists both the user and dial positions of all configured motors.

```
6.FOURC> wa

Current Positions  (user, dial)
Two Theta     Theta       Chi       Phi Top Slit1 Bot Slit1
      tth        th       chi       phi       ts1       bs1
  24.6310   12.3155   90.0000    0.0000    0.5000   -0.5000
  24.5260   12.2155   89.7865    0.0950    0.5000   -0.5000

7.FOURC>
```

spec also keeps track of software motor limits. These limits are always checked before any motors are moved. The `lm` macro lists these limits in both user and dial angles, as well as the current positions of the motors.

```
7.FOURC> lm

USER Limits (high, current, low):
Two Theta     Theta       Chi       Phi Top Slit1 Bot Slit1
 180.1050   90.1000  135.2135  179.9050    5.0000    0.0000
  24.6310   12.3155   90.0000    0.0000    0.5000   -0.5000
-179.8950  -89.9000 -134.7865 -180.0950    0.0000   -5.0000

DIAL Limits (high, current, low):
Two Theta     Theta       Chi       Phi Top Slit1 Bot Slit1
 180.0000   90.0000  135.0000  180.0000    5.0000    0.0000
  24.5260   12.2155   89.7865    0.0950    0.5000   -0.5000
-180.0000  -90.0000 -135.0000 -180.0000    0.0000   -5.0000

8.FOURC>
```

The macro `set_lm` *`motor low high`* changes the software limits for a single motor. The values for *`low`* and *`high`* are given in *user* angles (although they are stored internally in *dial* angles).

The `wm` *`motor`* [*`motor ...`*] macro lists complete information for up to six motors given as arguments.

```
8.FOURC> wm tth th

           Two Theta      Theta
                 tth         th
User
 High       180.1050    90.1000
 Current     24.6310    12.3155
 Low       -179.8950   -89.9000
Dial
 High       180.0000    90.0000
 Current     24.5260    12.2155
 Low       -180.0000   -90.0000

9.FOURC>
```

Once the diffractometer has been aligned, you can move to any allowed reciprocal space position using the `br` *`H K L`* (Bragg) macro.

```
9.FOURC> br 2 0 0

10.FOURC> wh

H = 2  K = 0  L = 0
ALPHA = -25.251  BETA = 25.251  AZIMUTH = 90  LAMBDA = 1.54

Two Theta     Theta       Chi       Phi
  50.5030   25.2515   90.0000    0.0000

11.FOURC>
```

You can see where the motors would move for particular values of (*H, K, L*) using the `ca` *`H K L`* (calculate) macro.

```
11.FOURC> ca 2 1 1

Calculated Positions:

H = 2  K = 1  L = 1
ALPHA = -25.252  BETA = 25.252  AZIMUTH = -90  LAMBDA = 1.54

Two Theta     Theta       Chi       Phi
  62.9960   31.4980   54.7355  135.0000

12.FOURC>
```

Conversely, spec will display the (*H, K, L*) that corresponds to a particular set of motor positions using the `ci` *`tth th chi phi`* (calculate inverse) macro.

A single motor may be moved in real space using the `mv` *`motor position`* macro. For example,

```
12.FOURC> mv tth 50

13.FOURC>
```

will move the 2θ motor to 50°. You are prompted for more input immediately, even though the motors are still moving.

You can tell when the motor has stopped moving by using the `w` macro. The program will pause until the motor has stopped moving and then generate a beep on the terminal. Alternatively, you can have the motor position displayed on the screen as it is moving by invoking the `umv` (updated-move) macro instead of `mv`. To stop the motors before they have finished moving, type the interrupt character, usually a **`^C`**.

You can use the `mvr` *`motor relative_position`* macro to move a motor relative to its current position.

```
13.FOURC> mvr th 1

14.FOURC>
```

will move θ by one degree.

The `tw` *`motor delta`* (tweak) macro is useful when lining up the diffractometer or when searching for the beam.

```
14.FOURC> tw th .1
Indicate direction with + (or p) or - (or n) or enter
new step size.  Type something else (or ^C) to quit.

th = 26.2515, which way (+)? <return>
th = 26.3515, which way (+)? <return>
th = 26.4515, which way (+)? <return>
th = 26.5515, which way (+)? <return>
th = 26.6515, which way (+)? -
th = 26.5515, which way (-)? <return>
th = 26.4515, which way (-)? ^C

15.FOURC>
```

Each time you hit <*return*>, the motor moves *`delta`* in the plus or minus direction.

Counting

You count photons using the `ct` macro. Without arguments, this macro counts for the time set by the variable `COUNT`, which is typically one second. According to the convention used in the standard macros, positive count times indicate counting to seconds and negative count times indicate counting to monitor counts.

```
15.FOURC> ct 10

Mon Feb 15 01:45:12 1994

    Seconds = 10
    Monitor = 389387 (38939/s)
   Detector = 192041 (19204/s)

16.FOURC> ct -40000

Mon Feb 15 01:45:28 1994

    Seconds = 1.027
    Monitor = 40000 (38948/s)
   Detector = 19756 (19237/s)

17.FOURC>
```

Type a **^C** to abort counting. The macro show_cnts will display the current scaler contents. The uct macro will update the screen with the current scaler contents during the counting period.

Scans

Scans in spec are built of macros. Many different standard scans are available. Absolute-position motor scans such as ascan, a2scan and a3scan move one, two or three motors at a time. Relative-position motor scans are lup (or dscan), d2scan and d3scan. The relative-position scans all return the motors to their starting positions after the last point. Two motors can be scanned over a grid of points using the mesh scan.

Simple reciprocal space scans are hscan, kscan and lscan. The hklscan macro moves the diffractometer along an arbitrary straight line in reciprocal space. Scans such as hkcircle or hkradial describe other trajectories. The hklmesh scan measures intensities over a grid of reciprocal-space points.

If you do not know the arguments for a scan or how a scan is used, you can call up its usage by typing its name with no arguments.

```
17.FOURC> ascan
Usage:  ascan motor start finish intervals time

18.FOURC> hscan
Usage:  hscan start finish intervals time

19.FOURC>
```

When the program does a scan such as hscan, the following happens: the program waits for motors to stop moving, calculates (*H, K, L*) for the current position and then scans *H*, holding *K* and *L* fixed for a reciprocal space scan along the *H* direction.

```
19.FOURC> hscan .9 1.1 20 1
Total 21 points, 21 seconds

Scan  20   Thu Feb 09 20:04:30 2017   file = cu110/90_01_31.a
hklscan 0.9 1.1 0 0 0 0 20 1
  #          H          K          L Detector  Monitor    Seconds
  0        0.9          0          0     2604    38939          1
  1       0.91          0          0     3822    38820          1
  2       0.92          0          0     5295    39034          1
  3       0.93          0          0     7259    38789          1
  4       0.94          0          0     9298    38804          1
  5       0.95          0          0    11505    38909          1
  6       0.96          0          0    13907    38821          1
  7       0.97          0          0    16022    39110          1
```

```
 8       0.98          0          0    17603    38839          1
 9       0.99          0          0    18834    38950          1
10          1          0          0    19103    38917          1
11       1.01          0          0    18701    39013          1
12       1.02          0          0    17652    39135          1
13       1.03          0          0    16011    38836          1
14       1.04          0          0    13848    38901          1
15       1.05          0          0    11585    38933          1
16       1.06          0          0     9302    39022          1
17       1.07          0          0     7237    39205          1
18       1.08          0          0     5324    38957          1
19       1.09          0          0     3780    38801          1
20        1.1          0          0     2580    38975          1

Peak at 1 is 19103   FWHM at 1 is 0.05   COM is 1
Sum = 231272  Ave.Mon./Time = 38921  Ave.Temp. = 0C
28 second

20.FOURC>
```

The output shown is what would generally appear on the screen. More detailed output is sent to the printer. Also, a complete scan header and the data points are stored in the data file. A rudimentary plot can be produced on the printer at the end of the scan. Typing `splot` will produce a plot of the data on the screen. Typing `pts` will list the data on the screen.

The `setplot` macro configures how the data will be displayed during and at the conclusion of scans.

```
20.FOURC> setplot

Do real-time screen plots during scans (NO)? y
Do screen plot after scan (YES)? <return>
Do printer plot after scan (NO)? y

21.FOURC>
```

Scans can be aborted by typing **^C**. Typing `scan_on` restarts an aborted scan at the current point.

Introduction To the spec User Interface

spec as a Calculator

In some respects, the spec user interface behaves like a BASIC language interpreter that uses the C language syntax. For example, you can easily print strings and the results of arithmetic expressions:

```
1.FOURC> p 2+2, sqrt(3), "2^16 =", 1<<16
4 1.73205 2^16 = 65536

2.FOURC>
```

(The p macro is defined as `print`, a built-in command.) You do not need to search for your calculator, as all the standard operators and functions are available.

The arithmetic operators (`=`, `*`, `/`, `%`, `+`, `-`, `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`), the relational operators (`>`, `<`, `<=`, `>=`, `==`, `!=`), the boolean operators (`!`, `&&`, `||`), the bitwise operators (`>>`, `<<`, `~`, `&`, `^`, `|`, `>>=`, `<<=`, `&=`, `^=`, `|=`) and the ternary operator (`? :`) are all available. Parentheses can be used for grouping within expressions. See the *Reference Manual* for a description of all the operators and their rules of precedence.

The most useful standard C math functions are included, such as `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `exp()`, `log()`, `log10()`, `pow()`, `sqrt()`, and `fabs()`. Conversions functions such as `deg()` and `rad()` convert between degrees and radians, while `bcd()` and `dcb()` convert between decimal and binary-coded decimal. A `rand()` function to return random numbers is also provided.

Numbers can be entered in decimal, octal or hexadecimal notation, just as in C.

```
2.FOURC> p 100, 0100, 0x100
100 64 256

3.FOURC>
```

Special string functions also exist. The `date()` function provides the current date and time as a string:

```
3.FOURC> p date()
Mon Feb 15 02:13:13 1994

4.FOURC>
```

The `date()` function can also take an argument that is the number of seconds from the UNIX epoch.

```
4.FOURC> p date(1e9)
Sat Sep  8 21:46:40 2001

5.FOURC> p date(0)
Wed Dec 31 19:00:00 1969

6.FOURC> p int(time()), date(time())
729760917 Mon Feb 15 02:21:57 1994

7.FOURC>
```

The second example shows the (Eastern Standard Time) moment of the UNIX epoch. The function `time()` returns the number of seconds since that moment, including a fractional part with a resolution determined by the system clock. The difference of subsequent calls to `time()` can, for example, give a reasonable elapsed time for each point in a scan.

The function `input()` reads a string from the keyboard. An optional argument will be printed first. For example, a macro or command file might prompt you for information:

```
7.FOURC> TITLE = input("Please enter a title:  ")
Please enter a title:  Au (001)  Sample #1

8.FOURC>
```

Other string functions such as `index()`, `substr()`, `length(`*s*`)` and `sprintf(`*format*`, [`*args*`])` are also available. See the *Reference Manual* for details.

Command Recall (History)

A command recall (or *history*) feature lets you recall previously typed commands. spec's command recall implements a subset of the features of the standard *csh* history mechanism. When using command recall, note that only keyboard input is saved, command recall cannot be used in command files, and the command recall characters must occur at the beginning of a new line.

When you run spec interactively, a command sequence number is always prepended to the prompt. The `history` command lists by number the commands that can be recalled. (At present, only the most recent 1000 commands are available for recall.)

```
8.FOURC> history
   1  p 2+2, sqrt(3), "2^16 =", 1<<16
   2  p 100, 0100, 0x100
   3  p date()
   4  p date(1e9)
   5  p date(0)
   6  p time(), date(time())
   7  TITLE = input("Please enter a title:  ")
   8  history

9.FOURC>
```

To use command recall, type `!!` or `!-1` to recall the previous command. Typing `!-2` will recall the second previous command. Type `!2` to recall command number 2. Also, `!TI` will recall the last command beginning with the string `TI`.

```
9.FOURC> !2
p 100, 0100, 0x100
100 64 256

10.FOURC>
```

Notice that the recalled command is first printed and then executed.

Recalled commands can be modified by appending text.

```
10.FOURC> !p , "= 100, 0100 and 0x100."
p 100, 0100, 0x100 "= 100, 0100 and 0x100."
100 64 256 = 100, 0100 and 0x100.

11.FOURC>
```

Arbitrary substitutions to recalled commands are allowed using the `:s/left/right/` modifier, as in

```
11.FOURC> !-1:s/./,respectively./
p 100, 0100, 0x100 "= 100, 0100 and 0x100, respectively."
100 64 256 = 100, 0100 and 0x100, respectively.

12.FOURC>
```

You can also use a circumflex `^` to make a substitution on the most recent command, just as with the standard UNIX *csh*.

```
12.FOURC> ^=^are^
p 100, 0100, 0x100 "are 100, 0100 and 0x100, respectively."
100 64 256 are 100, 0100 and 0x100, respectively.

13.FOURC>
```

Controlling Output To the Printer and Data Files

spec's output facility is unusual. Output files and devices, including the screen, are turned on or off for output. The output of each printing command, whether generated by a user command or internally, is sent to all the turned-on devices.

`open("filename")` opens a file or device to append output. The current contents of existing files are never erased. The `on("filename")` function turns on printing to the file or device and opens the file if `open()` was not previously called. The `off("filename")` function ends printing to that file or device, and `close("filename")` closes the file or device and removes the name from the program's table of file pointers. The name `"tty"` is special when used as an argument to these functions. It always refers to your current terminal.

Whenever there is an error or a **`^C`** interrupt, all files (except *log* files) are turned off, and output to the terminal is turned on. A log file is used for debugging purposes and is any file that begins with `log`. Output to all *on* files and devices is automatically copied to a log file.

To get the status of all open files, type:

```
13.FOURC> on()
‘tty’ has output ON.
‘/usr/alan/default.dat’ has output OFF.
‘/dev/null’ has output OFF.

14.FOURC>
```

If you change spec's current directory, you can reference open files either by the name with which the files were opened or by the correct path name relative to the new directory.

The standard macros use three output devices: the screen, a printer and a data file. The `ont`, `offt`, `onp`, `offp`, `ond` and `offd` macros are usually used to simplify controlling output to these devices, where `ont` is defined as `on("tty")`, etc. Typical usage is

```
ond; offt; printf("#S %d\n", ++SCAN_N); offd; ont
```

Often, printing commands are placed between `onp` and `offp` to direct the output both to the screen and the printer. For instance,

```
1.FOURC> onp; p "This is also being printed on the printer."; offp
This is also being printed on the printer.

2.FOURC>
```

Formatted printing is available using the `printf()` and `fprintf()` functions. The

format specifications are the same as for the C-language routine and can be found in the *printf()* write-up in any C reference manual.

```
2.FOURC> printf("The square root of two is %.12g.\n", sqrt(2))
The square root of two is 1.41421356237.

3.FOURC>
```

Using Variables

spec's variables can be used as both strings and as double-precision floating-point numbers. Variables are not declared, but come into existence through usage. Some variables are built-in, though, and of these, some have preassigned values. The variable `PI` is an example.

```
3.FOURC> {
4.more> k = 2 * PI / 1.54
5.more> print k
6.more> }
4.07999

7.FOURC>
```

Curly brackets (`{` and `}`) are used to delimit a block to be interpreted together, since variables are local to interpreted blocks. Notice that the prompt indicates the program is expecting further input before interpreting and taking action.

```
7.FOURC> print k
0

8.FOURC>
```

The value of k disappeared because k was local to the previous statement block. New variables start off with a value of zero.

A variable may be declared `global` to hold its value outside an interpreted block:

```
8.FOURC> global Lambda CuKa

9.FOURC> Lambda = 1.54

10.FOURC> CuKa = "Copper K-alpha"

11.FOURC> print CuKa, "=", Lambda
Copper K-alpha = 1.54

12.FOURC>
```

By convention, global variables in the standard macro package use capital letters or begin with an underscore. Variables can be made `constant` to protect them from accidental reassignment,

```
12.FOURC> constant Lambda 1.54

13.FOURC> Lambda = 1.7
Trying to assign to a constant ‘Lambda’.

14.FOURC>
```

Variables defined as constant are automatically global.

Most built-in variables with preassigned values are of the *immutable* type and cannot be changed at all:

```
14.FOURC> PI = 1
Trying to assign to an immutable ‘PI’.

15.FOURC>
```

Some built-in variables, such as `DEBUG`, can be changed by the user. Another variable, the `A[]` array, may be filled by the program with the current motor positions or can be set to target motor positions. For a list of all current symbols, type:

```
15.FOURC> syms
       ( Built-In/Global/Local Array Number String Constant/Immutable )
 8256 A              (BA...)    80 SLIT_W         (G.NS.)    80 _f1            (G.NS.)
   96 ADMIN          (G..S.)    96 SPEC           (B..SI)    80 _f2            (G.NS.)
   80 BG             (G.N..)   112 SPECD          (B..SI)    80 _f3            (G.NS.)
   96 COLS           (B.N..)    80 TEMP_CS        (G.NS.)    80 _fx            (G.NS.)
   80 COUNT          (G.N..)    80 TEMP_SP        (G.NS.)    80 _g1            (G.NS.)
   96 COUNTERS       (B.N.I)    96 TERAMP_MIN     (G.N..)    80 _g2            (G.NS.)
   96 COUNT_TIME     (G.NS.)    96 TERM           (B..S.)    96 _hkl_col       (G.N.C)
  128 CP_FILTER      (G..S.)    80 TIME           (G.NS.)    80 _m             (G.NS.)
  112 CWD            (B..SI)    96 TIME_END       (G.NS.)    80 _m1            (G.NS.)
  128 DATAFILE       (G..S.)    96 TITLE          (G..S.)    80 _m2            (G.NS.)
  112 DATA_DIR       (G..S.)    80 T_AV           (G.NS.)    80 _m3            (G.NS.)
   80 DATE           (G.NS.)    80 T_HI_SP        (G.N..)    80 _n1            (G.NS.)
   80 DEBUG          (B.N..)    80 T_L            (G.NS.)    80 _n2            (G.NS.)
   80 DEGC           (G.NS.)    80 T_LO_SP        (G.N..)    80 _nm            (G.NS.)
   80 DEGC_SP        (G.NS.)  2736 U              (BA...)    80 _numgeo        (G.N..)
   80 DET            (G.N..)  1056 UB             (BA...)    80 _pmot          (G.NS.)
   80 DOFILE         (G.NS.)   592 UNITS          (GA...)    96 _pmotflag      (G.NS.)
   96 DO_DIR         (G..S.)    80 UPDATE         (G.N..)    96 _pre_chk       (G.N..)
   80 EPOCH          (G.N.C)    96 USER           (B..SI)    80 _pwid          (G.NS.)
   80 FPRNT          (G.NS.)    96 USER_CHK_ACQ   (G.NS.)    80 _reg_f         (G.NS.)
 1296 G              (BA...)    96 USER_CHK_COUNT (G.NS.)    80 _reg_i         (G.NS.)
   96 GS_file        (G..S.)    96 USER_CHK_MOVE  (G.NS.)    80 _reg_n         (G.NS.)
   80 GS_scan        (G.N..)    80 VFMT           (G.NS.)    80 _reg_s         (G.NS.)
   80 GS_xcol        (G.N..)    80 VPRNT          (G.NS.)   112 _reg_scan      (G..S.)
   80 GS_ycol        (G.N..)    80 X_L            (G.NS.)    80 _reg_t         (G.NS.)
```

```
   96 GTERM          (B..S.)    80 Y_L           (G.NS.)    80 _s            (G.NS.)
   80 HEADING        (G.NS.)   816 Z             (BA...)    80 _s1           (G.NS.)
  112 HOME           (B..SI)    80 _1            (G.N..)    80 _s2           (G.NS.)
   96 MAIL           (G..S.)    80 _2            (G.NS.)    80 _s3           (G.NS.)
   80 MODES          (G.N.C)    80 _3            (G.NS.)    80 _sleep        (G.NS.)
   80 MON            (G.N..)    80 _4            (G.NS.)    80 _stime        (G.NS.)
   96 MON_RATE       (G.NS.)    80 _5            (G.NS.)    80 _stype        (G.NS.)
   80 MOTORS         (B.N.I)    80 _6            (G.NS.)    80 _sx           (G.NS.)
   80 MT_AV          (G.NS.)    80 _7            (G.NS.)    96 _upd_flg      (G.NS.)
   80 NPTS           (G.NS.)    80 _8            (G.NS.)    80 bg_m          (G.NS.)
   80 PFMT           (G.NS.)    80 _9            (G.NS.)    80 bg_pts        (G.N..)
   80 PI             (B.N.I)    80 _LAMBDA       (G.NS.)    80 bg_yI         (G.NS.)
   96 PLOT_MODE      (G.N..)    96 _bad_lim      (G.NS.)    80 chi           (B.N.I)
   80 PL_G           (G.N..)    80 _c1           (G.NS.)    96 chk_thresh    (G.NS.)
   80 PL_G1          (G.N..)    80 _c2           (G.NS.)    80 det           (B.N.I)
   80 PL_X           (G.N..)    80 _c3           (G.NS.)  1184 gmodes        (GA...)
   80 PL_Y           (G.N..)    80 _c4           (G.NS.)   656 mA            (GA...)
   80 PPRNT          (G.NS.)    80 _cols         (G.NS.)    80 mon           (B.N.I)
  112 PRINTER        (G..S.)    80 _const        (G.NS.)    80 phi           (B.N.I)
 1856 Q              (BA...)    80 _cp           (G.NS.)   128 rplot_col     (G..S.)
   80 REFLEX         (G.NS.)    80 _ctime        (G.NS.)    80 sec           (B.N.I)
   80 ROWS           (B.N..)    80 _d            (G.NS.)   128 splot_col     (G..S.)
10576 S              (BA...)    80 _d1           (G.NS.)    80 th            (B.N.I)
   80 SCAN_N         (G.NS.)    80 _d2           (G.NS.)    80 tth           (B.N.I)
   80 SLIT_H         (G.NS.)    80 _d3           (G.NS.)
   80 SLIT_N         (G.N.C)    80 _f            (G.NS.)
Memory usage is 41088 bytes.

16.FOURC>
```

The number preceding each name is the number of bytes of memory the variable consumes. All the global variables in the list above come from the standard start-up macro files. Those variables that begin with an underscore are internal to the standard macro package.

Flow Control

Flow control allows you to construct complex scans and other macros to control experiments and take data. The syntax of the flow control is very similar to standard C. For example, to list all the motor positions, you can use the following loop:

```
16.FOURC> for (i = 0; i < MOTORS; i++) {
17.more>    printf("Motor %d  %-10s = %g\n", i, motor_name(i), A[i])
18.more> }
Motor 0  Two Theta = 3
Motor 1      Theta = 1.5
Motor 2        Chi = 0
Motor 3        Phi = 0

19.FOURC>
```

As in C, the `for` statement contains within parentheses three optional expressions separated by semicolons. The first expression is executed before entering the loop. The second is the test done before each pass of the loop — if it evaluates false, the loop is terminated. The third expression is executed at the end of each loop.

The conditional statements

```
if ( condition ) statement
```

and

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

are also available. For example, to test whether a variable has been assigned a value, you could examine the return value of the built-in `whatis()` function (described on page 82 in the *Reference Manual*).

```
19.FOURC> if (whatis("DATAFILE")>>16&0x8000)
20.more> print "Warning, Data file is uninitialized!"
21.more> ;
Warning, Data file is uninitialized!

22.FOURC>
```

When there is a solitary `if` statement, a semicolon, extra newline or some other command must be read before the `if` statement will be executed, as it is not clear to the parser whether an `else` statement will follow. If there is an `else` statement, it must follow the `if` portion of the statement on the next line.

```
22.FOURC> if (whatis("DATAFILE")>>16&0x8000)
23.more> print "Warning, Data file is uninitialized!"
24.more> else
25.more> print "Data is being stored in", DATAFILE
Data is being stored in /usr/alan/default.dat

26.FOURC>
```

The `while` construction is also available. Usage is

```
26.FOURC> while (wait(0x22)) {
27.more>     getangles
28.more>     printf("%10.4f\r", A[tth])
29.more> }

30.FOURC>
```

As in C, `continue` and `break` statements may be used within loops. The statement `exit` can be used anywhere and always causes a jump back to the command level.

Macro Facility

One of spec's most powerful features is its provision for defining macros. Through macros, you can simplify use of the diffractometer as well as determine the style and format of the data output. Through the macro facility, you can customize the environment to include any enhancements or specialized requirements for your experiment. Standard macro sets included in the spec package support conventional two-circle, four-circle and z-axis diffractometers along with some specialized liquid surface diffractometers. These macros control the measurement and the recording of experimental data and establish a standard format for ASCII data files.

An example of a simple macro that can be used to record a comment on the printer is

```
30.FOURC> def com '
31.quot>       on(PRINTER)
32.quot>       printf("$*\n")
33.quot>       off(PRINTER)
34.quot> '

35.FOURC>
```

Notice the prompt shows the program is expecting the quote to be closed. The variable `PRINTER` contains a string naming the printer device used to document the diffractometer operation. To use the above macro, type:

```
35.FOURC> com This is a comment.
This is a comment.

36.FOURC>
```

The text `This is a comment` is substituted in the `printf()` function for the symbol `$*` and is printed on both the screen and the printer.

Each argument following a macro call is available to the macro using `$1, $2, ...`, where `$1` refers to the first argument, and so on. Up to 25 arguments may be used. An argument is a string of characters separated by *white space* (spaces and tabs) or enclosed in single or double quotes, `$*` represents all the arguments, and `$#` is the number of arguments.

When a macro definition contains argument substitution, and you invoke that macro with more arguments than needed, the extra arguments you typed up to the next `;`, `}` or newline disappear. However, if the macro does not use argument substitution in its definition, text typed following the macro invocation is not thrown away.

To see what a macro contains, use the command `prdef` to print out the macro definition.

```
36.FOURC> prdef com
def com '
      on(PRINTER)
      printf("$*\n")
      off(PRINTER)
'
37.FOURC>
```

Notice that the form of the definition, if written to a file, would be suitable for reading back in as a macro definition.

The standard macro library is read automatically the first time you run spec or when you start the program with the `-f` flag. You can get a listing of all the currently defined macros with the command `lsdef`.

```
37.FOURC> lsdef
ALPHA         (4)  bug         (275)  hklscan    (1639)  qdo          (10)
AZIMUTH       (4)  ca          (182)  hkradial    (334)  rplot        (10)
BETA          (4)  calcA         (7)  hlcircle    (339)  rplot_res   (112)
CEN          (10)  calcE         (7)  hlradial    (333)  save        (432)
Escan      (1448)  calcG         (7)  hscan       (139)  savegeo    (1121)
F_ALPHA       (4)  calcHKL       (7)  initdw       (42)  saveslits    (99)
F_AZIMUTH     (5)  calcL         (8)  initfx       (39)  saveusr       (0)
F_BETA        (4)  calcM         (7)  initnec      (44)  savmac      (118)
F_OMEGA       (5)  calcZ         (7)  initoki      (39)  scan_head     (5)
F_PHI         (5)  cat          (14)  inittemp     (33)  scan_loop     (5)
Fheader       (0)  cd           (11)  klcircle    (339)  scan_move     (5)
Flabel        (2)  ci          (177)  klradial    (333)  scan_on     (192)
Fout          (2)  cl           (22)  kscan       (139)  scan_plot     (0)
Ftail         (0)  com          (12)  l            (16)  scan_tail     (5)
H             (4)  comment     (184)  less         (15)  set         (344)
K             (4)  config      (109)  lm          (539)  set_E       (314)
L             (4)  count         (6)  lp_plot     (674)  set_dial    (649)
LAMBDA        (4)  ct           (47)  ls           (13)  set_lm      (349)
OMEGA         (4)  cuts        (764)  lscan       (139)  setaz       (448)
Pheader       (0)  cz          (177)  lup         (419)  setlat      (764)
Plabel        (2)  d            (12)  mail         (16)  setmode     (927)
Pout          (2)  d2scan      (564)  measuretemp   (1)  setmono     (368)
RtoT_0      (162)  d3scan      (688)  mesh       (1221)  setplot    (1119)
RtoT_1      (162)  debug       (212)  mk          (175)  setpowder   (867)
RtoT_2      (161)  do            (9)  move_E      (208)  setscans     (67)
RtoT_3      (163)  dscan        (95)  move_em       (8)  setsector  (1341)
TtoR_0      (160)  dtscan      (143)  mv          (175)  setslit     (464)
```

```
TtoR_1      (160)  dumbplot    (334)  mvd         (192)  setslits    (469)
TtoR_2      (159)  end_reflex  (131)  mvr         (191)  settemp     (217)
TtoR_3      (161)  freeze      (456)  mz          (268)  show_cnts   (327)
_check0     (240)  g_aa          (4)  ned          (14)  showslits   (114)
_chk_lim    (266)  g_al          (4)  newfile    (1165)  showtemp    (133)
_cleanup2     (0)  g_bb          (4)  newmac      (270)  splot         (8)
_cleanup3     (0)  g_be          (4)  offd         (13)  splot_res   (150)
_count      (158)  g_cc          (4)  offp         (12)  startgeo     (45)
_do         (479)  g_chi0        (5)  offsim      (126)  starttemp   (225)
_getcut       (8)  g_chi1        (5)  offt         (10)  startup     (245)
_head      (1363)  g_frz         (4)  ond          (12)  te          (105)
_hkl_lim     (84)  g_ga          (4)  onp          (11)  teramp      (458)
_hklline   (1254)  g_h0          (5)  onsim       (127)  th2th       (141)
_hklmesh    (638)  g_h1          (5)  ont           (9)  tscan       (732)
_loop       (413)  g_haz         (5)  or0         (549)  tw          (742)
_mo_loop    (175)  g_k0          (5)  or1         (551)  u            (10)
_mot        (124)  g_k1          (5)  or_swap     (320)  uan          (23)
_move        (37)  g_kaz         (5)  p             (8)  ubr          (31)
_pcount     (132)  g_l0          (5)  pa         (1301)  uct         (364)
_plot_scale (392)  g_l1          (5)  pl          (176)  umk          (31)
_pmove      (131)  g_laz         (5)  pl_CFWHM     (10)  umv          (19)
_scanabort  (102)  g_mo_d        (5)  pl_COM       (10)  umvr         (20)
_setcut       (8)  g_mo_s        (5)  pl_FWHM      (10)  unfreeze     (39)
_settemp      (1)  g_mode        (4)  pl_LHMX      (10)  upl          (24)
_tail        (83)  g_om0         (4)  pl_MAX       (10)  uwm         (694)
_update1    (204)  g_om1         (5)  pl_MAXX      (10)  vi           (13)
_update2    (255)  g_phi0        (5)  pl_MIN       (10)  vt52_rplot (1190)
_update4    (370)  g_phi1        (5)  pl_MINX      (10)  vt52plot    (914)
_var        (171)  g_sect        (4)  pl_SUM       (10)  w            (12)
a2scan     (1172)  get_E        (74)  pl_SUMSQ     (11)  wa          (231)
a3scan     (1439)  getvar       (65)  pl_UHMX      (10)  waitall       (7)
add_reflex  (408)  gpset       (123)  pl_xMAX      (11)  waitcount     (7)
an          (173)  gt101_rplot(1190)  pl_xMIN      (11)  waitmove      (7)
ansi_rplot (1196)  gt101plot   (915)  plot         (40)  wh           (48)
ansiplot    (933)  h             (4)  plot_res    (435)  whats       (744)
ascan       (865)  help         (24)  prcmd        (42)  wm         (1232)
beep         (12)  hi            (7)  pts          (87)  yesno       (162)
beg_reflex  (283)  hkcircle    (339)  pwd          (11)
br          (175)  hklmesh    (1095)  qcomment    (176)

38.FOURC>
```

Each macro is listed as well as the number of characters in its definition. Some macros have zero length — their definitions are assigned during the course of an experiment. In the standard library, macros that are only used within other macros (and not meant to be referenced directly by the user) begin with an underscore.

Another macro handling command allows you to remove a macro definition.

```
38.FOURC> undef com

39.FOURC> prdef com
com: undefined.

40.FOURC>
```

There are several special macro names. If a macro named `cleanup` is defined, it will be automatically invoked whenever there is an error or **^C** interrupt. This macro can be defined to print a message, update a file, return motors to a starting position, etc. For example, in the standard macro library, something like the following is defined for the duration of a scan:

```
def cleanup '
      comment "Scan aborted after %g points." NPTS
      undef cleanup
'
```

Similarly, a macro named `cleanup1` can be defined, which behaves the same way. However, if `cleanup` exists, it will be run first.

Also `begin_mac`, `end_mac` and `prompt_mac` have special meaning. (text forthcoming ...)

Command Files

Macros are generally defined and maintained using the command file facility. In addition, sequences of experimental scans are often called up using command files. Command files are ASCII files of text, created with any of the UNIX text editors, and contain input just as it would be typed interactively. Command files are read line by line by spec when invoked with the functions `dofile()` or `qdofile()`. For example,

```
40.FOURC> dofile("spec.mac")
Opened command file 'spec.mac' at level 1.

FOURC.1> (Commands from file echoed as read ...)
```

The `.1` extension to the prompt indicates the level of nesting. Command files can be nested to five levels.

The function `qdofile()` is identical to the function `dofile()` except that the commands are not echoed as they are read.

```
40.FOURC> qdofile("spec.mac")
Opened command file 'spec.mac' at level 1.

41.FOURC>
```

If a file named *spec.mac* exists in your current directory, it is read as a command file each time you run spec. You can have private initialization code and macros in this file.

Two standard macros are defined to simplify reading command files. The macros `do` and `qdo` are normally used in place of the above functions. In addition to supplying the parentheses and quotation marks around the file name and recording the *do* command on the printer and in the data file, these macros also allow you to repeat the last command file when a dot is given as the argument:

```
41.FOURC> qdo .

qdo spec.mac
Opened command file 'spec.mac' at level 1.

42.FOURC>
```

Just as with keyboard input, comments can be included in a command file. Everything following a # up to a newline is ignored by spec.

Status and Help

Below is a summary of the diagnostic commands — most have been previously mentioned.

`syms`	lists built-in, global and local symbols.
`lsdef`	lists the names and sizes of macros.
`prdef`	prints out macro definitions.
`lscmd`	lists built-in keywords and functions.

All the above commands can take *pattern* arguments, employing the metacharacters `?` and `*`. In a pattern argument, `?` stands for any single character, while `*` stands for any string of characters, including the null string. For example, `lsdef ???` lists all the three-letter macros:

```
42.FOURC> lsdef ???

CEN  (10)  com  (12)  ned  (14)  or0 (549)  qdo  (10)  uct (364)
_do (479)  lup (419)  ond  (12)  or1 (551)  set (344)  umk  (31)
bug (275)  mvd (192)  onp  (11)  pts  (87)  uan  (23)  umv  (19)
cat  (14)  mvr (191)  ont   (9)  pwd  (11)  ubr  (31)  upl  (24)

43.FOURC>
```

Likewise, `lsdef *scan` shows all the macro names that end in `scan`.

```
43.FOURC> lsdef *scan

Escan  (1448)  ascan   (865)  dscan    (95)  hscan   (139)  tscan   (732)
a2scan (1172)  d2scan  (564)  dtscan  (143)  kscan   (139)
a3scan (1439)  d3scan  (688)  hklscan(1639)  lscan   (139)

44.FOURC>
```

A single * matches everything.

An on-line *help* facility exists to display files from spec's *help* directory:

```
44.FOURC> help
     Help is available on the following subjects (type "h subject"):

386         config     files      geometry   macros     powder     sizes
ackno       counting   flow       gpib       news       print      spec
angles      debug      fourc      help       pdp        serial     syms
changes     do         functions  history    plot       simulate   syntax

45.FOURC>
```

help (and simply h) are macros that use the built-in gethelp() function to print files contained in the help directory. The above listing is also produced by typing gethelp ("help") . The command gethelp("news") is automatically executed each time the program starts up. New help files particular to a site may be added to the help directory.

UNIX Commands

The easiest way to write macro definitions is to use a standard text editor to create a command file, and the easiest way to get at the text editor is through the unix() function that spawns subshells.

```
45.FOURC> unix("vi macro.defs")
"macro.defs" 3 lines, 20 characters

46.FOURC> qdo macro.defs
Opened input file 'macro.defs' at level 1.

47.FOURC>
```

Any UNIX command may be spawned as in the above example. Because this is so useful, a macro has been written to simplify the syntax. You could type:

```
47.FOURC> u vi macro.defs
"macro.defs" 3 lines, 20 characters

48.FOURC> qdo macro.defs
Opened input file ‘macro.defs’ at level 1.

49.FOURC>
```

The `unix()` command (or the `u` macro) with no argument will spawn a subshell. You return to spec upon exiting the subshell. spec uses the shell environment variable *SHELL* or *shell*, if set, to select the type of UNIX shell. By default, */bin/sh* is used. With arguments, `unix()` uses */bin/sh* to execute the one-line command. For some common UNIX commands, macros such as the following are defined in the standard library.

```
def cat ’unix("cat $*")’
def ls  ’unix("ls $*")’
def l   ’unix("ls -l $*")’
def vi  ’unix("vi $*")’
```

The working directory of spec can be changed as with the shell.

```
49.FOURC> cd data
Now in ‘data’

50.FOURC>
```

The macro `cd` used above is defined using the built-in function `chdir()`. Only the working directory of the program spec is changed; the shell from which you started spec is not touched.

Moving Motors

A primary purpose of spec is to manipulate an X-ray diffractometer according to a calculated geometry. The automation of the angular settings is accomplished through the use of motor controllers interfaced to the computer. spec can be configured to control any number of motors.

As explained earlier, motor positions are referred to as dial positions and user positions. The diffractometer is operated in user positions. Dial positions are used to provide a stable point of reference. The two differ possibly by a sign and/or an offset. Dial positions should be set to always agree with the physical dials of the diffractometer motors. The user positions are then set in the line-up procedure of the

diffractometer. For example, they may be set to zero at the direct beam. The relations between the two positions are:

$$dial = hardware\_register / steps\_per\_unit$$
$$user = sign \times dial + offset$$

The *hardware_register* contains the value maintained by the stepper motor controller. The value of *steps_per_unit* is assigned in the hardware configuration file, as is *sign*. The latter must be chosen to agree with the conventions of the built-in geometry calculations.

The motor positions are often placed in the `A[]` array. The array is built-in and its elements can be used like any other variables. What makes it special, however, are the commands that use the array to convey the positions of the diffractometer motors. For example, the command `getangles` sets all of the elements of the `A[]` array to the current motor positions in user angles, while the `move_all` command sends the motors to the positions contained in `A[]` (in user angles). Typical usage is,

```
50.FOURC> waitmove       # Make sure no motors are active.

51.FOURC> getangles      # load A[] with user angles.

52.FOURC> A[0] = 3       # move motor #0 to 3.

53.FOURC> move_all       # start the move.

54.FOURC>
```

(The # symbols introduce comments.) It is important to first wait for any previous motions to complete. Then `getangles` is used to load the angle array with the current positions. Only the values for the motors to be moved are reassigned before using `move_all` to set the motors in motion.

A macro that would list the user positions of all the configured motors might be:

```
54.FOURC> def wa '
55.quot>    getangles
56.quot>    for (i = 0; i < MOTORS; i++)
57.quot>          printf("%9.9s = %g\n", motor_name(i), A[i])
58.quot> '

59.FOURC> wa
 Two Theta = 3
     Theta = 1.5
       Chi = 0
       Phi = 0

60.FOURC>
```

The motor_name() function returns the motor name assigned in the configuration file.

The motor positions are stored in three locations: in program memory, on the computer's hard disk and in the hardware registers associated with the motor controller. The program manipulates the angles in its memory. The values on the hard disk are updated every time a motor is moved but are only read when the program is initialized, or after the reconfig command is invoked. The controller registers count the actual number of steps moved and should be the true positions of the motors (unless a motor was switched off). Before each motor is moved, the controller registers are compared with program memory. If there are discrepancies, you are notified and asked which value is correct. The sync command can also be used to synchronize the controller registers with program memory. The angles can get out-of-sync by moving the motors with manual controls, by turning off the power to motor controllers or perhaps by a computer crash.

Although the motor controllers work in steps, it is much more convenient to use real units such as degrees (or, for linear motion, millimeters). The user and dial angles are in these units, converted from steps by the step-size parameters that are read from the configuration file.

The chg_dial(*motor, dial_angle*) function sets the dial register to *dial_angle* for one motor. The chg_offset(*motor, user_angle*) function sets the offset used to convert from dial positions to user positions for one motor. Often during the line-up procedure you will want to zero a particular angle:

```
60.FOURC> chg_offset(th, 0)  # set motor theta to zero

61.FOURC>
```

The set macro includes the above and documents the change on the printer and in the data file.

```
61.FOURC> set th 0

Wed Aug 19 11:53:33 1987.  Theta reset from 1.5 to 0

62.FOURC>
```

Dial and user settings may also be set by the spec administrator using the program *edconf*. See page 229 in the *Administrator's Guide* for further details.

Usually, diffractometer motions have a limit of travel, beyond which physical damage may occur (or a hardware limit switch tripped). Software limits therefore exist to prevent you from accidentally moving a motor out of range. The lower and upper limits for each motor are contained in internal arrays accessed through the

`set_lim(motor_number, low, high)` and `get_lim(motor_number, flag)` functions. With the latter, the lower limit is returned if `flag` is less than zero, otherwise the upper limit is returned.

If a `move_all` command would take any motor outside of its limits, an error message is printed and no motors are moved. The limit values are stored in dial angles since they correspond to physical limitations to the motion. The limit values are therefore preserved as the user-angle offsets are changed. The `set_lm` macro can be used to set a single motor's limits:

```
62.FOURC> set_lm tth 0 360

Two theta limits set to 0 360 (dial units).

63.FOURC>
```

The angle arguments to the macro `set_lm` are given in user angles. (In this example, dial and user angles are the same.)

Diffractometer Geometry

You can operate a two-circle diffractometer in terms of angles alone. However, for a four-circle diffractometer (and others such as the z-axis or liquid-surface diffractometers) it makes more sense to work in three-dimensional reciprocal space coordinates. It is therefore necessary to be able to calculate angles according to the diffractometer geometry.

spec is designed to accommodate a variety of diffractometer configurations. The particular calculations are contained in geometry code (the source for which is included in the standard spec package.) accessible through the `calc()` function. The arguments to `calc()` determine the particular code that is called. For example, a `calcA` macro is defined as `calc(1)`. Its purpose is to load the `A[]` array with the angles corresponding to the current reciprocal space coordinates. The four-circle configuration represents the three reciprocal space coordinates as the first elements of the built-in array `Q[]`. For convenience, the following definitions are made:

```
def H 'Q[0]'
def K 'Q[1]'
def L 'Q[2]'
```

Thus, to move to a position in reciprocal space such as the point [100], the appropriate commands would be

```
63.FOURC> H = 1; K = L = 0; waitmove; getangles; calcA; move_all

64.FOURC>
```

Whenever the move_all command is used, it is important that the A[] array contain the current motor positions for all motors except the ones to be moved. In the above example, the getangles command loads A[] with the current positions after the waitmove ensures all motors have stopped. The calcA changes the appropriate elements of the A[] array and the move_all starts the motors.

Often, you might change a single angle, or several angles, and then wonder where in reciprocal space the diffractometer is set. The calcHKL macro will take the positions in the A[] array and set the variables H, K, and L to the calculated coordinates. For example:

```
64.FOURC> waitmove; getangles; calcHKL; print H, K, L
1 0 0

65.FOURC>
```

The command getangles loads the A[] array with the current positions.

Counting

Another important function of the diffractometer program is to measure the scattered X-ray intensities. spec supports several types of timers, scalers and multichannel analyzers (MCAs). Timers control the count time. Scalers count detected photons. MCAs accumulate many channels of counts and are used with energy-dispersive detectors and positional-sensitive detectors.

To count the number of X rays incident on the detector per second, the counting hardware must be able to accumulate detector counts accurately within a fixed time period. The scaler hardware is gated by a clock that operates independently of the computer. Thus, the response time of the computer to interrupts (real-time events) does not affect the accuracy of the count. spec programs and starts the clock and senses when the clock period, and hence the counting, has ended. spec can then read the contents of the scalers and save the measurement in a data file.

Clearing the scalers and starting the clock is accomplished by the function tcount(seconds). To count for one second, type:

```
65.FOURC> tcount(1)

66.FOURC>
```

The contents of the scalers are accessed through the built-in S[] array. The hardware scalers are read and their contents loaded into the scaler array by the getcounts command. A second, associated string array, S_NA[], is defined in the standard macros and identifies each scaler:

```
66.FOURC> getcounts; printf("%s = %g\n%s = %g\n%s = %g\n",\
67.cont> S_NA[0], S[0]/1000, S_NA[1], S[1], S_NA[2], S[2])
seconds = 1
monitor = 347
detector = 35031

68.FOURC>
```

The first scaler, labeled `seconds`, is usually fed a 1 kHz signal, so it actually tracks milliseconds and is therefore divided by 1000. The number of scalers available depends on the particular hardware and the number of detectors and monitors used. The default scaler channel numbering for the first three scalers puts a 1 kHz time signal in scaler 0, monitor counts in scaler 1 and detector counts in scaler 2.

You can also count to a fixed number of monitor pulses, rather than to a fixed time period.

```
68.FOURC> mcount(1e4)

69.FOURC> getcounts; printf("%s = %.1f\n%s = %g\n%s = %g\n",\
70.cont> S_NA[0], S[0]/1000, S_NA[1], S[1], S_NA[2], S[2])
seconds = 28.8
monitor = 10000
detector = 1.00954e+6

71.FOURC>
```

Counting is asynchronous, i.e., the `tcount()` and `mcount()` functions return immediately after starting the clock. They do not wait until the counting period is over. Use the `wait()` function to determine when counting is finished.

A useful macro has been written to count and print the scaler contents:

```
71.FOURC> ct 5

Thu Aug 20 19:11:51 1987
Seconds = 5  Detector = 175103 (35020.6/s)  Monitor = 1730 (346/s)

72.FOURC>
```

If the argument is omitted, a default count time (stored in the global variable `COUNT`) is used. A positive argument to `ct` signifies seconds; a negative argument signifies monitor counts.

```
72.FOURC> ct -10000

Thu Aug 20 19:13:42 1987
Seconds = 28.3  Detector = 1.0434e6 (36869.3/s)  Monitor = 10000 (353.36/s)

73.FOURC>
```

CAMAC, GPIB and Serial

Besides the built-in hardware support for moving motors and counting using CAMAC (IEEE-583), GPIB (IEEE-488) and serial (RS-232C) interfaces, spec provides generalized input/output support over these interfaces from the command level.

The ca_get(*device*, *subaddress*) command will return the contents of the addressed module register (*F=0* in the standard *FNA* CAMAC command code), while ca_put(*data*, *device*, *subaddress*) will write the 24-bit value *data* to the addressed module register (*F=16*). The *device* argument is the I/O module index from the configuration file and can be 0, 1 or 2. The CAMAC slot number of the module is set in the configuration file. The *subaddress* argument is the module's subaddress (the *A* in the *FNA*).

spec allows you to send a string of characters to a GPIB instrument at any GPIB address and to read a string of characters from any instrument. When reading characters, spec will accept a string terminated by either a newline or by carriage return—newline, or you can specify the number of bytes to be read. For example, to initialize a particular voltmeter having GPIB address 12, you would issue the command:

```
73.FOURC> gpib_put(12, "D0R0Z0B0T0K1M0G1X")

74.FOURC>
```

That instrument might then be read with:

```
74.FOURC> {k_ohms = gpib_get(12); print k_ohms}
100.024

75.FOURC>
```

The command

```
75.FOURC> x = gpib_get(12, 4)

76.FOURC>
```

would read 4 bytes from device 12 and not look for terminators.

When sending strings using gpib_put(), you cannot send null bytes. Usually a device that requires null lower order bits in a data byte will ignore the high order (parity) bit. In this case, you can usually set that highest bit to avoid sending a null byte.

The ser_get(*device*, *n*) and ser_put(*device*, *string*) functions access the serial interface, where *device* is the index from the configuration file and can be 0, 1 or 2. In ser_get(), *n* is the most number of bytes to read. The function will return after reading one line (terminated by a newline or carriage return) from the device, even if the number of bytes is less than *n*. In ser_put(), *string* contains the characters to be written.

Using spec with C-PLOT and Other UNIX Utilities

Standard Data File Format

The format of the data files used by spec is set at the macro level. The files are ASCII, so they can be easily manipulated by other UNIX utilities such as *grep*, *sed*, *awk* or any of the editors. The format of files produced by the standard macros is described here.

When opened with the `newfile` macro, the following header is written to initialize the data file:

```
#F /tmp/data
#E 729994936
#D Wed Feb 17 19:22:16 1994
#C cu 110  User = bill
#O0 Two Theta  Theta  Chi  Phi
```

Information or control lines begin with a # character, with the character in the second column indicating the type of information that follows. The first line of the data file header contains the name by which the file was opened. The next line is the number of seconds from the UNIX epoch as returned by the `time()` function. In the data that will follow, each scan point will include a field containing the number of seconds elapsed since that file creation time. Next in the header is a line containing the date as returned by the `date()` function, then a comment line. That is followed by a line containing all the motor names in use. Each motor name is separated from the other by two spaces. What will then follow will be various comment lines created by the `comment` macros, user defined entries and scan data.

Each scan has a header that looks like the following, always beginning with a blank line:

```
#S 1  hklscan  0.9 1.1  0 0  0 0  20 1
#D Wed Feb 17 19:25:55 1994
#T 1  (Seconds)
#G0 0 0 0 0 0 1 0 0 0 0 0 0
#G1 1 1 1 90 90 90 3 3 3 90 90 90 1 0 0 0 1 0 60 30 0 0 0 0 60 30 0 -90 0 0 0
#Q .9 0 0
#P0 29.745 29.745 90 0
#N 7
#L H  K  L  Epoch  Seconds  Monitor  Detector
```

The first line of the scan header contains the scan number followed by the basic scan name and its arguments. (Scans invoked as `hscan` or `kscan` execute the basic `hklscan`, scans such as `lup` or `dscan` execute `ascan`, etc.) The next line is the date and time the scan was started. Following that, the `#T` control line indicates that the scan was counting to time and for what duration at each point. A `#M` would indicate

counting to monitor.

The next lines give information that describe the diffractometer configuration at the start of the scan. Following #G0 are the current values of the four-circle parameters, which are defined four-circle macro file, *macros/fourc.src*. Following #G1 are the current parameters describing the crystal lattice and orientation matrix. The identification of these parameters is in the macro file *macros/ub.mac*. The #Q line gives the *H, K, L* coordinates at the start of the scan, while the #P0 line gives the motor positions at the start of the scan with each column corresponding to the motor names in the #O0 line of the scan header. The 7 after #N indicates there will be seven columns of data in the scan, and the #L line gives the names for each column, each name separated from the other by two spaces. The detector counts are always placed in the last column, preceded by the monitor counts, if counting to time, or the elapsed time for that data point, if counting to monitor. The Epoch column has the number of seconds elapsed from the time of the #E at the start of the file.

Following the scan header is the scan data. These are just lines of space-separated numbers that correspond to the column headers given with #L. Intervening #C comment lines may lie within the rows of data if, for example, the scan was aborted and then restarted with the scan_on macro. Otherwise, the data continues until the next non-comment control line or blank line.

You can develop your own programs and scripts to extract data from the spec data file, or you may want to use the *scans.4* user function that is part of the C-PLOT package, or the stand-alone *scans* program that is based on *scans.4*.

Scans.4

The C-PLOT user function *scans.4* reads in files of ASCII data according to a modest set of conventions. In particular, *scans.4* manipulates the X-ray scattering data from the spec data files, doing scan averaging, background subtraction, data normalization and error bar calculation. The C-language source code to *scans.4* is always available on your system, is liberally commented and should always be consulted if there is any question as to what operations are being done on your raw data points.

The *scans.4* user function can be invoked from C-PLOT either as,

```
PLOT-> fn  scans.4
```

or

```
PLOT-> fn  scans.4 options scan-numbers
```

Use the second form when running from command files. The possible options are:

`.`	Use same options as last time.
`-i`	Initialize, used to start up function and return.
`-f` *`filename`*	Select scan file name.
`-p`	Print scan file contents.
`+e` or `-e`	Calculate (or don't) error bars from statistics.
`+s` or `-s`	Sort and merge (or don't) data by x values.
`+d` or `-d`	Collect (or don't) 3 columns of data.
`+r` or `-r`	Rerange (or don't) plot axis for each new data set.
`+S` or `-S`	Retrieve scans by scan (or file) number.
`+v` or `-v`	Print (or don't) each line of scan file (verbose).
`+n` or `-n`	Normalize (or don't) data points.
`-m`	Turn on `+n` flag and normalize to monitor counts.
`-t`	Turn on `+n` flag and normalize to time.
`x=#`	Set column for *x* values.
`y=#`	Set column for *y* values.
`z=#`	Set column for *z* values and turn on `+d` flag.

The default options are:

```
-f data  +esSn  -rvd  -m  x=1  y=-1
```

Retrieving Scans By Scan Number or File Position Number

Scans can be retrieved by entering either the scan number (option `+S`, default) or the file position number (option `-S`). Scan numbers are determined by the `#S` lines in the file. The file position number is the sequence position of the scan in the file, irrespective of scan number. Normally, the scan number and the file position number are the same.

When selecting by scan numbers, if there is more than one scan with the same number, the last one is retrieved. Specify which instance of a repeated scan to retrieve by using the *scan.sub* syntax. For example, selecting scan 10.3 retrieves the third instance of scan number 10.

Negative numbers count back from the end of the file and are always considered to be file position numbers. For example,

```
fn . -1
```

will always return the last scan in the file. You can read in multiple scans by giving several scan numbers as arguments. You can read in a group of consecutive scans with

```
fn . 3-7 10-14
```

The above reads in scans 3 through 7 and 10 through 14.

Merging Scans and Background Subtraction

The default `+s` option causes the data points from all the scans read in to be sorted by *x* values and data points with the same *x* value averaged. If data is to be normalized and/or error bars calculated, the appropriate weight is given to the count time for each point.

Scan numbers that end with the letter `b` are used as background scans. The sort-and-merge flag should be in effect when using background scans. Entering

```
fn . 12b 13 14b 15b 16 17b
```

for example, or

```
fn . 13-15 16-27b
```

will merge the data from the background scans with the data scans, subtracting the background counts from the data counts at each *x* value. When doing merging and background subtraction, the *x* values must be identical for the data points to be merged.

File Conventions

scans.4 only uses some of the control lines in the standard spec data files described earlier.

The control conventions used by *scans.4* are:

`#S number`	Starts a new scan. *number* is the user's numbering scheme.
`#M number`	Indicates data was taken counting to *number* monitor counts.
`#T number`	Indicates data was taken counting for *number* seconds.
`#N number`	Indicates there are *number* columns of data.

The following control lines do nothing, although they will be printed to the screen while reading a scan.

`#C comment ...`	Conventionally a comment.
`#D date`	Conventionally the date.
`#L lab1 lab2 ...`	Conventionally data column labels, with each label separated by two spaces.

Data Columns

By default, *x* values are taken from the first column, *y* values from the last column. Monitor counts are always taken from the column prior to the *y* column.

When entering column numbers, a negative number counts backwards from the last column. If the column for *x* is zero, the value put in for *x* is just the row number of the point with respect to the start of the data for the current scan.

More Details

After *scans.4* reads and indexes a data file, it remembers the file length. If you answer affirmative to the `Change modes?` query, *scans.4* will add to the index if the file has lengthened.

If you give a dot (`.`) as the command line argument or in response to `Scans/options` query, the previous argument or option string will be used. That is, the string is remembered, not the options chosen interactively using `Change modes?` For instance, if you enter a long sequence of scan numbers and read in the scans, then change something via `Change modes?`, you can simply enter a dot in response to `Scans/options` and recover the previous sequence of scan numbers.

When you do enter a string of flags and scan numbers, the modes set by the flags only apply to the scans that follow the flags and not the preceding scans.

The Index File

Indexing a long ASCII data file to find at what byte offset each scan begins takes time. Once *scans.4* has indexed a file, it saves the index information in a binary-format *index* file. The name of the index file is formed by appending *.I* to the original data-file name. As long as the index file is more recent than the data file, *scans.4* will take the index information from the index file.

Normalization and Error Bars

The values returned as error bars are those due to counting statistics (the square root of the number of counts). When the counts are derived from the algebraic combination of detector, background and monitor counts, the error bars are calculated using the appropriate *propagation of errors* formalism.[1]

Contents

A *contents* program is included in the spec package. The program tries to print summary scan information from spec's standard data files. Usage is

```
contents [ options ] file1 [ file2 ... ]
```

Current options are

`-o` *output*	Name of output file, otherwise standard output is used.
`-s` *start*	Starting number for first scan of first file.
`-p` *page*	Lines per page.
`-d`	Send control codes appropriate for DecWriter II.
`-c`	Print `#C` comment lines.

[1] P. R. Bevington, *Data Reduction and Error Analysis for the Physical Sciences*, (McGraw-Hill, New York, 1969), p. 64.

Showscans

The *showscans* program is a shell script that serves as a front end for an *awk* script called *show.awk*, which is installed in the spec auxiliary file directory. The awk script prepares a C-PLOT command file that will be automatically run to make summary plots of scans from spec data files. Twelve scan plots are placed on each page. The text is very small, so a high resolution display device is recommended.

Usage is

```
showscans [options] [file_options] file [[file_options] file ...]
```

where the global *options* relate to C-PLOT commands and are:

`-eb error_bars`	Error bar mode (default is 0, off).
`-ft font`	Font code (default is font 2).
`-sy symbol`	Symbol code (default is `L`, a line).
`-zi filter`	Graphics filter (default is *psfilter*).
`-w`	Wait for keyboard <return> after each page
`-x`	Shorthand for `-zi "x11 -rotate" -w`

The *file_options* apply only to the next file on the command line and are:

`-f from_scan_number`	Starting scan number for the following file.
`-t to_scan_number`	Ending scan number for the following file.

In no scan numbers are specified, all the scans within a file will be processed.

File names that end with *.I* are ignored as they are assumed to be *scans.4* index files, allowing you to use metacharacters to specify filenames a little more freely.

The *awk* script recognizes all the standard scan headings produced by the standard spec scan macros and uses that information to select the *x*-axis label and the column for the independent variable in the data file. The title of each of the small plots is the starting Q vector of the displayed scan, taken from the `#Q` line of the data files.

If the monitor counts are zero for any point in a scan, that scan is not plotted. The *show.awk* file can be edited to disable that feature.

REFERENCE MANUAL

Introduction

The material contained in this part of the documentation describes most of the built-in features of the spec program, that is, those parts of spec that are compiled into the code and cannot be changed at the installed site. These features include the user interface and the general hardware support, but do not include application-dependent features, such as geometry code for operating diffractometers and macro libraries.

Internal Structure Of spec

This section briefly explains spec's internal structure to give an overview of how it is constructed.

First, consider how a user's input gets interpreted. The initial translation of characters typed at the keyboard (or read from a command file) is done by the *input preprocessor*, which keeps track of the input sources and handles command recall (or history) substitution.

The input text is then broken into *tokens* by the *lexical analyzer*. Tokens represent the different kinds of input objects, such as *keywords*, *operators*, *variables*, *number constants* and *string constants*. When the lexical analyzer recognizes a predefined *macro* name, its definition, possibly with argument substitution, is pushed back onto the input stream to become further input to the lexical analyzer.

The *parser* in spec repeatedly calls the *lexical analyzer* to obtain tokens. The *parser* contains a set of *grammar rules* that determine acceptable sequences of tokens. A *syntax error* occurs when input violates these rules. When enough input is read to satisfy a complete set of rules, the parser returns a parse tree (or mini-program) to the *executor*. The executor code then steps through the parse tree, calling the appropriate internal functions for each action.

The macro-definition command `def` is an exception to the above rules. As soon as the parser recognizes the `def` command sequence, the macro name and its definition are saved and made available to the lexical analyzer, even while the parser is still building the mini-program. A different command, `rdef`, defers storing the macro definition until the mini-program is executed. The `rdef` command is useful when some flow control logic needs to be run to decide what definition to assign to the macro.

Understanding the difference between the parse phase and execution phase of spec is important. Each time the command-level prompt is given, a new parse tree will be created. If several semicolon-separated commands are given on the same line, a

separate parse tree will be created for each. However, curly brackets can be used to group any number of lines together to form just one parse tree. A significant consequence of the parse tree mechanism is limitation of the scope of a nonglobal variable to the statement block in which it is referenced.

spec may detect error conditions during each of the phases described above. Some of these errors (and the interrupt character, usually a `^C`) reset spec to the command-level prompt.

spec also manages auxiliary files. The *state* file contains the variables, macro definitions, output file names and additional parameter values unique to each user, terminal and diffractometer. The state file preserves the current situation when the user leaves the program so that the same situation can be restored when the user later returns to the program. The *history* file stores the user's command history. spec also creates an empty *lock* file to prevent a second instance of spec from running using the same state file. A *points* file stores the configuration and data associated with the deprecated data group facility. All these files are placed in the *userfiles* subdirectory of each configuration's auxiliary file area.

Syntax Description

Comments

A # introduces a comment. Everything following a # on an input line is ignored (unless the # is within a string). Comments are retained in macro definitions and are counted in the macro length.

Doc Strings

Everything between pairs of triple double quotes """ is a doc string comment. The comment block can span lines, but not files. Unlike comments that begin with a pound sign #, doc string comments will not be saved with a macro when included as part of the macro definition in the source file. Thus, doc string comments can be interspersed in macro code without putting any extra burden on the input preprocessor as it does macro substitution.

Identifiers

An identifier is a name — it can be a variable name, a macro name or an array name. An identifier may begin with the letters `a-z`, `A-Z` or _ (underscore). The remaining

letters in a name may be those characters or the digits `0-9`. There is no limit to the number of characters in a variable name. In the syntax rules described later, such names are represented by the term *identifier*.

Arrays

Arrays are formed using square brackets, as in *identifier* `[` *expression* `]` or *identifier* `[` *expression* `][` *expression* `]`. Both one- and two-dimensional arrays are supported. spec has two kinds of arrays with very different properties: *associative* arrays and *data* arrays.

Associative Arrays

With *associative* arrays, the array index *expression* can be any numeric or string-valued constant or expression. Associative array element values can be numbers or strings. For two-dimensional associative arrays, the two array indices for each element are stored as a single string formed using the string value of the first index, followed by the character `\034`, followed by the string value of the second index. One can access such a 2D array element using a single index constructed according to the above recipe. That is, `arr["list"]["one"]` refers to the same item as `arr["list\034one"]`.

Associative arrays can be initialized by assignment, as in

```
1.FOURC> x = [ "one":"now", "two":"is", "three":"the", "three":0:"time" ]

2.FOURC> print x
x["one"] = "now"
x["three"] = "the"
x["three"]["0"] = "time"
x["two"] = "is"

3.FOURC>
```

Here `x` is either an uninitialized variable or an existing associative array. If `x` already existed as something other than an associative array, the above assignment statement would produce an error. Note that one- and two-dimensional initializers can be mixed. Note also that the `print` command sorts the elements by index. The index specifiers are optional for 1D arrays in the initialization. If not used, the position in the list is used as the index, starting at zero.

```
3.FOURC> x = [ "now", "is", "the" ]

4.FOURC> print x
x["0"] = "now"
x["1"] = "is"
x["2"] = "the"

5.FOURC>
```

The values of associative array elements are saved in the state file, so are retained across spec sessions, unless starting fresh.

All built-in global arrays, such as those that hold motor positions and scaler counts, are associative arrays.

A number of built-in functions take associative arrays as arguments or return associative arrays as results. For example, `split()`, `rsplit()` and `match()` place results in an associative array passed as an argument. The `move_info()` function returns a one- or two-dimensional associative array depending on how it is called. The `fmt_read()` and `fmt_write()` functions pass file header elements using associative arrays.

The internal code always uses the string value for the index of an associative array. Thus, `arr["12"]` refers to the same element as `arr[12]`, but `arr[012]` is not the same element as either `arr[12]` or `arr["012"]`. The value 012 is an octal number, which is equal to a decimal 10, so `arr[012]` is the same element as `arr[10]` or `arr["10"]`.

Data Arrays

The second kind of array is the *data* array. While associative arrays are indexed by arbitrary strings or numbers and can store either strings or numbers, data arrays are indexed by consecutive integers (starting from zero, as is the C convention) and hold a specific data type, such as short integer, float, double, etc.

Data arrays must be specifically declared and dimensioned using the `array` keyword (unlike associative arrays, which can come into existence when used in an expression). The arrays can have one or two dimensions. The `mca_get()` and `image_get()` functions can directly fill arrays with data from one- or two-dimensional detectors.

Data arrays can be used in expressions containing the standard arithmetic operators to perform simultaneous operations on each element of the array. In addition, a subarray syntax provides a method for performing assignments, operations and functions on only portions of the array.

The functions `array_read()`, `array_dump()`, `array_copy()`, `array_op()`, `array_plot()`, `array_fit()` and `array_pipe()` handle special array operations. The functions `fmt_read()` and `fmt_write()` transfer array data to and from binary-format data files. The functions `mca_get()`, `mca_put()`, `image_get()` and `image_put()` accept array arguments.

The `print` command will print data arrays in a concise format on the screen, giving a count of repeated columns and rows, rather than printing each array element.

Array data can be placed in shared memory, making the data accessible to other processes, such as image-display or data-crunching programs. The shared arrays can both be read and written by the other processes. The implementation includes a number of special aids for making the processes work smoothly with spec.

The data values of data-array elements are not saved in the user's state file, unlike associative array elements.

Data Array Usage

Data arrays must be declared with the `array` key word. One- and two-dimensional arrays are declared as:

```
[shared] [type] array var[cols]
[shared] [type] array var[rows][cols]
```

On platforms that support System V Interprocess Communication (IPC) calls, the shared keyword causes spec to place the array in shared memory (see below). The type keyword specifies the storage type of the array and may be one of `byte`, `ubyte`, `short`, `ushort`, `long`, `ulong`, `long64`, `ulong64`, `float`, `double` or `string`. An initial `u` denotes the "unsigned" version of the data type. The `short` and `ushort` types are 16-bit (two-byte) integers. The `long` and `ulong` types are 32-bit (four-byte) integers. The `long64` and `ulong64` types are 64-bit (eight-byte) integers. The `float` type uses four bytes per element. The `double` type uses eight bytes per element. The default data type is `double`.

The array name *var* is an ordinary spec variable name. Arrays are global by default, although they may also be declared local within statement blocks.

Unlike traditional spec associative arrays, which can store and be indexed by arbitrary strings or numbers, a data array is indexed by consecutive integers (starting from zero), and can hold only numbers, or in the case of `string` arrays, only strings.

Operations on these arrays can be performed on all elements of the array at once, or on one or more blocks of elements. Consider the following example:

```
array a[20]
a = 2
a[3] = 3
a[10:19] = 4
a[2,4,6,10:15] = 5
```

The first expression assigns the value 2 to all twenty elements of the array. The second expressions assigns 3 to one element. The third assign the value 4 to the tenth through last element. The final expression assigns the value 5 to the elements listed.

A negative number as an array index counts elements from the end of the array, with `a[-1]` referring to the last element of `a`.

As per the usual conventions, the first index is row and the second is column. Note, however, spec considers arrays declared with one dimension to be a single row. For example,

```
array a[20]
```

is a one-row, twenty-column array. Use

```
array a[20][1]
```

to declare a 20-row, one-column array.

Also note well, all operations between two arrays are defined as element-by-element operations, not matrix operations, which are currently unimplemented in spec. In the following example:

```
array a[5][5], b[5][5], c[5][5]
c = a * b
```

`c[`*`i`*`][`*`j`*`]` is the product `for each` *`i`* `and` *`j`*.

When two array operands have different dimensions the operations are performed on the elements that have dimensions in common. In the case:

```
array a[5][5], b[5], c[5][5]
c = a * b
```

only the first row of `c` will have values assigned, since `b` only has one row. The remaining elements of `c` are unchanged by the assignment.

Portions of the array can be accessed using the subarray syntax, which uses colons and commas, as in the following examples:

```
array a[10][10]
a[1]                # second row of a
a[2:4][]            # rows 2 to 4
a[][2:]             # all rows, cols 2 to last
a[1,3,5,7,9][3:7]   # odd rows and cols 3 to 7
```

The elements of an array can be accessed in reverse order, as in:

```
a = x[-1:0]
```

which will assign to a the reversed elements of `x`. Note, though, that presently, an assignment such as `x = x[-1:0]` will not work properly, as spec will not make a temporary copy of the elements. However, `x = x[-1:0]+0` will work.

The functions `fabs()`, `int()`, `cos()`, `acos()`, `sin()`, `asin()`, `tan()`, `atan()`, `exp()`, `exp10()`, `log()`, `log10()`, `pow()` and `sqrt()` can all take arrays as an argument. The functions perform the operation on each element of the array argument and return the results in an array of the same dimension as the argument array.

The operations `<`, `<=`, `!=`, `==`, `>` and `>=` can be used with array arguments. The Boolean result (0 or 1) will be assigned to each element of an array returned as the result of the operation, based on the element-by-element comparison of the operands.

The bit-wise operators `~`, `|`, `&`, `>>` and `<<` can also be used with array operands.

Note, spec generally uses double-precision floating point for storing intermediate values and for mathematical operations. Double-precision floating point has only 52 bits for the significand (the remaining 12 bits are for sign and exponent). Thus, for most operations the 64-bit types will only maintain 52 bits of significance. (The 64-bit integer types were added in spec release 6.01.)

String Data Arrays

Arrays of type `string` are identical to `byte` arrays in terms of storage requirements and behavior in most operations. However, when used as described below, the string arrays do behave differently.

If a row of a string array represents a number and is used in a conditional expression, then the value of the conditional expression will be the number. For example, the strings `0.00` or `0x000` will evaluate as zero or false in a conditional expression. In contrast, for number arrays, a conditional evaluates as zero only if every element of the array is zero.

Functions that take string arguments, such as `on()`, `length()`, `unix()`, etc., will allow a row of a string array to be used as an argument. Use of a number array is invalid and produces an error.

The `print` command will print string arrays as ASCII text, while byte arrays display each byte as a number.

In assignments to a row of a string array, the right hand side is copied to the byte elements of the string array as a string, even if the right hand side is a number. Any remaining elements of the string array row are set to zero. Thus, the results differ in

the assignments below:

```
string array arr_string[20]
arr_string = 3.14159
print arr_string
3.14159

byte array arr_byte[20]
arr_byte = 3.14159
print arr_byte
{3 <20 repeats>}
```

In the first example, the string representation of the number is copied to the row of the string array, while in the second, each element of the array is assigned the (truncated) value of the number.

Row-wise and Column-wise Sense

For the functions `array_dump()`, `array_fit()`, `array_pipe()`, `array_plot()` and `array_read()` it matters whether each row or each column of a two-dimensional array corresponds to a data point. By default, spec takes the larger dimension to correspond to point number, and if both dimensions are the same, to use the rows as data points. The `row_wise` and `col_wise` arguments to `array_op()`, described below, can be used to force the sense of an array one way or the other, regardless of the array dimensions. If an array has row-wise sense, the contents of each row correspond to a data point, and one might then plot the contents of column two of each row versus column one, for example.

Shared Memory Arrays

When created with the `shared` keyword, the array data and a header structure are stored in shared memory. For each shared memory array, spec creates an immutable global variable named `SHMID_`*`var`* whose value is the shared memory ID associated with the shared memory segment and where where *`var`* is the name of the array. This ID is used by other programs that wish to access the shared memory.

spec can connect to an existing shared memory array created by another process running on the same platform, perhaps created by another instance of spec. The syntax is:

```
extern shared array [spec:[pid:]]arr
```

where the optional parameter *`spec`* is the name of the spec version that created the array, the optional parameter *`pid`* is the process ID of the version and *`arr`* is the name of the array. The first two arguments can be used in case more than one

instance of the shared array exists. Examples include:

```
extern shared array data
extern shared array fourc:data
extern shared array fourc:1234:data
```

The shared array segments include a header that describes the array. Two features of the header that are primarily associated with shared arrays that can accessed from spec user level are tags and frames. Shared arrays tags can be manipulated with the `array_op()` `tag` and `untag` options, as described in the next section.

Frame-size and latest-frame header elements allow a shared 2D array to be described as a series of 1D or 2D acquisitions (or frames). The frame size is the number of rows in a single frame. The latest frame is the most recently updated frame number. The latest frame value should allow an auxiliary program that maintains a live display to update the display efficiently. The frame values are also accessed via `array_op()`. Currently, the frame values are unused by spec in array operations, although specific hardware support may modify frames values.

The structure used for the shared memory data is given in the file ***SPECD***/*include*/*spec_shm.h*. A C file containing an API for accessing the spec shared memory arrays is included in the spec distribution and is named *sps.c*.

Keywords

The following names are reserved, being either grammar keywords or the names of built-in commands or functions. New reserved names may be added. The list can be obtained using the built-in command `lscmd`. Parentheses after a name indicate a function.

```
acos()        data_put()   gpib_par()   open()        sock_put()
array         data_read()  gpib_poll()  plot_cntl()   spec_menu()
array_copy()  data_uop()   gpib_put()   plot_move()   spec_par()
array_dump()  date()       gsub()       plot_range()  split()
array_fit()   dcb()        h5_attr()    port_get()    splot_cntl()
array_op()    decode()     h5_data()    port_getw()   sprintf()
array_pipe()  def          h5_file()    port_put()    sqrt()
array_plot()  deg()        h5_link()    port_putw()   srand()
array_read()  delete       history      pow()         sscanf()
asc()         dial()       if           prdef         stop()
asin()        dofile()     image_get()  print         strdef()
atan()        double       image_par()  printf()      string
atan2()       else         image_put()  prop_get()    sub()
bcd()         em_io()      in           prop_put()    substr()
break         encode()     index()      prop_send()   syms
byte          eprint       input()      prop_watch()  sync
ca_cntl()     eprintf()    int()        qdofile()     tan()
ca_fna()      eval()       length()     quit          tcount()
```

```
ca_get()       eval2()       local         rad()           time()
ca_put()       exit          log()         rand()          tty_cntl()
calc()         exp()         log10()       rdef            tty_fmt()
cdef()         exp10()       long          read_motors()   tty_move()
chdir()        extern        long64        reconfig        ubyte
chg_dial()     fabs()        lscmd         remote_async()  ulong
chg_offset()   fbus_get()    lsdef         remote_cmd()    ulong64
clone()        fbus_put()    match()       remote_eval()   undef
close()        file_info()   mca_get()     remote_par()    unglobal
cnt_mne()      float         mca_par()     remote_poll()   unix()
cnt_name()     fmt_close()   mca_put()     remote_stat()   user()
cnt_num()      fmt_read()    mca_sel()     return          ushort
constant       fmt_write()   mca_sget()    rsplit()        vme_get()
continue       for           mca_spar()    savstate        vme_get32()
cos()          fprintf()     mca_sput()    ser_get()       vme_move()
counter_par()  gensub()      mcount()      ser_par()       vme_put()
data_anal()    get_lim()     memstat       ser_put()       vme_put32()
data_bop()     getcounts     motor_mne()   set_lim()       vxi11_get()
data_dump()    getenv()      motor_name()  set_sim()       vxi11_par()
data_fit()     gethelp()     motor_num()   shared          vxi11_put()
data_get()     getline()     motor_par()   short           wait()
data_grp()     getsval()     move_all      sin()           whatis()
data_info()    getval()      move_cnt      sleep()         while
data_nput()    global        move_info()   sock_get()      yesno()
data_pipe()    gpib_cntl()   off()         sock_io()
data_plot()    gpib_get()    on()          sock_par()
```

Numeric Constants

Numeric constants can be integers or floating point numbers. Integer constants are considered octal if the first digit is `0`. Valid digits in the rest of the constant are `0` through `7`. A hexadecimal constant begins with `0x` or `0X` and is followed by digits or the letters `a` or `A` through `f` or `F`, which have values 10 through 15. Otherwise, a sequence of digits is a decimal constant.

Floating-point constants have an integer part, a decimal point, a fraction part, an `e` or `E` and an optionally signed exponent. The integer part or the fraction part, but not both, may be missing. The decimal point or the `e` and exponent, but not both, may be missing.

As spec stores number values internally in double-precision format, the range of integer and floating constants is determined by the range of double-precision numbers. With the usual 64 bits allocated for a double number, the significand uses 52 bits and the sign and exponent use 12 bits. Although signed integers can have values from $\pm 2^{63}$ or unsigned values from 0 to 2^{64}, the values will only have 52 significant bits.

The following are valid numeric constants.

```
65535  0177777    0xFFFF
+1066  1.066e3  1.066e+3
```

String Constants

Strings are delimited by pairs of single or double quotes. The following escape sequences, introduced by a backslash, can be included within strings to represent certain special characters:

`\a`	attention, audible alert (bell)
`\b`	back space
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\\`	backslash
`\'`	single quote
`\"`	double quote
`\`*ooo*	octal code
`\[`*xx*`]`	tty control code

Tty control codes are only recognized when embedded in strings passed to spec's built-in functions `tty_move()` and `tty_fmt()`. The recognized strings for *xx* are described in the description of the `tty_cntl()` function on page 94.

For any other character *x*, `\`*x* is just that character. The sequence `\`*ooo* represents one to three octal digits that have the ASCII value of a single character. For example, `\033` represents the *escape* character.

A character string can be continued over more than one line by using a `\` at the end of a line. On the other hand, new lines not preceded with a `\` are inserted literally into the string.

String Patterns and Wild Cards

For the commands `lscmd`, `lsdef`, `prdef` and `syms`, if the characters `?` or `*` appear in the arguments, the argument is taken as a pattern. Only information about those commands, macros or symbols that match the pattern is displayed. In forming the pattern, the character `?` matches any single character, while `*` matches any string of

characters, including the null string.

Tilde Expansion

The tilde expansion feature for path names replaces the tilde character at the start of a path name, as in `~/`, with the current user's home directory path, and replaces `~user/` with any *user*'s home directory path. These spec commands and functions do tilde expansion on path-name arguments: `chdir()`, `unix()`, `file_info()`, `open()`, `close()`, `on()`, `off()`, `dofile()`, `qdofile()`, `getline()`, `fprintf()`, `array_dump()`, `array_read()`, `data_dump()`, `data_read()`, `fmt_read()`, `fmt_write()`, `fmt_close()` and `h5_file()`. In addition, the spec server will do tilde expansion on the *filename* received when a client registers for `output/`*filename* property events.

Command Recall (History)

The basic spec history feature lets you recall previous commands[1]. Examples of the recognized syntax are:

`!!`	Redo the previous command.
`!14`	Redo command number 14.
`!-2`	Redo the second to previous command.
`!asc`	Redo the last command that began with `asc`.
`!asc -10000`	As above and append `-10000` to the command.
`history`	List the last 1000 commands.
`history N`	List the last *N* commands.
`history -N`	List the last *N* commands in reverse order.

The command number is prepended to the spec prompt as an aid in using the history feature. Only commands typed at the keyboard are remembered for history. By default, the previous 1000 commands are retained. The number of remembered commands can be changed using the `spec_par() "history_size"` option. See `spec_par()` on page 85. The history feature cannot be used in command files.

With the basic history feature, command recall must occur at the beginning of a line, although initial white space is allowed. Text may follow the command-recall word to

[1]spec is usually installed with the optional BSD *libedit* history library. With the *libedit* library, the history syntax is greatly expanded. In addition, features such as command-line editing, and command completion become available. See the on-line *libedit* help file for detailed information.

extend that command.

Appending `:s/left/right/` to a recalled command will modify the first occurrence of the string `left` in the recalled command to the characters `right`. The delimiter of the left and right strings may be any character. The final delimiter is optional. If `left` is empty, the last entered left string is used as the pattern to match.

In addition, `^left^right^` at the start of a line is shorthand for `!-1:s/left/right/`. In this case, the circumflex (`^`) must be used as the delimiter. The final delimiter is optional.

The history is saved along side the state file when exiting spec. Restarting spec reads in the saved history.

Starting Up

When you run spec, you invoke it using a name such as *fourc*, *twoc*, *surf*, *spec*, etc. That name determines the kind of geometry code that will be available and which macro and configuration files in the auxiliary file directory will be used, as explained below.

The following command line options are recognized by spec:

–C ***file*** – Open the command file *file* as a start-up command file to be read after the standard start-up command files, but before the optional file *spec.mac* in the current directory, which will always be read last. If there is an error in reading or executing the commands in these files, spec will jump to the main prompt and not read any remaining queued command files. Up to 32 files may be specified with multiple **–C** options.

–d ***debug*** – Sets the initial value of the debugging variable `DEBUG` to ***debug***, which maybe either in decimal or hexadecimal (with a leading 0x) format. The available debugging categories are described on page 63.

–D ***direc*** – Use ***direc*** as the auxiliary file directory, instead of the compiled-in name (usually */usr/local/lib/spec.d*) or the value of the *SPECD* environment variable.

–f – Fresh start. All symbols are set to their default values and the standard macros are read to establish the default state. Command-line history is reset unless the **–h** flag is also present.

–F – Clean and fresh start. All symbols are set to their default values but no command files are read and no macros are defined. Only the built-in commands are available.

–g ***geometry*** – Force loading of macro files and activation of geometry calculations for the specified geometry, while using the configuration files taken from the name by which spec is invoked.

–h – Retain history. When starting fresh, reset symbols and macros but keep command-line history. (Added in spec release 6.05.01.)

–l ***outputfile*** – Specify an output file. Output to the file will begin immediately, so will include the initial hardware configuration messages. Files will be opened even when starting fresh. Files opened this way will not be saved as output files in the state file, so will not be automatically reopened the next time spec starts (as of spec release 6.04.05).

–L – Do not check or create the state-file lock. Normally, spec prevents more than one instance of itself from running with the same state file (derived from the user name plus tty name). With some system configurations, if the state file resides on an NFS-mounted disk, the file locking doesn't work well and spec will not start. This flag overrides the lock test.

–N ***my_name*** – Use ***my_name*** instead of the name by which spec was invoked to establish the command prompt and the name of the directory in ***SPECD*** in which the configuration-dependent files exisit. This command also sets the geometry to ***my_name***. Follow this option with the **–g** option to choose a different name for the geometry.

–o ***option=value*** – Initializes the `spec_par()` *option* to *value*. The available `spec_par()` options are described on page 83.

–p ***fd pid*** – Indicates that spec input is coming from a pipe from another program. The argument ***fd*** is the file descriptor that spec should use for standard input. The argument ***pid*** is the process ID of the spawning process. If ***fd*** is zero, spec will not re-echo input from the file descriptor to spec's standard output.

–q – Indicates that spec should operate in quiet mode and allow output to all devices to be turned off. This option is only valid when used with the **–p** option.

–s – Simulation mode. No hardware commands are issued. Simulation mode cannot be turned off after entering the program.

–S – Start spec in server mode listening at the first available port in the default range of 6510 to 6530.

–S ***p1*** – Start spec in server mode listening at the specified port number *p1*.

–S ***p1-p2*** – Start spec in server mode listening on the first available port in the given range.

–t ***tty*** – Use the current user (or ***user***'s) last saved state from the terminal specified by ***tty*** as the initial state. The terminal can be specified as **–t** ***/dev/tty01*** or **–t** ***tty01***. Pseudo-tty names, such as */dev/ttyp0*, */dev/ttyp1*, etc., are saved as */dev/ttyp#*, since there is no special significance to the number.

–T ***fake_tty*** – This option creates a user state associated with ***fake_tty***, which may be any name. This option allows you to bypass the locking feature that prevents multiple instances of spec to be started by the same user from the same terminal.

–u ***user*** – Use ***user***'s last saved state as the current user's initial state.

–v – Print version information and exit.

–y – Yes, change motor controller registers initially if they disagree with the *settings* file. Normally, spec requires you to confirm such a change. This flag would be useful if you know controller power had been turned off, and the controller's current positions should be updated with the software positions.

In some installations, spec is installed as a set-user-id root process, to allow certain calls that allow privileged access to hardware device registers. The first thing spec does on start up is to set the effective user and group ids to that of the real user, so there is no danger of the user spawning subshells or creating files as root. The root effective id is only used for the duration of the calls that enable the privileged access.

spec then performs other initialization tasks, including obtaining values for its internal variables `DISPLAY`, `GTERM`, `TERM` and `HOME` from variables of the same name in the process environment. It also obtains the value of the `SHELL` environment variable for use with the `unix()` function.

spec then reads the hardware configuration from the appropriate *config* file from the auxiliary file directory. The path name of that file is ***SPECD***/***spec***/*config*, where ***SPECD*** is the auxiliary file directory, established when spec is installed (or by the **–D** invocation flag, or by the *SPECD* environment variable), and ***spec*** is the name by which spec is invoked.

The first time a user starts spec, up to seven macro files are automatically read. The path names of these files are

SPECD/*standard.mac*
SPECD/***geom***.*mac*
SPECD/***spec***/***geom***.*mac*
SPECD/*site_f.mac*
SPECD/*site.mac*
SPECD/***spec***/*conf.mac*
./*spec.mac*

where ***SPECD*** is the auxiliary file directory, as described above, ***geom*** matches the first four letters of the name by which spec was invoked and ***spec*** is the complete name by which spec was invoked. The files are only read if they exist. The files ***SPECD***/*standard.mac*, ***SPECD***/***geom***.*mac*, ***SPECD***/***spec***/***geom***.*mac* and ***SPECD***/*site_f.mac* are only read if the user is starting spec for the first time or has invoked spec with the **–f** (fresh start) flag.

Each time spec starts up, if a macro named `begin_mac` is defined, that macro will be run after reading any startup command files, but before input is read from the keyboard.

After reading the start-up macro or command files and possibly running `begin_mac`, spec prompts the user for commands.

Keyboard Interrupts

spec responds to two different asynchronous signals that can be sent from the keyboard to programs. These signals are *interrupt* and *quit*. A **`^C`** is usually used to generate the *interrupt* signal, while a `^\` usually generates the *quit* signal. The control key assignments are, in principle, arbitrary and can be changed using the *stty* command from the UNIX shell. To display the current key assignments, type **`stty -a`**.

With spec, the *interrupt* key halts all activity, including asynchronous motor motion or counting, and closes all command files. All output files and devices (except *log* files) are closed. On keyboard interrupts (and command and syntax errors), *cleanup* macros, as described below may be run.

Typing the *quit* character will asynchronously terminate spec without saving the user's state. However, if motors are moving, the program will wait for them to halt and then update the *settings* file. If spec appears hung while waiting for hardware to finish some activity, type the *interrupt* or *quit* characters once or twice more, which should cause spec to give up on waiting and quit.

Cleanup Macros

On keyboard interrupts (and command and syntax errors), if macros named `cleanup_once` or `cleanup_always` have been defined, their definitions are read as input. Typical uses of the `cleanup_once` macro are to return motors to starting positions on an interrupted delta scan or to display counter contents if counting is interrupted.

After running the `cleanup_once` and/or `cleanup_always` macros, spec gives the standard prompt and waits for the next command from the keyboard. The `cleanup_once` macro is removed before the next main prompt is issued, whether or not it was triggered.

Definitions for these macros should be constructed using the `cdef()` (chained macro) function (described on page 102) in order to allow various clean-up actions to be added and removed during the flow of arbitrary statement blocks.

The legacy cleanup macros `cleanup` and `cleanup1` remain supported. Like `cleanup_always`, the macros remain defined across the standard prompt. However, unlike `cleanup_always`, both `cleanup` and `cleanup1` are deleted if they generate an error or the commands contained therein are interruped by **`^C`**.

Exiting

A spec session is normally terminated by typing **`^D`** at the start of an input line. The `quit` command will also terminate spec, but only when typed directly from the keyboard or when read from a command file. The `quit` command can't be included in a macro.

If there is a macro named `end_mac` defined, it will be run after the **`^D`** or `quit` command.

When exiting, spec saves the user's state, and if any motors are moving, waits for them to halt and then updates the *settings* file.

If spec appears hung, typing the *quit* character (often `^\`) should terminate the program. If spec is waiting for unresponsive hardware to indicate it is finished with a move or a count, typing the *quit* or *interrupt* character again may be necessary.

Variables

A variable is brought into existence simply by using it. Variables assigned values at the top level (outside of any curly-bracketed statement block) are automatically made global. Otherwise, unless explicitly given *global* attributes, the scope of a variable lies only within the statement or statement block in which it occurs.

The possible attributes of a variable are as follows:

Local	A symbol with scope only in the statement block in which it appears.
Global	A symbol with scope that carries across separate statement blocks. All built-in symbols are global.
Built-in	A symbol that is compiled into spec and that cannot be removed.
Constant	A global symbol that cannot have its value altered by ordinary assignment. Such a symbol can only be changed using the `constant` command.
Immutable	Certain built-in symbols and motor and counter mnemonics from the hardware *config* file, which cannot have their values altered.

Variables can have string, number or array type and may have both string and number types simultaneously. The `print` command always prints the string representation of a variable. The formatted printing commands `printf`, `fprintf` and `sprintf` print the string or number value, depending on the format specification. If the string value cannot be interpreted as a number, its number value is zero. All number values are maintained as double-precision floating-point quantities.

Uninitialized variables have a number value of zero and a string value of `""` (the null string). Although associative array indices are internally derived from the string value of the index expression, if the index is an uninitialized variable, its value for purposes of indexing the array is the string `"0"`.

Functions such as `input()`, `getline()`, `gpib_get()` and `ser_get()` return string values that possibly represent numbers. When the string is used in a number context, automatic string to number conversion takes place. The conversion rules require that there are no extraneous characters in the string. An initial `0x` or `0X` introduces a hexadecimal integer. An initial `0` introduces an octal constant, unless there is a decimal point or an `e` or `E` exponential string, in which case the entire string is interpreted as a decimal number, leading zero notwithstanding.

Built-In Variables

The following is a list of most of spec's built-in variables. Some site-dependent code, along with most of the standard diffractometer geometry code, will create additional built-in variables. Also, motor and counter mnemonics entered in the hardware configuration file become built-in variables when the *config* file is read by spec.

`A[]` — is an array dimensioned to the number of motors as obtained from the *config* file. The function `read_motors(0)` fills the array with user angles. The user can assign values to any of the elements. The commands `move_all` and `move_cnt` use the values in the array. Also, the various site-dependent, geometry-specific calculations, accessed through the `calc()` user-hook function, base their results on the values in this array or place new values in it.

`CCDS` — is the number of 2D image-type devices being used as determined from reading the *config* file.

`COLS` — is a number-valued variable set to the number of text columns on the user terminal or workstation window. The value is used for formatting text-mode plots and on-line help files. `COLS` is generally automatically assigned a value from the system terminal data base when spec starts up, or, if available, by using the `TIOCGWINSZ` command in the *ioctl()* system call whenever a value for `COLS` is needed.

`COUNTERS` — is the number of counters being used as determined from reading the *config* file.

`CWD` — is a string-valued variable that contains the name of the user's current working directory. It is assigned a value when spec starts up, and is updated each time the `chdir()` function is executed.

`DEBUG` — is a user-assignable numeric variable that determines the level of debugging messages printed. The level is determined by the sum of the values given in this table:

Hex	Decimal	What is shown
0x1	1	Show input tokens during parsing.
0x2	2	Show node execution while running.
0x4	4	Show node allocate and free.
0x8	8	Show symbol table creation and lookup.
0x10	16	Show value get and set.
0x20	32	Show misc info.
0x40	64	Show hardware related info.
0x80	128	Show more hardware related info.
0x100	256	Show macro substitution.
0x200	512	Show memory allocate and free.
0x400	1024	Show input after macro expansion.
0x800	2048	Print warning messages.
0x1000	4096	Show certain low level hardware info.
0x2000	8192	Show data array allocation.
0x4000		Show signal blocking.
0x8000		Show sleeps and other low level hardware info.
0x10000		Show input file queuing.
0x20000		Show readable runtime debugging.
0x40000		Print input context on execution-time errors.
0x40000		Print input context on execution-time errors.
0x80000		Show sleeps.
0x100000		Show thread stuff.
0x200000		Show state changes.
0x400000		Use hexadecimal for socket debugging output.
0x800000		Show server/client socket messages.

If a debugging log file is open (any file that begins with the characters `dlog` or ends with the characters `.dlog`), debugging messages are only written to that file, not to the screen or any other file or device.

`DISPLAY` — is a user-assignable, string-valued variable. Its value at the time an X-Windows graphics filter process is spawned with the `plot_cntl("open")` function (with `GTERM` set to `"x11"`) determines on which host and screen the plot window will be displayed. The initial value for `DISPLAY` is taken from the environment variable of the same name.

`EVAL_ERR` — is a built-in variable that will be assigned error messages generated by errors from parsing or executing strings passed to the `eval()` or `eval2()` functions. See the `eval()` function on page 80, for details.

`EVAL_RESULT` — is a built-in variable with a value set by the `eval()` or `eval2()` functions. If `EVAL_RESULT` is −1, there was an error parsing or executing the

command string, or an attempt was made to return the value of a single uninitialized variable. If EVAL_RESULT is 1, the command string was a statement or was a statement list that ended in a statement. If EVAL_RESULT is zero, the command string or last statement in the string was an expression. See the eval() function on page 80, for details.

FRESH — is a built-in variable that has an initial nonzero value if spec was invoked with the **–f** (fresh start) flag or if a fresh start was forced by an incompatible state file version. The value is zero otherwise, and is set to zero in any case after all start-up command files and macros have been read and their commands executed. In the standard start-up macros, the value of FRESH is checked to see if initial default parameter assignments should be made.

GETLINE_EOF — is a built-in variable set by the getline() function to distinguish between an end of file, an error or a literal –1 read from the file. The value of GETLINE_EOF will be 1 if there was an end-of-file condition on the read of the file, –1 if there was an error reading the file or if the file couldn't be opened and zero if the read was successful. See the getline() function on page 90.

GTERM — is a user-assignable, string-valued variable containing a value describing the display type to use for high-resolution graphics. Its value will be taken from an environment variable of the same name when spec starts up, if such a variable exists. The only currently supported GTERM value is x11 for X Window System graphics. If GTERM is not set in the environment or has not been assigned a value, it defaults to x11. Legacy values of vga, ega, herc, sun, etc. are no longer supported.

HDW_ERR — is a user-assignable number-valued variable that can be set before hardware calls to affect how spec responds to certain errors. After hardware calls, the value of HDW_ERR conveys information on any errors encountered. By default, certain hardware errors cause spec to reset to command level. By setting HDW_ERR to –1 before a call to user-level hardware access functions, there is no reset, the function returns and HDW_ERR contains an error code.

Functions which set HDW_ERR include counter_par(), motor_par(), mca_par(), ca_get(), ca_put(), ca_fna(), gpib_get(), gpib_put(), gpib_poll(), gpib_cntl(), gpib_par(), port_get(), port_getw(), port_put(), port_putw(), ser_get(), ser_put(), ser_par(), sock_get(), sock_put(), sock_par(), fbus_get(), fbus_put(), vxi11_get() and vxi11_put().

When there is an error in these functions, they usually return –1, or sometimes zero. However, such values can also be valid return values. If one is doing error checking, one should check HDW_ERR after the function call. The following table lists the possible values for HDW_ERR and what each value indicates:

1 Generic error
2 GPIB no listener
3 Timeout
4 A non-fatal error
5 Function called with bad argument
6 Trying to access unconfigured hardware
7 Function called with bad address
8 Lost connection
9 User abort (`^C`)
10 Unresponsive hardware
11 Disabled hardware
12 Bad parameter
13 Read only parameter
14 Write only parameter

If the value of `HDW_ERR` is −1 before the call to the user-level hardware access function, no reset to command level will take place for any errors. Otherwise, traditionally fatal errors will still be fatal.

`HOME` − is string valued and is initialized to the user's home directory as taken from the environment variable *HOME*. If not found in the environment, its value is set to the current directory.

`HOSTNAME` − is a built-in variable containing the platform host name as returned by the *gethostname()* system call.

`IS_SERVER` − is a built-in variable that contains the TCP port number on which spec is listening if running in server mode. Otherwise, the value is zero.

`MCAS` − is the number of 1D MCA-type devices being used as determined from reading the *config* file.

`MOTORS` − is the number of motors being used as determined from reading the *config* file.

`OUTFILES[][]` − is a two-dimensional associative array that holds information about all open output files. The first index is the name by which the file was opened using `open()`, `on()`, `fprintf()` or the `-l outputfile` start-up option. Currently, only two options are available for the second index: `"name"` and `"path"`. The `"name"` element repeats the name used for the first index. The `"path"` element is the full path name of the file. Additional elements may be added in the future. For example:

```
5.FOURC> for (i in OUTFILES[]["name"])
6.more>      printf("%10s %s\n",OUTFILES[i]["name"],OUTFILES[i]["path"])
      dlog /tmp/dlog
     data1 /tmp/data1
       tty /dev/tty
 /dev/null /dev/null

7.FOURC>
```

or

```
7.FOURC> p OUTFILES[DATAFILE]["path"]
/private/tmp/data1

8.FOURC>
```

Note, the special built-in name "pipe" is not included in OUTFILES. The special built-in name "/dev/null" includes the full path in both "name" and "path" to work with the standard macros that always refer to that special file by the full path name.

PI — is a number-valued symbol with the value 3.14159... .

ROWS — is a number-valued variable set to the number of text rows on the user terminal or workstation window. The value is used for formatting text-mode plots and on-line help files. ROWS is generally automatically assigned a value from the system terminal data base when spec starts up, or, if available, by using the TIOCGWINSZ command in the *ioctl()* system call whenever a value for ROWS is needed.

S[] — is an array that will be filled with the hardware scaler contents when the command getcounts is executed.

SPEC — is string valued and set to the name by which spec is invoked, such as *fourc*.

SPECD — is string valued and set to spec's auxiliary file directory. The default name is compiled in when spec is installed, but can be overridden by the **–D** invocation option or by the *SPECD* environment variable.

TERM — is a user-assignable, string-valued variable. It is initialized to the user's terminal type as taken from the environment variable *TERM*. If not found in the environment, it is set to terminal-type *vt100*.

USER — is string valued and is set to the login name of the current user.

VERSION — is string valued and is set to the version number of spec, as in 3.03.11.

The values of A[], S[], DEBUG, HDW_ERR, DISPLAY, TERM, GTERM, ROWS and COLS can be changed by the user.

Motor mnemonics obtained from the *config* file become built-in, immutable variables. User-added code, such as the X-ray diffractometer geometry code, typically creates other built-in variables, such as `G[]`, `Q[]`, `U[]` and `Z[]`.

Operators

The following tables summarize the operators available in spec. (Almost all these operators work the same as in the C language, so a C-language reference manual could be consulted to provide more detailed information on the use of unfamiliar operators.) Operators that require integral operands use the integer part of nonintegral operands. The precedence rules give the evaluation order when multiple operators appear in the same expression.

spec stores number values as double precision floating point, which only has 52 bits for the signifcand. The bitwise operators (`<<`, `>>`, `&`, `^`, `|`, `<<=`, `>>=`, `&=`, `^=`, `|=`) will mask the operands to 52 bits. The result of the operation will be no more than 52 bits. The bit-not operator (`~`) will mask the result to the low 52 bits. The modulus operators (% and %=) will convert the operands to 64 bits, perform the operation using 64-bit integer arithmetic, then convert the result to a double for the return value.

Unary Operators

Unary operators have equal precedence, are evaluated right to left and have higher precedence than the binary operators.

Operator	Name	Result
`-`	Unary minus	Negative of operand.
`+`	Unary plus	Operand.
`!`	Logical negation	1 if operand is zero, 0 otherwise.
`++`	Increment	Increment the operand by one.
`--`	Decrement	Decrement the operand by one.
`~`	Bitwise not	One's complement of operand. (A tilde.)

When used as prefix operators, ++ and `--` operate before the result is used in the expression. When used as postfix operators, the ++ and `--` operations are done after the current value of the operand is used in the expression.

Indirection Operator

The @ character is a special unary operator that provides indirection when used in front of a symbol name. The behavior is similar to the C language * indirection operator. In spec, the @ operator allows reference to a variable whose name is the string value of another variable. For example:

```
8.FOURC> a = "b"; b = PI; print a, @a
b 3.14159

9.FOURC>
```

Binary Operators

All binary operators have lower precedence than the unary operators. Binary operator precedence is indicated in the grammar rules that are listed on page 73, where higher precedence operators are listed first, and operators with the same precedence are listed on the same line. Binary operators with the same precedence are evaluated left to right.

Operator	Name	Result (**L** is left operand, **R** is right)
*	Multiplication	**L** × **R**
/	Division	**L** / **R**
%	Modulus	Remainder of **L** / **R** (operands are integers).
+	Addition	**L** + **R**
–	Subtraction	**L** – **R**
<	Less than	1 if **L** < **R** (or 0 if not).
>	Greater than	1 if **L** > **R** (or 0 if not).
<=	Less than or equal	1 if **L** ≤ **R** (or 0 if not).
>=	Greater than or equal	1 if **L** ≥ **R** (or 0 if not).
==	Logical equality	1 if **L** is equal to **R** (or 0 if not).
!=	Logical inequality	1 if **L** is not equal to **R** (or 0 if it is).
&&	Logical and	1 if both **L** and **R** are nonzero, otherwise 0.
||	Logical or	1 if either **L** or **R** are nonzero, otherwise 0.
<<	Bitwise left shift	**L** shifted left by **R** bits (both integers).
>>	Bitwise right shift	**L** shifted right by **R** bits (both integers).
&	Bitwise and	“Bitwise and” of integers **L** and **R**.
^	Bitwise exclusive or	“Bitwise exclusive or” of integers **L** and **R**.
|	Bitwise or	“Bitwise or” of integers **L** and **R**.
	Concatenation	**LR**

If either of **L** or **R** are strings, the relational operators <, >, <=, and >= use the lexicographic comparison provided by the C subroutine *strcmp()*.

The concatenation operator comes into effect when expressions are combined separated only by space characters. The resulting expression is the concatenation of the string values of the constituent expressions. Concatenation is only allowed on the right side of an assignment operator, in the arguments to the `print` command and in the value assigned to a variable in a `constant` statement. Concatenation has lower precedence than the other operators. For example,

```
9.FOURC> print "ab" "cd"  1 2 + 3 4
abcd154

10.FOURC>
```

Assignment Operators

Operator	Name	Result
=	Equals	**L** = **R**
+=	Plus equals	**L** = **L** + **R**
−=	Minus equals	**L** = **L** − **R**
*=	Times equals	**L** = **L** × **R**
/=	Divide equals	**L** = **L** / **R**
%=	Mod equals	**L** = **L** % **R**
<<=	Left-shift equals	**L** = **L** << **R**
>>=	Right-shift equals	**L** = **L** >> **R**
&=	Bitwise-and equals	**L** = **L** & **R**
^=	Bitwise-exclusive-or equals	**L** = **L** ^ **R**
|=	Bitwise-or equals	**L** = **L** | **R**

Ternary Operator

spec, like the C language, has one ternary operator, which works in a manner similar to the if-else construction and uses the characters ? and : . Its use is

$$expression_1 \; ? \; expression_2 \; : \; expression_3$$

where the result of this entire expression is $expression_2$ if $expression_1$ is nonzero, otherwise $expression_3$.

Flow Control

Conditional Statement

The forms of the conditional statement are

```
if ( expression ) statement
if ( expression ) statement else statement
```

The *expression* is evaluated in both cases. If nonzero, the first *statement* is executed. In the second form, if *expression* is zero, the second *statement* is executed. Each `else` is always matched with the last else-less `if` within the same statement block.

While Statement

The form for the `while` statement is

```
while ( expression ) statement
```

The *expression* is evaluated and the *statement* executed until *expression* is zero.

For Statement

The `for` statement comes in two forms. The first is

```
for ( expression1 ; expression2 ; expression3 ) statement
```

The flow expressed in this statement can be thought of as

```
expression1
while ( expression2 ) {
      statement
      expression3
}
```

Any of the expressions can be missing. A missing *expression*$_2$ is equivalent to

```
while ( 1 )
```

The second form of the `for` statement is used with associative arrays. The construction

```
for ( identifier in assoc-array ) statement
```

will run through each element of the associative array *assoc_array* assigning to *identifier* the string value of the index of each element. The elements will be sorted using a “natural sort”. "Natural sort" means that consecutive digits are treated as a single character and sorted according to their value as a group, such that *a10* will come after **a9*, contrary to the order with strict lexicographical sorting.

For two-dimensional associative arrays, the construction

```
for ( identifier in assoc-array[expr] ) statement
```

will step through each element of *assoc-array* having *expr* has the first index.

Break Statement

The statement `break` terminates the smallest enclosing `while` or `for` loop.

Continue Statement

The statement `continue` passes control to the loop-test portion of the smallest enclosing `while` or `for` loop.

Exit Statement

The statement `exit` terminates execution of the current parse tree and jumps control back to command level just as if an error occurred or a **^C** was typed.

Grammar Rules

This syntax summary defines how all the built-in keywords, operators and commands of spec can be combined. These grammar rules are similar to those given in standard C-language manuals. Operators are listed in order of precedence, with the highest precedence operators listed first.

The following terms are used in the grammar rules:

lvalue	- "Left value", something on the left side of an equals sign.
binop	- A binary operator (+, –, etc.).
asgnop	- An assignment operator (=, +=, etc.).
assoc-array	- An associative (original-style) array.
assoc-elem-list	- A *space*- or *comma*-separated list of associative array elements.
identifier	- A variable.
identifier-list	- A *space*- or *comma*-separated list of identifiers.
pattern	- An alphanumeric string possibly containing the metacharacters `?` or `*`.
pattern-list-opt	- An optional *space*-separated list of patterns.
expression-list	- A *comma*-separated list of expressions.
expr-opt	- An optional expression.
`[;]`	- A semicolon or a newline. (A semicolon after a statement is optional if the statement is followed by a newline.)

Note, that in the following list, the entry

expression `in` *assoc-array*

is included in the rules of what constitutes an *expression*. This is a special expression that evaluates to nonzero (or **true**) if *assoc-array*`[`*expr*`]` is an existing element of the array, and zero (or **false**) otherwise. For a two-dimensional associative array,

$expr_1$ `in` *assoc-array*`[`$expr_2$`]`

is nonzero if *assoc-array*`[`$expr_2$`][`$expr_1$`]` is an element of the array.

These are the grammar rules:

expression:
lvalue
numeric-constant
string-constant
(*expression*)
function (*expression-list*)
– *expression*
! *expression*
~ *expression*
++ *lvalue*
–– *lvalue*
lvalue ++
lvalue ––
expression ? *expression* : *expression*
expression binop expression
expression in *assoc-array*
lvalue asgnop expression
expression , *expression*
expression expression

lvalue:
identifier
identifier [*expression*]
identifier [*expression*] [*expression*]

binop:
```
*    /    %
+    –
>>    <<
>    <    <=    >=
==    !=
&
^
|
&&
||
```

asgnop:
```
=    +=    –=    *=    /=    %=    >>=    <<=    &=    ^=    |=
```

data-array-type

```
string
byte
ubyte
short
ushort
long
ulong
long64
ulong64
float
double
```

data-array-declaration:

`array` *identifier*`[`*expression*`]`
data-array-type `array` *identifier*`[`*expression*`]`
`array` *identifier*`[`*expression*`][`*expression*`]`
data-array-type `array` *identifier*`[`*expression*`][`*expression*`]`

compound-statement:

`{` *statement-list* `}`

statement-list:

statement
statement statement-list

statement:

compound-statement
expression `[;]`
`if (` *expression* `)` *statement*
`if (` *expression* `)` *statement* `else` *statement*
`while (` *expression* `)` *statement*
`for (` *expr-opt* `;` *expr-opt* `;` *expr-opt* `)` *statement*
`for (` *identifier* `in` *assoc-array* `)` *statement*
`break [;]`
`continue [;]`
`exit [;]`
`history` *expr-opt* `[;]`
`lscmd` *pattern-list-opt* `[;]`
`print` *expression-list* `[;]`
`global` *identifier-list* `[;]`
`constant` *identifier expression* `[;]`

`constant` *identifier* `=` *expression* `[;]`
`unglobal` *identifier-list* `[;]`
`delete` *assoc-elem-list* `[;]`
`delete` *assoc-array* `[;]`
`local` *identifier-list* `[;]`
`syms` *pattern-list-opt* `[;]`
data-array-declaration `[;]`
`local` *data-array-declaration* `[;]`
`global` *data-array-declaration* `[;]`
`shared` *data-array-declaration* `[;]`
`extern shared` *data-array-declaration* `[;]`
`def` *identifier string-constant* `[;]`
`rdef` *identifier expression* `[;]`
`undef` *identifier-list* `[;]`
`prdef` *pattern-list-opt* `[;]`
`lsdef` *pattern-list-opt* `[;]`
`memstat [;]`
`savstate [;]`
`reconfig [;]`
`getcounts [;]`
`move_all [;]`
`move_cnt [;]`
`sync [;]`

Built-In Functions and Commands

These built-in functions and commands are described in the following sections.

<table>
<tr><th colspan="12">UTILITY FUNCTIONS AND COMMANDS</th></tr>
<tr><th colspan="2">System Functions</th><th colspan="10">Miscellaneous</th></tr>
<tr><td>chdir()
time()
getenv()</td><td>unix()
date()
file_info()</td><td colspan="10">lscmd
history
memstat
savstate
gethelp()
whatis()
calc()
eval()
eval2()
spec_par()
sleep()</td></tr>
<tr><th colspan="12">KEYBOARD AND FILE INPUT, SCREEN AND FILE OUTPUT</th></tr>
<tr><th colspan="2">Controlling Output Files</th><th>Reading From Files</th><th>Keyboard Input</th><th colspan="8">Text Output</th></tr>
<tr><td>open()
close()</td><td>on()
off()</td><td>getline()
dofile()
qdofile()</td><td>input()
yesno()
getval()
getsval()</td><td colspan="8">print
eprint
printf()
eprintf()
fprintf()
spec_menu()
tty_cntl()
tty_move()
tty_fmt()</td></tr>
<tr><th colspan="3">VARIABLES</th><th colspan="9">MACROS</th></tr>
<tr><td>global
unglobal</td><td>constant
local
syms</td><td>delete
array</td><td>def
rdef
cdef()</td><td colspan="5">prdef
lsdef
undef</td><td>clone()
strdef()</td></tr>
<tr><th colspan="12">STRING AND NUMBER FUNCTIONS</th></tr>
<tr><th colspan="3">Math Functions</th><th colspan="2">String</th><th colspan="4">Regular Expr</th><th colspan="2">Conversion</th></tr>
<tr><td>exp()
log()
exp10()
log10()
pow()
atan2()</td><td>srand()
rand()
sqrt()
int()
fabs()</td><td>cos()
sin()
tan()
acos()
asin()
atan()</td><td>index()
split()
substr()</td><td>length()
sprintf()
sscanf()</td><td colspan="4">rsplit()
sub()
gsub()
gensub()
match()</td><td colspan="2">asc()
bcd()
dcb()
deg()
rad()</td></tr>
<tr><th colspan="5">DATA HANDLING AND PLOTTING FUNCTIONS</th><th colspan="6">OLD-STYLE</th></tr>
<tr><td>array_dump()
array_read()
array_pipe()
array_plot()</td><td>array_copy()
array_op()
array_fit()</td><td>plot_cntl()
plot_move()
plot_range()
splot_cntl()</td><td>fmt_read()
fmt_write()
fmt_close()</td><td>h5_attr()
h5_file()
h5_link()
h5_data()</td><td colspan="6">data_grp()
data_info()
data_nput()
data_get()
data_put()
data_uop()
data_bop()
data_dump()
data_read()
data_pipe()
data_plot()
data_fit()
data_anal()</td></tr>
<tr><th colspan="5">CLIENT/SERVER FUNCTIONS</th></tr>
<tr><td>prop_send()</td><td>prop_watch()
prop_get()
prop_put()</td><td>remote_cmd()
remote_eval()
remote_async()</td><td>remote_poll()
remote_stat()
remote_par()</td><td>encode()
decode()</td></tr>
<tr><th colspan="12">HARDWARE FUNCTIONS AND COMMANDS</th></tr>
<tr><th colspan="4">Moving and Motors</th><th colspan="2">Counting</th><th colspan="6">Misc</th></tr>
<tr><td>move_all
move_cnt
sync</td><td>motor_mne()
motor_name()
motor_num()
motor_par()</td><td>read_motors()
dial()
chg_dial()
get_lim()</td><td>move_info()
user()
chg_offset()
set_lim()</td><td>mcount()
tcount()
getcounts</td><td>cnt_mne()
cnt_name()
cnt_num()
counter_par()</td><td colspan="6">reconfig
set_sim()
wait()
stop()</td></tr>
<tr><th colspan="2">MCA (1D)</th><th>Images (2D)</th><th colspan="3">Special Interfaces</th><th colspan="6">Sockets</th></tr>
<tr><td>mca_par()
mca_get()
mca_put()
mca_sel()</td><td>mca_spar()
mca_sget()
mca_sput()</td><td>image_par()
image_get()
image_put()</td><td>taco_io()
taco_db()
taco_dc()</td><td>tango_io()
tango_get()
tango_put()
tango_db()</td><td>epics_par()
epics_get()
epics_put()
em_io()</td><td colspan="6">sock_par()
sock_get()
sock_put()</td></tr>
<tr><th>Serial</th><th>GPIB</th><th>VME</th><th>PC Port I/O</th><th>VXI</th><th>Field Bus</th><th colspan="6">CAMAC</th></tr>
<tr><td>ser_par()
ser_get()
ser_put()</td><td>gpib_par()
gpib_get()
gpib_put()
gpib_poll()
gpib_cntl()</td><td>vme_move()
vme_get()
vme_put()
vme32_get()
vme32_put()</td><td>port_get()
port_getw()
port_put()
port_putw()</td><td>vxi11_par()
vxi11_get()
vxi11_put()</td><td>fbus_get()
fbus_put()</td><td colspan="6">ca_cntl()
ca_get()
ca_put()
ca_fna()</td></tr>
</table>

Utility Functions and Commands

All functions return a number or string value that may be used in an expression. The return values of some functions are only of use in conditional expressions, so their return values are given as **true** or **false**. The corresponding number values are 1 and 0, respectively.

System Functions

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

`chdir(`*`directory`*`)` — As above, but changes to the directory *`directory`*, which must be a string constant or expression.

`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 not set. Returns the integer exit status of the shell.

`unix(`*`command`*`)` — As above, but uses */bin/sh* to execute the one-line command *`command`*, which must be a string constant or expression. Returns the integer exit status of the command.

`unix(`*`command, str`* `[,` *`len`* `])` — As above, but the argument *`str`* is the name of a variable in which to place the string output from the command in the first argument. The maximum length of the string is 4096 bytes (including a null byte). The optional third argument can be used to specify a larger size.

`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 microsecond resolution as provided by the host *gettimeofday()* system call.

`date()` — Returns a string containing the current date as

```
Tue Feb  7 21:02:23 EST 2017
```

`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,

```
print date("%m-%d-%Y %T.%.6")
```

would display

```
02-07-2017 21:04:57.927905
```

`date(`*`seconds`* `[,` *`fmt`*`)` — As above, but the returned string represents the epoch given by *`seconds`*. See `time()` above.

`getenv(`*`string`*`)` — Returns the value of the environment variable represented by the string *`string`*. If the environment variable is unset, the null string is returned. Environment variables are exported to spec by the invoking shell program.

`file_info(`*`filename`* `[,` *`cmd`* `])` — Returns information on the file or device named *`filename`*. With a single *`filename`* argument, `file_info()` returns **true** if the file or device exists. If the argument *`filename`* is the string `"?"`, the possible values for *cmd* are listed. If *`filename`* is the string `"."`, spec uses the information from the last *stat()* system call made using the previous argument for *`filename`*, avoiding the overhead associated with an additional system call.

Possible values for *cmd* and the information returned follow. Note that the first set of commands parallel the contents of the data structure returned by the *stat()* system call, while the second set of commands mimic the arguments to the *test* utility available in the shell.

`"dev"` — The device number on which *`filename`* resides.

`"ino"` — The inode number of *`filename`*.

`"mode"` — A number coding the access modes and file attributes.

`"nlink"` — The number of hard links for *`filename`*.

`"uid"` — The user id of the owner.

`"gid"` — The group id of the owner.

`"rdev"` — The device ID if *`filename`* is a block or character device.

`"size"` — The size in bytes of *`filename`*.

`"atime"` — The time when *`filename`*'s data was last accessed.

`"mtime"` — The time when *`filenames`*'s data was last modified.

`"ctime"` — The time when *`filenames`*'s attributes were last modified.

`"isreg"` or `"-f"` — Returns **true** if *`filename`* is a regular file.

`"isdir"` or `"-d"` — Returns **true** if *`filename`* is a directory.

`"ischr"` or `"-c"` — Returns **true** if *`filename`* is a character device.

`"isblk"` or `"-b"` — Returns **true** if *`filename`* is a block device.

`"islnk"` or `"-h` or `"-L"` — Returns **true** if *`filename`* is a symbolic link.

`"isfifo"` or `"-p"` — Returns **true** if *`filename`* is a *named pipe* (sometimes called a *fifo*).

`"issock"` or `"-S"` – Returns **true** if *`filename`* is a socket.

`"-e"` – Returns **true** if *`filename`* exists.

`"-s"` – Returns **true** if the size of *`filename`* is greater than zero.

`"-r"` – Returns **true** if *`filename`* is readable.

`"-w"` – Returns **true** if *`filename`* is writable.

`"-x"` – Returns **true** if *`filename`* is executable.

`"-o"` – Returns **true** if *`filename`* is owned by you.

`"-G"` – Returns **true** if *`filename`* is owned by your group.

`"-u"` – Returns **true** if *`filename`* is setuid mode.

`"-g"` – Returns **true** if *`filename`* is setguid mode.

`"-k"` – Returns **true** if *`filename`* has its sticky bit set.

`"lines"` – Returns the number of newline characters in the file. If the file does not end with a newline, the count is increased by one.

`file_info(`*`pid`*`, "alive")` – Return **true** if the process associated with the process ID *`pid`* exists.

Miscellaneous

`lscmd` – This command lists all the built-in commands, functions and keywords.

`lscmd` *`pattern ...`* – As above, except only names matching *`pattern`* are listed.

`memstat` – Shows the current memory usage.

`eval(`*`string`*`)` – Parses and executes the string *`string`*, which can contain one or more statements or statement blocks. If the last statement in the string *`string`* is an expression, its value is returned. A single number, a quoted string or a single variable is considered an expression and its value will be returned. However, a single uninitialized variable will generate an error.

If the last statement is not an expression, `eval()` returns **true** (nonzero) if there were no errors executing the statement(s). The type of errors that normally cause spec to reset to command level (syntax errors, for example) will cause `eval()` to return **false** rather than reset to command level.

To distinguish between a **true** (1) or **false** (0) return value and an expression returning a value, `eval()` assigns a code to the built-in global variable EVAL_RESULT (as of spec release 6.04.05). If EVAL_RESULT is –1, there was an error parsing or executing the string, or an attempt was made to return the value of a single uninitialized variable. If EVAL_RESULT is 1, the string was a statement or was a statement list that ended in a statement. If EVAL_RESULT is zero, the string or last statement in the string was an expression.

An empty string or a string containing only white-space characters is a special case. EVAL_RESULT is set to 1 to indicate a statement, but the return value of eval() is zero.

If the string *string* contains multiple statements, the value of EVAL_RESULT will 0 or 1 based on the result of the last statement. If any of the statements generate an error, EVAL_RESULT will be set to –1 and the parsing and execution of the statements in the string *string* is abandoned.

If there is an error parsing or executing the string, in most cases, an error message will be assigned to the built-in global variable EVAL_ERR (as of spec release 6.04.05).

EVAL_RESULT and EVAL_ERR are reset to 0 and "" respectively on each call of eval(). With nested calls to eval(), EVAL_RESULT and EVAL_ERR will be associated with the most recently completed call.

Note, local symbols defined in the statement block in which eval() is used are not visible to the statements in the string *string*. Use eval2(), below, to access such symbols.

eval2(*string*) – Similar to eval(), but local symbols in the statement blocks surrounding the eval2() function call are visible and can be read or modified. To use local variables within the eval2() string that won't be associated with local variables of the same name outside the eval2() call, use curly brackets to enclose the statements with the local declaration within the string *string*.

sleep(*t*) – Suspends execution for a minimum of *t* seconds, where *t* may be non-integral. Actual sleep times may vary depending on other activity on the system and the resolution of the system clock. Returns **true**. Can be interrupted with **^C**.

gethelp(*topic* [, *flags*]) – Displays the help file *topic* on the screen. If *topic* contains a /, the argument is treated as an absolute or relative pathname. Otherwise, the argument refers to a file in the *spec_help* (or *help*) subdirectory of the SPECD directory. Returns non-zero if the file couldn't be opened.

The SPECD/*spec_help*/*.links* contains a list of alternative help files names and topics. If an entry matching *topic* is listed in *.links*, the corresponding file will be displayed.

Prior to spec release 6, spec contained an internal help file parser and formatter. With release 6, help files are formatted using the standard *groff* utility and paginated using the standard *less* utility.

If the argument *topic* is a pathname and the optional *flags* argument has bit 0x02 set, spec will use *groff* formatting macros appropriate for help files

written using pre-release 6 conventions.

Current spec help files are written in the format called *reStructuredText*. The *.rst* files are included in the *help* subdirectory of the spec distribution. The spec distribution also includes versions of the help files converted to *groff* format and versions preformatted for pagination for a screen width of 80 columns. It is these files that are copied to the directories `SPECD`/*spec_help*/*help_man* and `SPECD`/*spec_help*/*help_pre*, respectively. The preformatted files are displayed if the current screen width is between 80 and 92 columns or if the *groff* utility is unavailable. Otherwise the *groff* files are processed and displayed to fit the current screen width. The `gethelp()` function will also create commands to process *.rst* files and pass them through *groff* and the paginator, but only if *Python* and the *Python docutils* package are installed.

The paths and options for the system tools used by spec's help facility are configured in the file `SPECD`/*spec.conf*.

`whatis(`*`string`*`)` — Returns a number that encodes what the identifier in *`string`* represents. The return value 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.

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

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

The following two macros uses the `whatis()` function. The first determines whether or not a symbol has been given a value by looking at the *unset* bit. The second checks whether or not a symbol is the name of a macro.

```
10.FOURC> def unset(x) '{ return(whatis(x)&0x8000000? 1:0) }'

11.FOURC> def is_macro(x) '{ return(whatis(x)&2? 1:0) }'

12.FOURC>
```

`whatis(`*`string`*`, 1)` — With two arguments, returns a printable string explaining in words what kind of thing the thing named in *`string`* is to spec.

`spec_par(`*`par`* `[, `*`value`* `])` — Sets internal parameters. Typing `spec_par("?")` lists the available parameters, their current value and the default value, if different. The command `spec_par("set_defaults?")` sets all parameters to their default values. The currently available parameters are:

`"auto_file_close"` — The auto-file-close option is available to automatically close output files that haven't been accessed for some interval of time. The parameter units are hours, and the parameter can have nonintegral values. When the auto-close option is enabled, each time an `on()`, `off()`, `open()`, `close()` or `fprintf()` function is called, spec will check its list of opened output files. Any files which have not been written to for the length of time given by *`value`* hours will be closed. Enabling this option can help prevent errors when your macros or commands do not close files when appropriate, resulting in spec running out of resources to open additional files.

As files are opened automatically when sent output, auto-close mode operates transparently for the most part. However, if you change to a different working directory between the time the file is first opened and subsequently automatically closed, and if the file is not opened by an absolute path name, the next time you refer to the file, spec will reopen it using a relative path based on the current directory.

If *`value`* is zero, the mode is disabled. By default, the mode is initially disabled.

`"auto_hdw_poll"` — When automatic hardware polling is turned on (which it is, by default), spec will automatically poll busy motor controllers, timers and acquisition devices to determine when they are finished. For some devices, spec needs to perform an action, such as starting a motor backlash move, when the device is finished with its current business. Without automatic hardware polling, a call to the `wait()` function is necessary. A reason to turn it off may be to reduce the amount of debugging output during hardware debugging.

`"check_file_names"` — The check-file-names option can prevent you from (accidentally) creating files with names containing nonstandard characters. When enabled, if a file name passed to the `on()`, `open()` or `fprintf()`

functions contains any of the characters ()[]{}|$’’*?;!&<>", the space character, any control characters or any characters with the eighth bit set, and the file doesn't already exist, spec will print an error message and the function will fail. By default, this mode is on.

"confirm_quit" – If set, spec prompts with a "Really quit?" message when the quit or ^D commands are entered. The question must be answered in the affirmative to exit the program. The value for "confirm_quit" is not saved in the state file. The option must be set again on each spec invocation.

"elog_timestamp" – The time interval for the optional time stamps for the *elog* error file capability is set using this option. The units of the "elog_timestamp" parameter are minutes. The default value is five minutes. Note, time stamps are only added before a command or error message is logged, so that the interval between time stamps can be greater than that specified if no commands are being typed or errors generated.

"epics_timeout" – Sets the default timeout for channel access *ca_pend_io()* calls on EPICS. The default value is 0.5 seconds. This option appears only when spec is linked with the EPICS channel access libraries. Timeout values for individual process variables can still be changed with the epics_par() function. This parameter can be set in the *config* file.

"flush_interval" – The flush-interval parameter controls how often spec flushes output to the hard disk or other output device. Traditionally, spec flushed output after each print command. However, as some users report that this frequent flushing introduces considerable delays when the output device is to an NFS-mounted file system, the flushing interval can now be controlled. The "flush_interval" parameter specifies how many seconds to allow between output buffer flushing. The default value is 0.5 seconds. If the interval is set to zero, the traditional frequent-flushing behavior will be restored. Output to the screen is still flushed immediately. Output is also flushed each time the main spec prompt is issued.

"HKL_rounding" – Traditionally, spec rounded values for H, K, and L (and other geometry values derived from motor positions) to five significant digits for configurations using reciprocal space calculations. The rounding is off by default. It can be turned on using the command spec_par("HKL_rounding", 1e5) where the argument indicates the magnitude of the rounding, i.e., one part in 1e5, for example. Note, values with an absolute value less than 1e-10 are still rounded to zero whether or not the optional rounding is turned on.

"hdw_poll_interval" – When the wait() function is called to wait for polled motors, timers or other acquisition devices to finish, spec sleeps for a small interval between each check of the hardware. Use this spec_par() option to change that interval. The units of the parameter are milliseconds, and the default sleep time is 10msec. A value of zero is allowed, though not recommended if the computer is being used for anything else.

"history_join" – As of spec release 6.06.01, multi-line commands entered at the spec prompt are saved as one item in the history list. This option allows that mode to be disabled.

"history_size" – Configures the number of commands to be saved in the history list. The value can be any positive integer up to a set limit, currently 32768. Prior to spec release 6.05.01, the history size was fixed at 1000.

"keep_going" – Normally, when taking commands from a command file, spec resets to command level and the main interactive prompt when there are syntax errors in the file, certain floating point exceptions, references to unconfigured hardware, hardware access errors, along with a number of other errors. When the "keep_going" option is set, spec will keep reading and executing commands from a command file no matter what errors occur. When there is an error, the next line from the current command file will be read. Note, depending on where the error is in a file, reading subsequent lines may generate more errors, particularly if the error occurs inside a statement block.

"legacy_limit_check" – Prior to spec release 6.03.10, when a motor stopped, if either limit status was active, spec would report the motor hit a limit. Depending on how the controller reported the limit condition, this behavior could be an issue if the motor had executed a small move off an active limit, but hadn't moved enough to clear the limit switch. Even though the move was away from the limit, the limit status was still active, and spec would behave as if the move terminated due to a limit.

Release 6.03.10 improves the logic so that the direction of the move is checked against the limit status, and no action is taken if the active limit is not in the direction of the move. It turns out that if spec's code for a particular motor controller had the sense of the plus and minus limits reversed, the old code worked, but the new code does not. The error in limit sense was found and fixed with one controller model. If the issue is found with additional controllers, CSS will correct the code. In the meantime, the "legacy_limit_check" option can be set to a nonzero value to enable the pre 6.03.10 behavior.

Note, motors still stop when they hit a limit switch. The behavior that

is affected concerns whether spec notices, prints a message and perhaps aborts other moves or a scan.

`"modify_step_size"` – Normally, spec doesn't allow users to modify the motor step-size parameter with the `motor_par()` function, as the consequences are generally undesirable. However, in the rare circumstance that it is necessary, this parameter allows you to enable such modifications.

`"old_shared"` – With spec release 5.02.01, the structure of the shared array header was changed so that the data portion of the array would lie on a memory page boundary. To allow compatibility with applications built with the old header structure, the `"old_shared"` option can be set. However, this option can only be set as a `-o` command line start-up option, and the parameter is not saved in the state file. It must be set each time spec is invoked.

`"parse_units"` – When parsing of units is turned on, numbers typed as input to spec's parser with one of the following suffixes appended will automatically be multiplied by the corresponding factor.

1r	57.2958	radian
1mr	0.0572958	milliradian
1d	1	degree
1md	0.001	millidegree
1mm	1	millimeter
1um	0.001	micrometer
1m	0.0166667	minute
1s	0.000277778	second

Note, however, suffixes on numbers converted from strings or entered using the `getval()` function are not parsed. The only known use for unit-suffix parsing is with the user-contributed macros in the file *macros/units.mac*. These macros require that unit suffixes be supplied for all motor position arguments in the standard spec macros. The default is for this mode to be off.

`"show_prdef_files"` – When this mode is on, the source file for each macro definition is displayed with the `prdef` command. The default is for this mode to be on.

`"specwiz"` – Allows spec administrators to gain access to motors locked out in the *config* file to ordinary users. This feature allows the administrator to position the motors without having first to go into the configuration editor to change access modes. Entering `spec_par("specwiz", 1)` causes spec to prompt for the "Wizard's password". If entered correctly, the characters `_WIZ` are appended to the spec prompt to remind the

administrator of the special powers, and motors configured with protected status can be moved. Entering `spec_par("specwiz", 0)` disables the special mode.

spec looks for the encrypted password belonging to the spec_wiz (or specwiz) user in the files `SPECD`/*passwd*, /*etc*/*shadow*, and /*etc*/*passwd* in turn. If a shadow password file is used, ordinary users won't be able to read it, and the normal password file won't contain the encrypted password.

The spec distribution includes a *wiz_passwd* utility the can be used to create a *passwd* file in the spec auxiliary file directory that contains just the entry for the spec_wiz user. The `SPECD`/*passwd* file should probably be owned and writable only by root or the spec administrator.

Note, the standard macros `onwiz` and `offwiz` are convenient wrappers for the *specwiz* feature.

`"use_sem_undo"` — This flag relates to whether the `SEM_UNDO` flag is set when semaphores are used. It exists to get around a memory leak bug observed with some releases of the Solaris 2 operating system. The flag should be ignored unless you are instructed otherwise by CSS.

`"warn_not_at_pos"` — When enabled, spec prints a warning message whenever a motor doesn't reach its final position (as of release 5.08.02-8).

`calc(`*i*`)` — Calls a user-added function having code *i*. Codes are assigned in the distribution source file *u_hook.c*. Returns a user-supplied value or zero if there is no user-supplied value.

`calc(`*i*`, `*x*`)` — As above, but passes the argument *x* to the user-supplied function.

The geometry calculations that transform motor positions to reciprocal space coordinates and vice versa are implemented using calls to `calc()`. A description of the particular calls implemented for the four-circle diffractometer are in the *Four-Circle Reference*. See page 225 in the *Administrator's Guide* for information on how to include other user-added functions in the program.

`history` — This command lists the most recently entered commands. The maximum number of commands can be set using the `spec_par("history_size")` command (see page 85) and defaults to 1000. See the description of the command-recall (history) feature on page 56.

`history` *N* — As above, but only prints the *N* most recent commands. If *N* is negative, the commands are printed in reverse order.

`savstate` — Writes the current state to the state files.

The *state* file contains the variables, macro definitions, output file names and additional parameter values unique to each user, terminal and diffractometer. A separate file stores the user's command line history. The state files preserve the current situation when the user leaves the program, so that the same situation can be restored when the user later returns to the program.

spec can be invoked with a **–u** *user* flag and a **–t** *tty* flag. These flags instruct the program to initialize the current user's state from the files associated with the other user and/or terminal. Subsequent `savstate` commands access the user's natural state file.

The `savstate` command does not save the state files until the entire parsed mini-program in which the command occurs has been run. spec also automatically does a `savstate` before re-reading the hardware *config* file when the `reconfig` command is entered and when spec exits.

Note, a *show_state* utility is included in the spec distribution. This utility can display the contents of spec state files. Type `show_state -` to see the utility's usage message.

Keyboard and File Input, Screen and File Output

Controlling Output Files

With this group of functions, the names `"tty"` and `"/dev/tty"`, when used for *`filename`*, are special and refers to the user's terminal. The names `"null"` and `"/dev/null"` are also special and when used as as an output device, result in no output. The name `"pipe"` is also special, but only when spec is invoked with the **–p** flag, where it refers to the special data stream from spec to a front-end program.

`open()` — Lists all open files, including their directories, and indicates which files are currently turned on for output. Returns zero.

`open(`*`filename`*`)` — Makes *`filename`*, 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. If the `spec_par()` `"check_file_name"` option is on, and if *`filename`* contains any of the characters `()[]{}|$´`*?;!&<>"`, spec will print an error message and the function will return an error, unless the file already exists.

`close(`*`filename`*`)` — Closes *`filename`* and removes it from the table of files available for output. Returns zero for success, –1 if the file wasn't open. Files should be closed before modifying them with editors.

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

`on(`*`filename`*`)` — Turns on *`filename`* 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 *`filename`* 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(`*`filename`*`)` — Turns off output to *`filename`*, 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.

spec remembers the directory the files are in when they are first opened. If the user changes spec's current directory, open files may be referenced either by the name with which the files were opened or by the correct path name relative to the current directory. If an open file disappears from the file system (for example, if a user removes the file using a subshell), the next time the file is written to, spec prints a warning message and creates a new instance of the file.

Files should be closed before attempting to edit them outside of spec.

Errors during parsing or execution of commands, or typing a **`^C`** turns off all open files except *log* files (see next section).

Log Files (log, dlog, elog, tlog)

Four types of *log* files can be created for debugging or archiving purposes. The name or extension establishes the type of *log* file created. A regular *log* file is a file with a name beginning with the characters *log* or ending with the characters *.log*. All output sent to any device is sent to a *log* file.

A file beginning with *dlog* or ending with *.dlog* will collect all output sent to any device as does a regular *log* file, but debugging messages (generated when a value is assigned to the built-in variable `DEBUG`) are only written to such a file, not to the screen or any other file or device.

A file beginning with *elog* or ending with *.elog* records typed commands, error messages and optional time stamps. The file is intended to be useful to administrators trying to diagnose user problems. Commands entered at the spec prompt are logged prefixed by a `#C`. Error messages produced by the built-in C code or generated by the `eprint` or `eprintf()` built-in keywords, are logged prefixed by a `#E`. If time stamps are enabled (via the `spec_par()` `"elog_timestamp"` option), the UNIX epoch and the corresponding date string are logged (at the time-stamp interval) prefixed by a `#T`.

Finally, a file beginning with *tlog* or ending with *.tlog* collects output sent to the `"tty"` device (the screen). Unlike regular *log* files, output sent to other files or devices will not be saved to a *tlog* file.

For all types of log files, output is not turned off on errors or **^C** interrupts, and output generated by functions that "paint" the screen, such as `tty_fmt()`, `tty_move()`, `plot_move()`, `show_help()`, `spec_menu()` and `data_plot()`, isn't written to the files.

Reading From Files

The first function below is for reading strings from a file one line at a time. The second and third functions cause spec to switch its source of command input from the keyboard to the specified files.

`getline(`*`file`* `[,` *`arg`* `])` — This function reads successive lines from the ASCII file *`file`* each time it is called and returns the string so obtained, including the trailing newline. If *`arg`* is the string `"open"`, the function returns zero if the file can 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 filename argument changes.

To distinguish between an end of file, an error or a literal –1, `getline()` assigns a value to a built-in variable named `GETLINE_EOF` (as of spec release 6.03.05). The value of `GETLINE_EOF` will be 1 if there was an end-of-file condition on the read of the file, –1 if there was an error reading the file or if the file couldn't be opened and zero if the read was successful.

`dofile(`*`file`* `[,` *`line_num`*`|`*`search_pattern`* `])` — Queues the file *`file`* for reading commands. *`file`* must be a string constant or expression. Returns nonzero if the file doesn't exist or permit read access. An optional second argument can specify a line number or a text pattern that will be used to locate the point in the file to begin reading. If the argument 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 in the file.

`qdofile(`*`file`*`)` — As above, but does not show the contents of the file on the screen as the file is read.

Normally, most errors that occur while reading from a command file cause spec to reset to the level of the main interactive prompt with any open command files then closed. The `spec_par()` `"keep_going"` option is available to override that behavior. See `spec_par()` on page 85.

When a command file is opened within a statement block, the source of input isn't switched to the command file until all the commands in the statement block are executed. Thus it isn't possible to execute commands from a command file within a statement block. Note, though, the `getline()` (on page 90) function is available to scan strings from files.

When multiple command files are queued on a single command line, the input source can only change after the current line is exhausted, as the following example demonstrates:

```
12.FOURC> print "hi";dofile("file1");dofile("file2");print "bye"
hi
bye

FOURC.2> Reading "file2".
print "This is text from file2"
This is text from file2

FOURC.1> Reading "file1".
print "This is text from file1"
This is text from file1

13.FOURC>
```

Here *file1* contains the single line `print "This is text from file1"`, and *file2* contains `print "This is text from file2".`

Keyboard Input

`input()` — Reads a line of input from the keyboard. Leading white space and the trailing newline are removed and the string is returned. Returns the null string `""` if only white space was entered. Example:

```
13.FOURC> def change_it '{
14.quot>    local it
15.quot>    printf("Change it? ");
16.quot>    if ((it=input()) != "")
17.quot>          change_mac(it)
18.quot> }'

19.FOURC>
```

`input(`*`prompt`*`)` — As above, but prompts with the string *`prompt`*. Examples:

```
19.FOURC> input("Hit return when ready ... ")
Hit return when ready ... <return>

20.FOURC>
```

`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(`*`val`*`)` — Reads a line of input from the keyboard. The function returns 1 if the user answers with a string beginning with `Y`, `y` or `1`. The value of *`val`* is returned if the user simply enters return. Otherwise the function returns zero.

`yesno(`*`prompt, val`*`)` — As above, but prompts the user with the string *`prompt`*. The characters `" (YES)? "` are appended to the prompt string if *`val`* is nonzero. Otherwise the characters `" (NO)? "` are added.

`getval(`*`val`*`)` — Reads a line of input from the keyboard. If the user enters a value, that value is returned. The value of *`val`* is returned if the user simply enters return. The function works with both number and string values.

`getval(`*`prompt`*`, `*`val`*`)` — As above, but prompts the user with the string *`prompt`*. The string is printed followed by the current value of *`val`* in parenthesis, a question mark and a space. For example,

```
20.FOURC> DATAFILE = getval("Data file", DATAFILE)
Data file (pt100.133)?  <return>

21.FOURC>
```

`getsval(`*`prompt`*`, `*`val`*`)` — Like `getval()` above, prompts the user with the string *`prompt`*, if present, then waits for a user response. The value of *`val`* is returned if the user simply enters return. If the prompt string *`prompt`* is present, the string is printed followed by the current value of *`val`* 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.

Text Output

`print `*`a`*` [, `*`b`*` ... ]` — Prints the string value of each argument, adding a space between each string. If the argument is an associative array, each element of the array is printed in a list, as in:

```
21.FOURC> print mA
mA["0"] = 2
mA["1"] = 0
mA["2"] = 1
mA["3"] = 3

22.FOURC>
```

If the argument is a data array, the contents of the array are printed in a compressed format, as in:

```
22.FOURC> array data[64][64]; data[0] = 1; data[1] = 2
23.FOURC> print data
{{1 <64 repeats>}, {2 <64 repeats>}, {0 <64 repeats>} <62 repeats>}

24.FOURC>
```

`eprint `*`a`*` [, `*`b`*` ... ]` — As above, except that if an error-log file is open, the generated string will also be written to that file prefixed by the `#E` characters.

`printf(`*`format`*` [, `*`a`*` ... ])` — Does formatted printing on the turned-on output devices. *`format`* contains the format specifications for any following arguments. See the description of *printf()* in any C-language manual. Returns **true**.

`eprintf(`*`format`* `[, a ... ])` — As above, except that if an error-log file is open, the generated string will also be written to that file prefixed by the `#E` characters.

`fprintf(`*`file_name`*`, `*`format`* `[, a ... ])` — Does formatted printing on *`file_name`*. All other devices (except *log* files) are turned off while the string is printed. The specified file is opened, if necessary, and remains open until closed with the `close()` function.

`tty_cntl(`*`cmd`*`)` — 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 *`cmd`* 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 cursor up one line.
- `"do"` — Move cursor down one line.
- `"le"` — Move cursor left one space.
- `"nd"` — Move cursor right one space (nondestructive).
- `"resized?"` — A special option that 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?")`.

Returns **true** if *`cmd`* is recognized, otherwise returns **false**.

`tty_move(`*`x, y`* `[, `*`string`* `])` — Moves the cursor to column *`x`* and row *`y`* of the display, where column 0, row 0 is the upper left corner of the screen. If the third argument *`string`* is present, it is written as a label at the given position. The sequences and string are only written to the `"tty"` device and only if it is turned on for output. Special tty control sequences of the form `\[`*`xx`*`]`, where *`xx`* is one of the codes listed for the `tty_cntl()` function above, can be used

with *string*. Negative *x* or *y* position the cursor relative to the right or bottom edges of the screen, respectively. Relative moves are possible by adding ±1000 to *x* or *y* position arguments. Both coordinates must specify either relative or absolute moves. If one coordinate specifies a relative move, the absolute move in the other coordinate will be ignored. Please note, not all terminal types support relative moves. Returns **true**.

`tty_fmt(`*x*`, `*y*`, `*wid*`, `*string*`)` — Writes the string *string* 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 *string* is longer than the width given by *wid*, the string is split at space characters such that no line is longer then *wid*. Newlines in the string are retained, however. The function will truncate words that are wider than *wid* and drop lines that would go off the bottom of the screen. Special tty control sequences of the form `\[`*xx*`]`, where *xx* is one of the codes listed for the `tty_cntl()` function above, can be used with *string*. Negative *x* or *y* position the cursor relative to the right or bottom edges of the screen, respectively. The function returns the number of lines written.

`spec_menu(`*menu* `[, `*modes* `[, `*dumbterm*`]])` — Creates an interactive menu from specifications in the associative array *menu*. The menu can be used for configuring macros and other spec options. The standard spec macros `setplot`, `setshow`, `mstartup` and `plotselect` all use the `spec_menu()` function. When the menu is active, the user can use arrow and control keys to navigate the options and edit strings. While waiting for input in `spec_menu()`, spec will continue to poll hardware and respond to client commands if in server mode, (just as spec does while waiting for input at the main command prompt).

The *menu* argument is a two-dimensional associative array that completely describes the menu as detailed below. The first index of the array is a number that identifies an item. The second index of the array is a string that matches one of a set of keys that identify an attribute for the item. The value of the array element depends on the key.

The optional *modes* argument is a single value where each bit can be associated with a menu item. Up to 52 items may have associated bits.

If the number of menu items plus the number of lines needed to display the hints and *"info"* strings is greater than the number of rows in the terminal window, the menu items will be scrollable. The number of additional lines above or below the visible window will be displayed in the left of the screen. Navigating past the top or bottom will automatically scroll the menu.

If the width of the menu is greater than the width of the window, the menu items will be compressed or scrolled horizontally, to fit the available space.

If *dumbterm* is nonzero, spec_menu() will not use cursor positioning. Instead, the menu options will be displayed one at a time in sequence and the user will be prompted for a value. In this mode, entering a single - for any response will jump back and prompt again for the previous item in the sequence.

The function will return a possibly modified value for *modes*, reflecting the user's choices. If the menu is exited with **^C** or the **x** command, the values of the *menu* items on entry are restored to their starting values.

Each item selected by the first index into the *menu*[][] array can have various attributes configured via the following string keys placed in the second array index.

The "title", "head" and "subhead" items don't have any other modifiers. All other items require at least a "desc" key. An item with a "desc" key also needs at least one of "bit", "value", "svalue", "@" or "list". The other keys are optional.

"title" – The title for the menu. No further attributes apply to a "title" item. Only one title will be displayed, and it will be displayed at the top of the page.

"width" – The width of the menu. If not specified, the entire window will be used.

"head" – A section header for the menu. It will be preceded by a blank line.

"subhead" – A subsection header for the menu. It will not be preceded by a blank line.

"desc" – A required descriptive text string for each item. The string is displayed when prompting for the value. A null description creates a blank line.

"info" – Optional text that gives a description of the parameter. The "info" text is displayed only when the current item is active. The text will be formatted for width using the rules of the tty_fmt() function. Formatting will work best if there are no embedded newlines in the string.

The value of a menu item is modifiable if at least one of the following five keys is included in the item description.

"bit" – Associates a bit in the *modes* argument to this item. The element value should be a single bit and each item in the array that has a bit set should have a unique value. The associated bit will be set or cleared in the return value of spec_menu(). Also, a "bit" attribute can be associated with items that have a "value", "svalue", "toggle" or "list" attribute so that such items can be enabled or disabled for editing using the "bit_and", "bit_or" and "bit_not" attributes. Currently, bits 0 through 51 can be used.

"toggle" — Similar to the "bit" key in that the value can be toggled on or off, but there is no limit to the number of such items and no connection to the *modes* argument. The item's value contains the initial state on entry and the selected state when spec_menu() returns.

"choices" — For "bit" and "toggle" items, provides prompt strings to replace the default YES and NO. The choices are provided in one colon-delimited string, for example, "ENABLE:DISABLE" or "ON:OFF". The choices don't need to be capitalized and can include spaces. The first choice is associated with the value of one and the second with zero.

"value" — Indicates a number value is wanted. Also used to hold the "list" item selection. This item's value will be modified. Not used if the "@" indirection key is present. Note, the number value can be entered as an expression which will be evaluated with the result displayed.

"svalue" — Indicates a string value is wanted. Can also be used to select the current "list" item. This item's value will be modified. Not used if the "@" indirection key is present.

"list" — Presents a list of possible values, only one of which can be selected. The selection is set and returned in "value", "svalue" or indirectly via the "@" key. Use "value" to indicate selection based on position in the list, counting from one. Use "svalue" to indicate selection based on a matching strings. If "@" is used, spec will use a number or string as appropriate. By default, one or more space characters separate the list items. An alternate delimiter character or string can be specified using the "delim" attribute.

"@" — The value of this element contains the string name of a spec variable that is to be modified. If its current value is a number, then the rules for entering number values will be followed. If its current value is a string or the parameter is unset, the string rules will be followed. This key takes precedence over "value" or "svalue" attributes. Names of scalars and associative array elements are allowed, but data array elements are not. If the named variable doesn't exist, spec_menu() will display the variable name and "not found" as the value.

"format" — Can be used to set an alternative *printf()* format for display of a number-valued item. For example, "0x%X" can be used to display a value in hexadecimal. In fact, the format can be used to specify the same number value in multiple formats, such as "hex=0x%x dec=%d". The default format is "%.9g".

"delim" — Provides an alternative delimiter for "list" items. Can be one or more characters. The default delimiter is a space character. When the delimiter is a space character, multiple space and tab characters count as one delimiter. When the delimiter is any other character or string,

each instance is a delimiter.

`"default"` — For `"list"` items, sets the default choice if the value passed via `"value"`, `"svalue"` or `"@"` is invalid or out of range.

`"min"` — For `"value"` items, a minimum value to allow.

`"max"` — For `"value"` items, a maximum value to allow.

The item types `"bit"`, `"value"`, `"svalue"`, `"toggle"` and `"list"` can have the following attributes set to enable or disable the item for editing based on the bit values of other items in the current value of modes.

`"bit_and"` — Editable if all the matching bits (or bit) are set in *`modes`*.

`"bit_or"` — Editable if any of the matching bits (or bit) are set in *`modes`*.

`"bit_not"` — Editable if the matching bits (or bit) are not set in *`modes`*. Finally, this key applies to bit-toggle items:

`"bit_flip"` — Flip the YES/NO sense of this item. Normally, when a bit is set in *`modes`*, the value is presented as `"YES"`. If `"bit_flip"` is present, the logic is reversed. This option can be used to avoid double negatives in the query/response, particularly when one doesn't have the freedom to define what a set bit means. For example, `"Draw error bars (YES)?"` is preferable to `"Don't draw error bars (NO)"`.

On a normal return with the `q` or `^D` keys, `spec_menu()` will return an updated value for *`modes`* that reflects the new values of all the `"bit"` items. In addition, new values for `"value"`, `"svalue"` and variables passed indirectly using `"@"` will be assigned. Finally, for any items that have been modified an element will be added to the `*menu*` array with a second index named `"updated"` and a value set to one.

The menu is presented as a list with the cursor positioned at the current value for a modifiable item. The up- and down-arrow keys, the `^P` (previous) and `^N` (next) control keys and the return key accept the current choice and move up or down the list.

One exits the menu saving the modifications using the `q` (quit) key or `^D`. One can abandon the menu without saving the modifications using the `x` (exit) key or **`^C`**.

The display is refreshed with `^L`, taking into account the current window size.

Bit-value and toggle items are toggled using the space bar or the left- or right-arrow keys. In addition, the `y`, `Y` and `1` keys select YES, while the `n`, `N` and `0` keys select NO.

List items are also navigated using the left- and right-arrow keys and the `^B` (back) and `^F` (forward) to move one position. The `^A` and `^E` keys move to the first item and last item, respectively. Also, typing the first character of a list

item will move the cursor to the first item in the list that begins with that character. The space bar makes the currently highlighted item the list selection.

Text entry items (both string- and number-valued) allow insertion and deletion of text from any point in the string. The left- and right-arrow keys and the `^B` (back) and `^F` (forward) move one position. The `^A` and `^E` keys move the cursor to the start and end of the entry, respectively. The `^D` key and the keyboard forward-delete key both delete forward. The backspace and delete key delete backward. The `^U` and `^K` keys delete from the current position to the beginning and end of line, respectively.

For number-valued items, the text entered will be evaluated. Thus, expressions are allowed. For example, `PI/2`, `2+2` or `pow(2,12)` are all valid entries. Only commands and functions that make sense in the context of generating an expression are allowed. Other commands will not be executed and will generate an error message that will be displayed near the bottom of the menu window. Among the commands not allowed are those that generate screen output or control hardware.

To enter a string that starts with one of the navigation command letters, namely `q` or `x`, use one of the text-editing keys, such as left- or right-arrow to switch to text-entry mode.

When items are disabled by way of the `"bit_and"`, `"bit_or"` and `"bit_not"` logic, the values appear as `---`, and the up/down navigation passes over those items.

The standard spec macros `setplot`, `setshow`, `mstartup` and `plotselect` use the `spec_menu()` function.

Here is another example:

```
def menu1 '{
    local i, menu[], modes
    local   group_size

    modes = 1
    group_size = 512
    menu[++i]["title"] = "Menu Example"
    menu[++i]["desc"] = "Enable List Examples"
    menu[  i]["bit"] = 0x0001

    menu[++i]["desc"] = "  List Example 1"
    menu[  i]["bit_and"] = 0x0001
    menu[  i]["list"] = "Now is the time for all good men"
    menu[  i]["svalue"] = "time"
```

```
        menu[++i]["desc"] = "  List Example 2"
        menu[  i]["bit_and"] = 0x0001
        menu[  i]["list"] = "The quick::brown fox::jumps over::the lazy dog"
        menu[  i]["delim"] = "::"
        menu[  i]["value"] = 3

        menu[++i]["desc"] = "  List Example 3"
        menu[  i]["bit_and"] = 0x0001
        menu[  i]["list"] = "128 256 512 1024 2048 4096 8192"
        menu[  i][""""] = "group_size"

        menu[++i]["desc"] = ""
        menu[++i]["desc"] = "Use logarithmic y-axis"
        menu[  i]["bit"]     = PL_YLOG
        menu[++i]["desc"] = "Force y-axis minimum to zero"
        menu[  i]["bit"]     = PL_YZERO
        menu[  i]["bit_not"] = PL_YLOG

        modes = spec_menu(menu, modes)

        print modes, group_size
    '
```

Variables

`global` *`name ...`* — Declares *name* to be a global symbol. A global symbol retains its value after each parsed program is executed. If *name* is used as an array name, each element of the array is global. By appending empty square brackets to *name* the type of the symbol can be forced to be an associative array, which may be useful if *name* is to be used as an argument to a macro function before its type has been established by usage.

`unglobal` *`name ...`* — Makes the global or constant symbol *name* no longer global.

`constant` *`name`* `[=]` *`expression`* — Declares *name* to be a constant, global symbol having the value given by *`expression`*. A constant symbol cannot be changed by assignment.

`local` *`name ...`* — Allows reuse of a preexisting name and gives the new instance of that name scope only within the statement block in which it is defined. The name may be that of a macro, in which case the macro definition is unavailable within the statement block. By appending empty square brackets to *name* the type of the symbol can be forced to be an associative array, which may be useful if *name* is to be used as an argument to a macro function before its type has been established by usage.

`delete` *`assoc-array`*`[`*`elem`*`] ...` — Removes the element *`elem`* of the associative array *`assoc-array`*.

`syms [-v] [+|-BGLADNSIC] [`*`pattern ...`*`]` — Lists spec's current variables. Without arguments, all the variables are listed, along with their memory consumption and type. With the `-v` flag, the variables are listed along with their values in a format that can be saved to a file and read back as commands. If arguments are given as *`pattern`*, only symbols matching the arguments are printed. Such arguments may contain the `?` and `*` metacharacters.

In addition, the type of symbols listed can be controlled using the flags in the following table where a `-` flag prevents symbols with the given attribute from being listed and a `+` flag includes symbols with the given attribute in the list.

`B`	Built-In
`G`	Global
`L`	Local
`A`	Associative array
`D`	Data array
`N`	Number type
`S`	String type
`I`	Immutable attribute
`C`	Constant attribute

`[[extern] shared] [`*`type`*`] array `*`var`*`[`*`cols`*`]` — Declares a one-dimensional data array.

`[[extern] shared] [`*`type`*`] array `*`var`*`[`*`rows`*`][`*`cols`*`]` — Declares a two-dimensional data array.

Macros

Built-In Commands

`def `*`name string`* — Defines a macro named *`name`* to be *`string`*. Each time *`name`* occurs on input, it is replaced with *`string`*. The definition is made immediately, so the macro can be used later in the same statement block in which it is defined and can be redefined within the same statement block.

Note that the macro definition is made regardless of any surrounding flow control statements, since the enclosing mini-program is not yet completely parsed and is not executing.

`rdef` *`name expression`* — Defines a macro named *`name`* to be *`expression`*, which is almost always a string constant. Each time *`name`* occurs on input, the value *`expression`* is substituted. Unlike `def`, described above, the macro definition is not made until all the encompassing statement blocks are parsed and the resulting mini-program is executed. Consider the following example.

```
if (flag == 1)
        rdef plot "onp;offt;lp_plot;ont;plot_res;offp"
else if (flag == 2)
        rdef plot "splot;onp;offt;lp_plot;plot_res;ont;offp"
else if (flag == 3)
        rdef plot "onp;plot_res;offp"
else
        rdef plot ""
```

Clearly, it is necessary for the mini-program to be parsed and executed to decide which is the appropriate definition to assign to the `plot` macro.

`prdef` — Displays all macro definitions. The displayed definitions are prepended with `def name ´` and terminated with `´` so if saved to a file, the definitions can be read back. (See the standard macro `savmac` on page 162.)

`prdef` *`pattern ...`* — As above, except only macro names matching *`pattern`* are listed, where *`pattern`* may contain the metacharacters `?` or `*`, which have the usual meaning: `?` matches any single character and `*` matches any string.

`lsdef` — Lists the name and the number of characters in each macro definition.

`lsdef` *`pattern ...`* — As above, except only macro names matching *`pattern`* are listed, where *`pattern`* may contain the metacharacters `?` or `*`.

`undef` *`name ...`* — Removes the named macros, which can be ordinary macros, macro functions or `cdef()` macros.

`cdef("`*`name`*`",` *`string`* `[, "`*`key`*`" [,` *`flags`* `]])` — Defines parts of *chained* macros. A chained macro definition is maintained in pieces that can be selectively included to form the complete macro definition. The argument *`name`* is the name of the macro. The argument *`string`* contains a piece to add to the macro.

The chained macro can have three parts: a beginning, a middle and an end. Pieces included in each of the parts of the macros are sorted lexicographically by the keys when putting together the macro definition. 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.

The *`key`* argument allows a piece to be selectively replaced or deleted, and also controls the order in which the piece is placed into the macro definition. The *`flags`* argument controls whether the pieces are added to the beginning or to the end of the macro, and also whether the pieces should be selectively

included in the definition depending on whether *key* is the mnemonic of a configured motor or counter.

The bit meanings for *flags* are as follows:

0x01 – only include if key is a motor mnemonic
0x02 – only include if key is a counter mnemonic
0x10 – place in the beginning part of the macro
0x20 – place in the end part of the macro

If *flag* is the string "delete", the piece associated with *key* is deleted from the named macro, or if name is the null string, from all the chained macros. If *key* is the null string, the *flags* have no effect.

If *flags* is the string "enable", the parts of the named macro associated with *key* are enabled, and if *flags* 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. However, the commands lsdef, prdef and undef will function with chained macros.

When spec starts and when the reconfig command is run (or the config macro is invoked), all the chained macros are adjusted for the currently configured motors and counters.

cdef("?") – Lists all the pieces of all the chained macros.

cdef("*name*", "", "?") – Lists the pieces of the macro named *name*.

clone(*destination, source*) – Duplicates the macro *source* as a new macro named *destination*. A clone of a chained macro becomes an ordinary macro.

strdef("*name*" [, *arr*]) – Returns a string containing the macro definition of *name*. If *name* is not a defined macro, returns the string *name* itself. If an associative array *arr* is included as an argument and if the macro *name* is a macro function, elements of *arr* indexed starting at 0 will be assigned the string names of the arguments to the macro function. The element *arr*["file"] will be assigned the name of file where the macro was defined or "tty" if the macro was defined at the keyboard.

strdef("*name*", *key* [, *arr*]) – If *name* is a chained macro, returns a string containing only the definition segment associated with *key*. If *name* is not a defined macro, returns the string *name* itself. If *name* is a macro, but not a chained macro, returns the definition. If *name* is a chained macro, but doesn't contain a

segment associated with *key*, returns the null string. If an associative array *arr* is included as an argument and if the macro *name* is a macro function, elements of *arr* indexed starting at 0 will be assigned the string names of the arguments to the macro function. The element *arr*`["file"]` will be assigned the name of file where the macro was defined or `"tty"` if the macro was defined at the keyboard.

Built-In Macro Names

The following macro names are built-in to spec. They are run at the specified times only if they have been given a definition.

`begin_mac` — If a macro by this name exists, it will be run after reading the hardware configuration file and all the start-up command files, but before reading commands from the keyboard.

`end_mac` — If a macro by this name exists, it will be run when spec exits from either a **`^D`** or a `quit` command.

`config_mac` — If a macro by this name exists, it will be run after reading the configuration file at start up and after the `reconfig` command is executed.

`prompt_mac` — If a macro by this name exists, it will always be run just before spec issues the main, level-zero prompt. If an error occurs while running `prompt_mac`, it will be automatically undefined.

`cleanup, cleanup1` — If either or both exists, they will be run whenever an error is encountered, the `exit` command is encountered, or a user types **`^C`**. The `cleanup` macro is run first. After running the clean-up macros, spec gives the standard prompt and waits for the next command from the keyboard.

`cleanup_once` — A clean-up macro that is always deleted before a new spec main prompt is issued. If defined, its definition will be pushed on to the input stream whenever an error is encountered, the `exit` command is encountered, or a user types **`^C`**.

`cleanup_always` — Like `cleanup_once`, but its definition is not removed except by an explicit `undef` command.

The `cleanup` and `cleanup1` macros are no longer used in the standard macros. The more recent `cleanup_once` and `cleanup_always` macros are preferred.

Macro definitions for these built-in macros should be maintained using `cdef()` so that independent macro packages can make use of the macros without interference.

Macro Arguments

Within ordinary macros (not macro functions), the symbols `$1`, `$2`, `...`, are replaced by the arguments with which the macro is invoked. Arguments are defined as strings of characters separated by spaces. Also,

`$0` is replaced with the macro name,
`$*` is replaced with all the arguments,
`$@` is replaced with arguments delimited by `\a`,
`$#` is replaced with the number of arguments,
`$$` is a literal `$`.

A macro argument is a string of characters delimited by spaces. Use quotes to include spaces within a single argument. Use `\"` or `\'` to pass literal quotes. Arguments can be continued over more than one line by putting a backslash at the end of the line.

When a macro defined without arguments is invoked, only the macro name is replaced with the definition.

When a macro defined with arguments is invoked, all characters on the input line up to a `;`, a `{` or the end of the line are eaten up, whether or not the macro uses them as arguments.

When numbered arguments are referred to in the macro definition, but are missing when the macro is invoked, they are replaced with zeros. If `$*` is used in the definition and there are no arguments, no characters are substituted.

Argument substitution occurs very early in the input process, before the substituted text is sent to the parser. It is not possible to use variables or expressions to specify the argument number.

It is often useful when parsing macro arguments, particularly when the macro is called with a variable number of arguments, to use the `split()` function to place the arguments into an associative array. Typical syntax is:

```
{
   local ac, av[]
   ac = split("$*", av)
}
```

Note, that usage does not respect quoted arguments, since `$*` removes quotation marks when concatenating the macro arguments.

The sequence `$@` is replaced with the concatenated arguments delimited by the special character `\a` (the audible bell, **^G**, ASCII 7). The string can then be split as follows:

```
{
   local ac, av[]
   ac = split("$@", av, "\a")
}
```

The elements of `av[]` will respect the quoted arguments in the macro invocation. There is no syntax to escape the `\a`.

There are no limits on the length of macro definitions, the number of macro arguments or on the total combined size of all macro definitions.

Beware of unwanted side affects when referencing the same argument more than once. For example,

```
def test 'a = $1; b = 2 * $1'
```

invoked as **`test i++`**, would be replaced with `a = i++; b = 2 * i++`, with the result that `i` is incremented twice, even though that action is not apparent to the user. The previous definition also would cause problems if invoked as `test 2+3`, as that would be replaced with `a = 2+3; b = 2 * 2+3`. The latter expression evaluates to 7, not 10, as might have been intended by the user. Use of parenthesis to surround arguments used in arithmetic expressions in macro definitions will avoid such problems, as in `b = 2 * ($1)`.

Macro Functions

Macro functions are a type of macro that can return values and can be used as an operand in expressions. The macro definition can include function arguments, which then become available to the body of the macro function. For example,

```
def factorial(n) '{
        if (n <= 1)
                return(1);
        return(n * factorial(n-1))
}'
```

The syntax of macro functions requires the macro name followed by a set of parenthesis which can contain a comma-separated list of argument names. The arguments names become local variables within the macro definition. The definition must be a statement block, that is, the statements must be enclosed in curly brackets.

String and Number Functions

Math Functions

`exp(x), exp10(x)` — Returns e^x and 10^x respectively.

`log(x), log10(x)` — Returns the natural logarithm and the base 10 logarithm of x respectively.

`sqrt(x)` — Returns the square root of x.

`pow(x, y)` — Returns x^y.

`fabs(x)` — Returns the absolute value of x.

`int(x)` — Returns the integer part of x. The integer part is formed by truncation towards zero.

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

`rand(x)` — If x is positive, returns a random integer between 0 and x, inclusive. If x is negative, returns a random integer between $-x$ and x, inclusive. Values of x greater than 32767 or less than −16383 are set to those limits. If x 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 number generator used by the `rand()` function to the integer value *seed*. This function allows the same sequence of random numbers to be generated reproducibly by resetting the seed to the same value.

`sin(x), cos(x), tan(x)` — Returns the sine, cosine and tangent, respectively, of the argument x, which must be in radians.

`asin(x), acos(x), atan(x)` — Returns the arc sine, arc cosine and arc tangent, respectively, of the argument x. The return value is in radians. `asin()` and `atan()` return values in the range $-\pi/2$ to $\pi/2$, while `acos()` returns values from 0 to π.

`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 π. Having both y and x zero is an error.

String Functions

`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(string, arr)` — Splits the string *string* at space characters and assigns the resulting substrings to successive elements of the associative array *arr*, starting with element 0. The space characters are eliminated. The functions returns the number of elements assigned.

`split(string, arr, delimiter)` — As above, but splits the string into elements that are delimited by the string *delimiter*. The delimiting characters are eliminated.

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

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

`length(string)` — Returns the length of the string *string*.

`sprintf(format [, a, b, ... ])` — Returns a string containing the formatted print. See *sprintf()* in a C-language reference manual.

`sscanf(string, format, a [, b, ... ])` — Scans the literal string or string variable *string* for data, where *format* contains a format specification in the same style as the C language *scanf()* function. Each subsequent argument 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.

Regular Expression Functions

Regular expressions are sequences of special characters for searching for patterns in strings. spec implements extended regular expression using the C library *regcomp()* and *regexec()* functions, which have a somewhat platform-dependent implementation. See the regular expression man page (**`man 7 regex`** on *Linux* and **`man re_format`** on OS X) for details of regular expression syntax. The names and usage of the following spec functions resemble those used in the UNIX *awk* (or *gawk*) utility. (These functions added in spec release 6.03.04.)

`rsplit(str, arr, regex)` — Similar to `split()` above, but the optional delimiter argument can be a regular expression. The string *str* is split into elements that are delimited by the regular expression *regex* and the resulting substrings are assigned to successive elements of the array *arr*, starting with element 0. The delimiting characters are eliminated. Returns the number of elements assigned.

`sub(regex, sub, str)` — Replaces the first instance of the regular expression *regex* in the source string *str* with the substitute string *sub*. An `&` in the substitute string is replaced with the text that was matched by the regular expression. A `\&` (which must be typed as `"\\&"`) will produce a literal `&`. Returns the

modified string.

`gsub(`*`regex, sub, str`*`)` — Replaces all instances of the regular expression *`regex`* in the source string *`str`* with the substitute string *`sub`*. An `&` in the substitute string is replaced with the text that was matched by the regular expression. A `\&` (which must be typed as `"\\&"`) will produce a literal `&`. Returns the modified string.

`gensub(`*`regex`*`, `*`sub`*`, `*`which`*`, `*`str`*`)` — Replaces instances of the regular expression *`regex`* in the source string *`str`* with the substitute string *`sub`* based on the value of *`which`*. If *`which`* is a string beginning with `G` or `g` (for global), all instances that match are replaced. Otherwise, *`which`* is a positive integer that indicates which match to replace. For example, a `2` means replace the second match.

In addition, the substitute text may contain the sequences `\`*`N`* (which must be typed as `"\\`*`N`*`"`), where *`N`* is a digit from 0 to 9. That sequence will be replaced with the text that matches the *`N`*th parenthesized subexpression in *`regex`*. A `\0` is replaced with the text that matches the entire regular expression. Returns the modified string.

`match(`*`str`*`, `*`regex`*` [, `*`arr`*`])` — Returns the position in the source string *`str`* that matches the regular expression *`regex`*. The first position is 1. Returns 0 if there is no match or −1 if the regular expression is invalid. If the associative array *`arr`* is provided, its contents are cleared and new elements are assigned based on the consecutive matching parenthesized subexpressions in *`regex`*. The zeroth element, *`arr`*`[0]`, is assigned the entire matching text, while *`arr`*`[0]["start"]` is assigned the starting position of the match and *`arr`*`[0]["length"]` is assigned the length of the match. Elements from 1 onward are assigned matches, positions and lengths of the corresponding matching parenthesized subexpressions in *`regex`*.

Conversion Functions

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

`bcd(`*`x`*`)` — Returns a 24-bit integer that is the binary-coded decimal representation of the nonnegative integer *x*.

`dcb(`*`x`*`)` — Returns the nonnegative integer corresponding to the 24-bit binary-coded decimal representation *x*.

`deg(x)` — Returns the argument converted from radians to degrees.

`rad(x)` — Returns the argument converted from degrees to radians.

Data Handling and Plotting Functions

`array_op(cmd, arr [, args ...])` — Performs operations on the array based on the following values for *cmd*:

`"fill"` — Fills the array *arr* with values. For a two-dimensional array,

```
array_op("fill", arr, u, v)
```

produces for each element

```
arr[i][j] = u × i +  v × j
```

With subarrays, *i* and *j* refer to the subarray index. Also, *i* and *j* always increase, even for reversed subarrays, so

```
array_op("fill", arr[-1:0][-1:0], 1, 1)
```

fills *arr* in reverse order.

`"contract"` — For *args u* and *v*, returns a new array with dimensions contracted by a factor of *u* in rows and *v* in columns. Elements of the new array are formed by averaging every *u* elements of each row with every *v* elements of each column. If there are leftover rows or columns, they are averaged also.

`"min"` or `"gmin"` — Returns the minimum value contained in the array.

`"max"` or `"gmax"` — Returns the maximum value contained in the array.

`"i_at_min"` or `"i_at_gmin"` — Returns the *index* number of the minimum value of the array. For a two-dimensional array dimensioned as `D[N][M]`, the index number of element `D[i][j]` is `(i × M) + j`. If *arr* is a subarray, the index is with respect to the full array, although the minimum is the minimum value in the specified subarray.

`"i_at_max"` or `"i_at_gmax"` — Returns the index number of the maximum value of the array. See `"i_at_min"` for subarray considerations.

`"row_at_min"` or `"rmin"` — Returns the row number containing the minimum value of the array. If *arr* is a subarray, the row is with respect to the full array, although the minimum is the minimum value in the specified subarray.

`"row_at_max"` or `"rmax"` — Returns the row number containing the maximum value of the array. If *arr* is a subarray, the row is with respect to the full array, although the maximum is the maximum value in the specified subarray.

`"col_at_min"` or `"cmin"` – Returns the column number containing the minimum value of the array. If *arr* is a subarray, the column is with respect to the full array, although the minimum is the minimum value in the specified subarray.

`"col_at_max"` or `"cmax"` – Returns the column number containing the maximum value of the array. If *arr* is a subarray, the column is with respect to the full array, although the maximum is the maximum value in the specified subarray.

`"i_<=_value"` – Returns the *index* number of the nearest element of the array with a value at or less than *u*. For a two-dimensional array dimensioned as `D[N][M]`, the index number of element `D[i][j]` is `(i` × `M) + j`. Unlike `"i_at_min"`, `"i_at_max"`, etc., if *arr* is a subarray, the index is with respect to the subarray.

`"i_>=_value"` – Returns the *index* number of the nearest element of the array with a value at or greater than *u*, starting from the last element. For a two-dimensional array dimensioned as `D[N][M]`, the index number of element `D[i][j]` is `(i` × `M) + j`. Unlike `"i_at_min"`, `"i_at_max"`, etc., if *arr* is a subarray, the index is with respect to the subarray.

`"fwhm"` – Requires two array arguments, each representing a single row or single column. Returns the full-width in the first array at half the maximum value of the second array.

`"cfwhm"` – Requires two array arguments, each representing a single row or single column. Returns the center of the full-width in the first array at half the maximum value of the second array.

`"uhmx"` – Requires two array arguments, each representing a single row or single column. Returns the value in the first array corresponding to half the maximum value in the second array and at a higher index.

`"lhmx"` – Requires two array arguments, each representing a single row or single column. Returns the value in the first array corresponding to half the maximum value in the second array and at a lower index.

`"com"` – Requires two array arguments, each representing a single row or single column. Returns the center of mass in the first array with respect to the second array. The value is the sum of the products of each element of the first array and the corresponding element of the second array, divided by the number of points.

`"x_at_min"` – Requires two array arguments, each representing a single row or single column. Returns the element in the first array that corresponds to the minimum value in the second array.

`"x_at_max"` – Requires two array arguments, each representing a single row or single column. Returns the element in the first array that corresponds to the maximum value in the second array.

`"sum"` or `"gsum"` — Returns the sum of the elements of the array. If there is a third argument greater than zero, the array is considered as a sequence of frames, with the third argument the number of rows in each frame. The return value is a new array with that number of rows and the same number of columns as the original array. Each element of the returned array is the sum of the corresponding elements of each frame. For example, if the original array is dimensioned as `data[N][M]`, the return value for

```
arr = array_op("sum", data, R)
```

is a new array of dimension `arr[N/R][M]`, where each element `arr[i][j]` is the sum of `k` from `0` to `R - 1` of `data[i + k × N / R][j]`.

`"sumsq"` — Returns the sum of the squares of the elements of the array. If there is a third argument and it is greater than zero, the interpretation is the same as above for `"sum"`, except the elements in the returned array are sums of squares of the elements in the original array.

`"transpose"` — Returns a new array of the same type with the rows and columns switched.

`"updated?"` — Returns nonzero if the data in the array has been accessed for writing since the last check, otherwise returns zero.

`"rows"` — Returns the number of rows in the array.

`"cols"` — Returns the number of columns in the array.

`"row_wise"` — With a nonzero third argument, forces the `array_dump()`, `array_fit()`, `array_pipe()`, `array_plot()` and `array_read()` functions to treat the array as row-wise, meaning each row corresponds to a data point. With only two arguments, returns nonzero if the array is already set to row-wise mode.

`"col_wise"` — As above, but sets or indicates the column-wise sense of the array.

`"sort"` — Returns an ascending sort of the array.

`"swap"` — Swaps the bytes of the named array. The command can change big-endian short- or long-integer data to little-endian and vice versa. For most built-in data collection, spec automatically swaps bytes as appropriate, but this function is available for other cases that may come up.

`"frame_size"` — The number of rows in a frame. The frame size is part of the shared array header and may be useful to auxiliary programs, although the value is maintained for non-shared arrays. Note, setting the frame size to zero will clear the `"frames"` tag. Setting the frame size to a non-zero value will set the `"frames"` tag.

`"latest_frame"` — The most recently updated frame. The latest frame is part of the shared array header and may be useful to auxiliary programs,

although the value is maintained for non-shared arrays.

"tag" – Shared arrays can be tagged with a type that will be available to other processes accessing the array. Usage is array_op("tag", *arr*, *arg*) where *arr* is the array and *arg* is "mca", "image", "frames", "scan" or "info".

"untag" – Removes tag information.

"info" – Returns or sets the *info* field of a shared array segment. The field can be contain up to 512 bytes of arbitrary text. When setting the field, if the string argument is longer than 512 bytes, the first 512 bytes will be copied. The function returns the number of bytes copied, –1 if *arr* is not a shared array or 0 if *arr* is a shared array that doesn't support the *info* field. The *info* field is included in SHM_VERSION version 6 headers, added in spec release 6.00.08.

"meta" – Returns or sets the *meta* area of a shared array segment. With spec, the field can contain up to 8,192 bytes of arbitrary text. When setting the field, if the string argument is longer than 8,192 bytes, the first 8,192 bytes will be copied. The function returns the number of bytes copied, –1 if *arr* is not a shared array or 0 if *arr* is a shared array that doesn't support the *meta* field. The *meta* field is included in SHM_VERSION version 6 headers, added in spec release 6.00.08.

array_dump([*file*,] *arr* [, *arr2* ...][, *options* ...]) – Efficiently writes the data in the array *arr* and optionally arrays *arr2*, ..., etc. If the initial optional *file* argument is given, the output is to the named file or device. Otherwise, output is to all "on" output devices, most notably the screen. The additional optional *options* arguments are strings that control the formatting.

A format argument can specify a *printf()*-style format for the values. The default format is "%.9g", which prints nine digits of precision using fixed point or exponential format, whichever is more appropriate to the value's magnitude. Recognized format characters are e or E (exponential), f (fixed point), g or G (fixed or exponential based on magnitude), d (decimal integer), u (unsigned integer), o (octal integer), x or X (hexadecimal integer). All formats accept standard options such as precision and field width. For example, "%15.8f" uses fixed-point format with eight digits after the decimal point and a fifteen-character-wide field. For the integer formats, double values will be converted to integers. Also, initial characters can be included in the format string, for example, "0x%08x" is valid.

The option "%D=*c*", specifies an alternate delimiter character *c* to replace the default space character delimiter that is placed between each element in a row of output. For example, one might use a comma, a colon or the tab character with "%D=,", "%D=:" or "%D=\t", respectively. Use "%D=" for no delimiter.

Also, by default, the output is one data row per line. Thus, for one-dimensional row-wise arrays, all elements will be printed on one line, while one-dimensional column-wise array will have just one data element per line. For two-dimensional arrays, each line will contain one row of data. The number of elements per line can be controlled with the options `"%#[C|W]"`. For one-dimensional arrays, the number `#` is the number of elements to print per line. For two-dimensional arrays, `#` is the number of rows to print per line. If an optional `W` is added, the number becomes the number of elements to print per line, which can split two-dimensional arrays at different points in the rows. If an optional `C` is added to the option string, a backslash will be added to each line where a row is split. (The C-PLOT *scans.4* user function can properly interpret such "continued" lines for one-dimensional MCA-type array data.)

Finally, the various options can be combined in a single string. For example,

```
array_dump(data, "%15.4f", "%D=:", "%8W")
```

and

```
array_dump(data, "%15.4f%D=:%8W")
```

work the same.

`array_copy(`*`dst`*`, `*`src`*` [, ...])` — Fills consecutive bytes in the destination array (or subarray) *`dst`* with bytes from the source arrays or strings in the subsequent arguments. The arrays can be of different types, which allows creating a binary data stream of mixed types. If a source argument is not a data array, the string value of the argument is copied.

As an example of how `array_copy()` might be useful, consider a device that sends and receives a binary stream consisting of four floats followed by two integers then seven more floats. Here is how to prepare a byte array containing the mixed binary data types:

```
float array float_d[11]
ulong array long_d[2]
ubyte array ubyte_d[52]

# ... assign values to float_d and long_d, then ...

array_copy(ubyte_d, float_d[0:3], long_d, float_d[4:])
sock_put("host", ubyte_d)
```

Note, the source array is not erased prior to the copy. The above assignment could also be carried out as follows::

```
array_copy(ubyte_d[0:15,24:], float_d)
array_copy(ubyte_d[16:], long_d)
```

Four floats consume sixteen bytes. Two integers consume eight bytes. The

subarray notation selects the first sixteen bytes of `ubyte_d` for the first four floats, then skips eight bytes for where the integers will go, then copies the remainder of the floats. Since only as much data will be copied as is contained in the source array and since the source arrays are fixed size, it is not necessary to specify the final byte position in the destinations.

If the returned data uses the same format, floats and integers can be extracted using similar syntax:

```
sock_get("host", ubyte_d)
array_copy(float_d, ubyte_d[0:15], ubyte_d[24:])
array_copy(long_d, ubyte_d[16:23])
```

The function will only copy as many bytes as fit into the preallocated space of the destination array.

If the source arguments are not data arrays, spec will take the string value of the argument and copy the ASCII value of each byte to corresponding bytes in the destination. The terminating null byte is not copied. If the argument is a number, the string value of the number is what one would see on the display with the `print` command.

The function returns the updated array *`dst`*. If *`dst`* is a subarray, the full array is returned. A –1 is returned if *`dst`* is not a data array.

Note, this function allows arbitrary bytes to be copied to the elements of float and double arrays, which can result in undefined or not a number (NaN) values for those elements.

The `array_copy()` function appeared in spec release 6.00.07.

`array_read(`*`file`*`, `*`arr`*` [, `*`options`*`])` — Reads data from the ASCII text file *`file`*, and stuffs the data into the array *`arr`*. For a row-wise array, the values on each line of the file are assigned into successive columns for each row of the array. If there are more items on a line in the file than columns in the array, or if there are more points in the file than rows in the array, the extra values are ignored. For a column-wise array, each row of the data file is assigned to successive columns of the array.

If *`arr`* is a string array, successive bytes from each line of the file are assigned to elements of the array (as of spec release 6.04.05).

Lines beginning with the # character are ignored, except for the case where *`arr`* is a string array. There is no limit on the length of the input line. Prior to spec release 6.03.05, the maximum length was 2,048 characters.

The only currently recognized option is a `"C=#"`, where # is the starting column number in the file to use when making assignments (as of spec release

6.03.05).

Returns –1 if the file can't be opened, otherwise returns the number of points (bytes in the case of a string array) read and assigned.

`array_pipe(`*`program`* `[, `*`args`* `[, `*`arr_out`* `[, `*`arr_in`*`]]]))` –

`array_plot(`*`arr`* `[, `*`arr2`* `...]))` – Plots the data in the array *`arr`* (and optional additional array arguments). Depending on whether *`arr`* is a row-wise or column-wise array, the first column or first row elements are used for *x*. Subsequent elements (up to a maximum of 64) are plotted along the *y* axis. If preceded by a call of `plot_cntl("addpoint")` and the ranges have not changed, only the last point in the array is drawn. If preceded 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 `array_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.

`array_fit(`*`pars`*`, `*`arr`* `[, `*`arr2`* `...])` – Performs a linear fit of the data in the array *`arr`*. The fitted parameters are returned in the associative array *`pars`*. The function returns the *chi-squared* value of the fit, if the fit was successful. A –1 is returned if 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(`*`cmd`*`)` – Selects built-in plotting features. The argument *`cmd`* is a string of comma- or space-delimited options. The following options may be preceded by a minus sign to turn the associated feature off, nothing (or an optional plus sign) to turn the feature on or a question mark to return a value of one or a zero that indicates whether the associated feature is currently on or off:

`"xlog"` – Use a logarithmic *x* axis.

`"ylog"` – Use a logarithmic *y* axis.

`"xexact"` – Force *x*-axis minimum and maximum to be set to the scan endpoints (as opposed to being rounded).

`"colors"` – Enable the use of colors.

`"dots"` – Draw graphics-mode points with large dots.

`"lines"` – Connect graphics-mode points with lines.

`"ebars"` – Draw vertical lines through each point of length equal to the twice the square root of the *y* value.

`"persist"` – Keep graphics mode on after ordinary user input.

`"perpetual"` – Keep graphics mode on continuously (appropriate if using X windows, for example).

Other `plot_cntl()` options are:

`"colors=`*`bgnd`*`:`*`win`*`:`*`text`*`:`*`axis`*`:`*`symb`*`:..."` – Assigns colors for drawing the various graphics-mode elements. The values for *`bgnd`* (the background color of the area outside the axis), *`win`* (the background color of the area inside the axis), *`text`* (the color of the text), *`axis`* (the color of the axis) and *`symb`* `...` (the color of the plotting symbols) are integers.

The first 10 colors are standardized according to the following table:

0	background	(normally white or black)	
1	foreground	(normally black or white)	
2	blue	3	red
4	green	5	yellow
6	cyan	7	magenta
8	white	9	black

Other colors may be available depending on the particular device. You don't have to assign values to all colors.

`"colors[`*`numb`*`]"` – Returns the current color assignments, where *`numb`* is a number between zero and 67. Numbers zero through three return the colors assigned to the *`bgnd`*, *`win`*, *`text`* and *`axis`* elements respectively. Numbers from 4 through 67 return the colors assigned to the symbols for data elements zero through 63.

`"filter`*`numb`*`"` – Selects filter number *`numb`*, where *`numb`* can be any of the numerals from 1 through 5. All plotting commands are directed to this filter. The default filter is filter 1. Each filter is associated with a separate process. On an X windows display, each filter is associated with a separate window.

`"title=`*`string`*`"` – On an X windows display, the title given by string is used in the *XSetWMName()* and *XSetWMIconName()* calls to set the window and icon labels. With most X11 window managers, that means the title will appear in the window's title bar.

`"geometry=`*`width`*`x`*`height`*`+`*`xoff`*`+`*`yoff`*`"` – With the *x11* high-resolution plot windows, sets the size and position of the window. As with the conventional X11 syntax for specifying window geometry, not all parts of the geometry string are required.

`"open"` – Turn on graphics mode. If there is no graphics filter program currently active for the current filter number (see above), the filter program associated with the current `GTERM` variable is started. Recognized `GTERM` values are `vga`, `ega`, `cga`, `herc`, `x11`, and `sun`.

"close" – Turn off graphics mode, unless the perpetual or persistent mode has been selected.

"kill" – Turn off graphics mode and terminate graphics process.

"erase" – Clear the graphics-mode screen (or the text screen if graphics mode is off).

"addpoint" – Before a call to data_plot() will cause the plot to be made with minimal redrawing. Used for updated plotting during scans.

"addline" – Before a call to data_plot() will prevent the current data from being erased and the new data from changing the ranges when the new data points are drawn. Used for plotting several data sets from different data groups on top of each other.

"mca" – Before a call to data_plot() will cause the data points to be displayed using a minimal redrawing algorithm, appropriate for displaying data actively being accumulated by an MCA-type device. The "dots" and "ebars" modes must be turned off for the algorithm to work effectively.

"lp" – Before a call to data_plot() will generate printing instructions appropriate for plotting on a 132-column printer.

plot_move(*x*, *y* [, *string* [, *color*]]) – 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 *string* 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. 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(). Returns **true**.

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**.

splot_cntl(*cmd*) –

spec can store data in up to 256 independent data arrays called groups. Each group is configured (see below) to have a particular number of data elements per point. For example, each point in a group could have elements for H, K, L, and detector counts. Alternatively, each point could have just one element and be used to hold data obtained from an MCA.

Groups are configured using the data_grp() function. A group can have up to 2048 elements per point. The maximum number of points in a group is determined by the

product of the number of elements per point and the number of points. That product can be no more than 65,536, and may be slightly less depending on how the number of elements divides into 2048. The maximum number of points for all groups is 262,144. (These limits are arbitrary and are set to control the size of static data arrays and auxiliary files. If requested, CSS can make the limits larger.)

When starting spec for the first time or with the **–f** (fresh) flag, one data group (group 0) is configured for 4096 points, with each point consisting of two elements.

When leaving spec, the current data group configuration and data points are saved.

spec has several functions to manipulate the internal data. These functions allow unary and binary arithmetic operations, math functions and analysis operations to be performed on all the elements of a group or among elements in different groups.

In the functions described below, if an element number is negative, the element number is obtained by adding the number of elements per point in the group to the negative element number. For example, element –1 is the last element, element –2 is the second to last, etc.

All functions reset to command level if an invalid group, point or element is given as an argument. Functions that don't need to return anything in particular return zero.

`data_grp(`*`grp`*`, `*`npts`*`, `*`wid`*`)` – Configures data group *`grp`*. The group will have *`npts`* points, each having *`wid`* elements. If *`npts`* and *`wid`* 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 *`wid`* is zero, the group is eliminated. If *`npts`* is zero, as many points as possible are configured. If *`npts`* is negative, as many points as possible, but not more than *`–npts`* are configured. If *`grp`* is –1, the current group configuration is displayed.

`data_info(`*`grp`*`, `*`what`*`)` – Returns data group configuration information for group *`grp`*, according to the the string *`what`*. Values for *`what`* are:

`"npts"` – the number of configured points.

`"elem"` – the number of configured elements.

`"last"` – the number of the last point added to the group.

`"precision"` – the number of bytes per element, either 4 or 8.

If the group number is invalid, or if the string *`what`* is none of the above, returns –1.

`data_get(`*`grp`*`, `*`npt`*`, `*`elem`*`)` – Returns the value of element *`elem`* of point *`npt`* in group *`grp`*.

`data_put(`*`grp`*`, `*`npt`*`, `*`elem`*`, `*`val`*`)` – Assigns the value *`val`* to element *`elem`* of point *`npt`* in group *`grp`*.

`data_nput(`*`grp`*`, `*`npt`*`, `*`val0`*` [, `*`val1`*` ... ])` — Assigns values to point *npt* of group *grp*. Element 0 is assigned *val0*, element 1 is assigned *val1*, etc. Not all elements need be given, although elements are assigned successively, starting at element 0.

`data_uop(`*`g_src`*`, `*`e_src`*`, `*`g_dst`*`, `*`e_dst`*`, `*`uop`*` [, `*`val`*` ])` — Performs the unary operation specified by the string *uop* on element *e_src* for all points in group *g_src*. The results are put in element *e_dst* of the corresponding points in group *g_dst*. 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. Values for *uop* are:

`"clr"` — clear to zero.
`"fill"` — each element is set to point number, starting at 0.
`"neg"` — Negative of source.
`"abs"` — Absolute value of source.
`"inv"` — Inverse of source.
`"sin"` — Sine of source.
`"cos"` — Cosine of source.
`"tan"` — Tangent of source.
`"asin"` — Arcsine of source.
`"acos"` — Arccosine of source.
`"atan"` — Arctangent of source.
`"log"` — Natural logarithm of source.
`"exp"` — Exponential of source.
`"log10"` — Log base 10 of source.
`"pow"` — The *val* power of source.
`"copy"` — Value of source.
`"rev"` — Reversed copy of source.
`"sqrt"` — Square root of source.
`"set"` — All elements set to the value of *val*.
`"contract"` — Every *val* points are averaged to make a new point.
`"add"` — Source plus *val*.
`"sub"` — Source minus *val*.
`"mul"` — Source times *val*.
`"div"` — Source divided by *val*.

If any of the operations would result in an exception (divide by zero, log or square root of a negative number, etc), the operation is not performed and a count of the operations skipped is printed as an error message.

`data_bop(`*`g0_src`*`, `*`e0_src`*`, `*`g1_src`*`, `*`e1_src`*`, `*`g_dst`*`, `*`e_dst`*`, `*`bop`*`)` — Performs the binary operation specified by the string *bop* on elements *e0_src* and *e1_src* for

all points in the groups *g0_src* and *g1_src*. The results are put in element *e_dst* for the corresponding points of group *g_dst*. 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. Values for *bop* are:

"add" – The sum of the source elements.
"sub" – Source 0 minus source 1.
"mul" – The product of the source elements.
"div" – Source 0 divided by source 1.

If the divide would result in an exception, the operation is not performed and a count of the operations skipped is printed as an error message.

data_anal(*grp*, *start*, *npts*, *el_0*, *el_1*, *op* [, *val*]) – Performs the operations indicated by *op* on *npts* points in group *grp*, starting at point *start*. The operations use the values in element *el_0* (if applicable) and *el_1*. If *npts* is zero, the operations are performed on points from *start* to the last point added using data_nput() or data_put(). The values for *op* are:

"min" – Returns the minimum value of *el_1*. (*el_0* is unused.)
"max" – Returns the maximum value of *el_1*. (*el_0* is unused.)
"i_at_min" – Returns the point number of the data point with the minimum value of *el_1*. (*el_0* is unused.)
"i_at_max" – Returns the point number of the data point with the maximum value of *el_1*. (*el_0* is unused.)
"i_<=_value" – Returns the point number of the nearest data point in *el_1* at or below *val*, starting from the first point. (*el_0* is unused.)
"i_>=_value" – Returns the point number of the nearest data point in *el_1* at or above *val*, starting at the last point. (*el_0* is unused.)
"uhmx" – Returns the value in *el_0* corresponding to half the maximum value in *el_1* and at a higher index.
"lhmx" – Returns the value in *el_0* corresponding to half the maximum value in *el_1* and at a lower index.
"sum" – Returns the sum of the values in *el_1*. (*el_0* is unused.)
"fwhm" – Returns the full-width in *el_0* at half the maximum value of *el_1*.
"cfwhm" – Returns the center of the full-width in *el_0* at half the maximum value of *el_1*.
"com" – Returns the center of mass in *el_0* with respect to *el_1*. The value is the sum of the products of each *el_0* and *el_1* divided by the number of points.
"x_at_min" – Returns the value of *el_0* at the minimum in *el_1*.

"x_at_max" – Returns the value of *el_0* at the maximum in *el_1*.

"sumsq" – Returns the sum of the squares in *el_1*. (*el_0* is unused.)

The following operations treat a data group as a two dimensional data array with rows indexed by the point number and the columns indexed by the element number. The operations work on the portion of the group determined by the starting row *start* the number of rows *npts*, the starting column *el_0* and the end row *el_1*. As usual, if *npts* is zero, all points (rows) from *start* to the last are considered. A negative element (column) number is added to the group width to obtain the element (column) to use.

"gmin" – Returns the minimum value.

"gmax" – Returns the maximum value.

"gsum" – Returns the sum of all values.

"i_at_gmin" – Returns the index number of the minimum value. The index number is the row number times the group width plus the element number.

"i_at_gmax" – Returns the index number, as defined above, of the maximum value.

data_read(*file_name*, *grp*, *start*, *npts*) – Reads data from the ASCII file *file_name*, and stuffs the data into group *grp* starting at point *start*, reading up to *npts* points. If *npts* 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. Lines beginning with the # character are ignored. Returns –1 if the file can't be opened, otherwise returns the number of points read.

data_fit(*pars*, *grp*, *start*, *npts*, *el_data*, *el_data* [, ...]) – Performs a linear fit of the data in element *el_data* to the terms in the elements specified by *el_pars*. 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).

data_plot(*grp*, *start*, *npts*, *el_0*, *el_1* [, *el_2* ...]) – Plots the current data in group *grp* starting at point *start* and plotting *npts* points. Element *el_0* is used for x. Elements given by the subsequent arguments (up to a maximum of 64) are plotted along the y axis. The element arguments can be combined in a single space- or comma-delimited string, which can make creation of macros to plot a variable numbers of curves in the same plot window easier.

If *npts* is zero, only the points from *start* 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 *start* + *npts* − 1 is drawn. If preceded 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_dump(*grp*, *start*, *npts*, *el_0*, [, *el_1 ...*] [, *fmt1*] [, *fmt2*]) — Efficiently writes elements from group *grp* to turned on output devices. The starting point is *start* and the number of points is *npts*. The elements specified by *el_0*, *e1_1*, etc., are printed. If *el_0* is the string "all", all the elements for each point are printed. If *npts* is zero, only the points from *start* to the last point added using data_nput() or data_put() are printed. The element arguments can be combined in a single space- or comma-delimited string.

The optional argument *fmt1* is a string, having the format "%#", that specifies how many data points (specified by the number #) are to be printed on each line. If the number # is followed by the letter C, a backslash is added to each continued line, appropriate for saving MCA data in manageable length lines. New versions (since May 1, 1995) of the C-PLOT *scans.4* user function interpret the continued lines correctly for MCA data. 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.

Binary Input/Output

The facility for binary file input and output allows users and sites to create arbitrary binary file formats for writing and reading spec data arrays. C source code for a number of formats is included in the spec distribution.

fmt_read(*file*, *fmt*, *arr* [, *header* [, *flags*]]) —

`fmt_write(`*`file`*`, `*`fmt`*`, `*`arr`*` [, `*`header`*` [, `*`flags`*` ]]) —`

`fmt_close(`*`file`*`, `*`fmt`*`) —`

In these functions *`file`* is the name of the data file, *`fmt`* selects which format to use and *`arr`* is the data array. The optional *`header`* argument is an associative array that may containing identifying information to be saved with the binary values in the data array. The optional *`flags`* argument is reserved for future enhancements.

`data_pipe("?")` — Lists the currently running data-pipe processes with name and process id.

`data_pipe(`*`program`*`, "kill")` — Kills the process associated with *`program`*.

The Data-Pipe Facility

spec's `data_pipe()` function allows integration of external code with spec. With the data-pipe facility, spec sends information to the external program, allows the external program to execute for a time, and then receives information back from the external program. The information can be in the form of a string or a number, and can also include the contents of a spec data group or data array. The handshaking and data transfer between spec and the data-pipe program is done in an overhead module included in the spec distribution that is linked with the external code.

From spec, access to the data-pipe facility is through the `data_pipe()` function called from the user level. Usage is as follows.

`data_pipe(`*`program`*` [, `*`args`*` [, `*`grp_out`*`|`*`arr_out`*` [, `*`grp_in`*`|`*`arr_in`*` ]]]) —` Initiates or resumes synchronous execution of the special process named *`program`*. If *`program`* contains a `/` character, then it contains the complete absolute or relative path name of the program to run. Otherwise the program must be in the `SPECD`/*data_pipe* directory, where `SPECD` is the built-in spec variable containing the path name of spec's auxiliary file directory. You can use the string `"."` for *`program`* as an abbreviation for the same program name as used in the last call to `data_pipe()`.

The string value of *`args`* is made available to the user code in the program as described in the next section.

The optional arguments *`grp_out`* and *`grp_in`* are data group numbers. If *`grp_out`* is present, the contents of that group are sent to the data-pipe program. If *`grp_in`* is present, it is the number of the data group that will receive values from the data-pipe program. The data-pipe program configures the size of *`grp_in`* for an implicit call to `data_grp()` within `data_pipe()`. If the *`grp_in`* argument is absent, spec will not receive data-group data from the data-pipe program. If *`grp_out`* is also absent, group data won't be sent to the

data-pipe program. Even without group arguments, the data-pipe program can still return values to spec in the form of assigning a number or string return value to `data_pipe()`.

Either or both of the data group arguments can be replaced with the array arguments *arr_out* and *arr_in*. The arrays referred to by these arguments must be the data arrays declared explicitly with the `array` keyword. When sending array data to the data-pipe program, the array data is first converted to double precision floating point format. The received data is always double, but is converted to fit the declared data type of *arr_in*. Only as much data as will fit into the array will be assigned. The number of columns in *arr_in* should match the width of the data sent over by the data-pipe program. If not, the data will still be assigned to the array, but will be misaligned.

Prior to spec release 4.03.13, only one `data_pipe()` function could be active at a time.

The user C code can be complied and linked using the command

```
dpmake program [UOBJ=...] [LIBS=...] [optional_make_args]
```

The command `dpmake` is a short shell script which invokes the *make* utility using the makefile *data_pipe.mak* in the `SPECD/`*data_pipe* directory. The file *program*.c will be compiled and linked with the data-pipe overhead module, and the result placed in an executable file named *program*. If additional object modules or libraries need to be linked, they can be specified with the `UOBJ=` or `LIBS=` parameters. If the tools provided are not sufficient, you can create custom makefiles based on the distributed *data_pipe.mak*.

After linking *program*, move it to the `SPECD/`*data_pipe* directory for easy access by all users.

The subroutines available from the user C code portion of the data-pipe program are described below.

The skeleton user C-code part of the data-pipe program should contain the following:

```
#include <user_pipe.h>

user_code(argc, argv)
char    **argv;
{
      ...
}
```

The include file *user_pipe.h* contains declarations of the subroutines available in the C code. The file resides in the `SPECD/`*data_pipe* directory.

The subroutine `user_code()` is called by the overhead part of the data-pipe program each time `data_pipe()` is invoked in spec. The parameter *argc* is set to the number of space-delimited words present in the string value of the *args* parameter to `data_pipe()`. The parameter *argv* is an array of character pointers that point to each of the *argc* space-delimited words in the *args* string. Alternatively, the `get_input_string()` function (see below) returns the *args* string in its entirety.

The `user_code()` routine will be called every time the `data_pipe()` function is called from spec. The data-pipe program does not exit between calls of `user_code()`, so you should be careful about allocating memory or opening files each time `user_code()` is called without freeing the memory or closing the files each time `user_code()` returns. Alternatively, you can make sure such things are only done the first time `user_code()` is called.

Besides the `argc`, `argv` technique for accessing the *args* typed in the `data_pipe()` call, the following function is available:

`char * get_input_string()` — Returns a pointer to memory holding a copy of the second argument *args* entered with the call to `data_pipe()`.

If `data_pipe()` is sending a data group or array to the user code, the following subroutines provide access to the data parameters and values.

`int get_group_number()` — Returns the group number specified as the `data_pipe()` *grp_out* argument. A –2 is returned if an array was specified. A –1 is returned if neither data group or array was specified.

`int get_group_npts()` — Returns the number of points in the `data_pipe()` *grp_out* or the number of rows in *arr_out*.

`int get_group_width()` — Returns the number of elements per point in the `data_pipe()` *grp_out* or the number of columns in *arr_out*.

`int get_input_data(double *`*x*`, int` *pts*`, int` *wid*`)` — Transfers data from the *grp_out* or *arr_out* specified in the call to `data_pipe()` to the memory area indicated by the pointer *x*. The pointer *x* is treated as an array dimensioned as *x*[*pts*][*wid*]. If the data group/array has more points/rows than *pts* or more elements/columns than *wid*, only as many points/rows or elements/columns as are available in the data group/array are copied. Data from only a single element/column may be retrieved using one or more calls of `get_input_elem()` below. If the data in the data group from spec is float rather than double (which depends on spec's installation configuration), float-to-double conversion is done within the call to `get_input_data()`. The return value is the number of points/rows copied.

`int get_input_elem(double *`*x*`, int` *pts*`, int` *el*`)` — Transfers one element of the data from the *grp_out* or *arr_out* specified in the call to

data_pipe() to the memory area indicated by the pointer *x*. No more than *pts* points are copied from element/column *el* of the the data group/array. If the data in the data group from spec is float rather than double (which depends on spec's installation configuration), float-to-double conversion is done within the call to get_input_data(). The return value is the number of points/rows copied.

The following subroutines allow you to send group/array data back to spec when data_pipe() is invoked with a *grp_in* or *arr_in* argument. For a data group, the call to data_pipe() will implicitly call data_grp() to configure the size of the return group according to the parameters set in the following subroutines. For an array, the array must already be declared and dimensioned.

There are two ways to send group/array data back to spec. The subroutine set_return_data() allows you to send the entire data group in one call that passes both a pointer to the data and the data group size to the data-pipe program overhead code. Alternatively, you can use the set_return_group() subroutine to configure the data group/array size, followed by one or more calls to set_return_elem() to set one element/column of the data group/array at a time.

int get_return_group_number() — Returns the group number specified as the data_pipe() *grp_in* argument. A –2 is returned if an array was specified. A –1 is returned if neither data group or array was specified.

void set_return_data(double *x, int *pts*, int *wid*, int *last*) — Configures the return data group and copies the data at the same time. The pointer *x* is considered as an array of dimension *x*[*pts*][*wid*] for the purpose of transferring data to the data group. The argument *last* sets the index number of the last element added to the group, which is used by the various data analysis and plotting functions available in spec.

void set_return_group(int *pts*, int *wid*) — Configures the size of the return data group without copying data. This subroutine must be called once before calling set_return_elem() below.

void set_return_elem(double *x, int *pts*, int *el*, int *last*) — Copies one element to the return data group, which must have been previously configured by a call of set_return_group(), above. If the parameters *pts* or *el* exceed the values configured, or if the return group hasn't been configured, the subroutine returns –1. Otherwise zero is returned.

You can set the value that the data_pipe() function returns in spec from the user C code in your *data-pipe* process. You can have data_pipe() return a number or a string or, if necessary, reset to command level. If no explicit return value is assigned in the user C code, data_pipe() returns zero.

`int set_return_string(char *s)` — Sets the return value of `data_pipe()` to the string *s*. This subroutine returns –1 if memory could not be obtained for the string *s*, otherwise it returns zero.

`void set_return_value(double v)` — Sets the return value of `data_pipe()` to the value *v*.

`void do_error_return()` — Calling this subroutine from the user C code causes control to pass back to spec without returning data group or array values, if they have been set. The return value of `data_pipe()` will be the value set by `set_return_value()` above, if such a value has been set, otherwise the return value of `data_pipe()` will be –1. This subroutine does not return.

`void do_abort_return()` — Calling this subroutine from the user C code causes control to pass back to spec without returning data group or array values, if they has been set. In spec, there is no return from `data_pipe()`, rather spec resets to command level. This subroutine does not return.

`void do_quit_return()` — Calling this subroutine from the user C code causes control to pass back to spec normally as if `user_code()` returned normally, but the data-pipe program will then exit. This subroutine does not return.

Client/Server Functions

`prop_send(`*property*`, `*value*`)` —

`prop_get(`*host*`, `*property*`)` —

`prop_put(`*host*`, `*property*`, `*value*`)` —

`prop_watch(`*host*`, `*property*`)` —

`remote_stat(`*host*`)` —

`remote_stat(`*host*`, "?")` —

`remote_par(`*host*`, "connect")` —

`remote_par(`*host*`, "close")` —

`remote_par(`*host*`, "abort")` —

`remote_par(`*host*`, "timeout" [, `*value*`])` —

`remote_cmd(`*host*`, `*cmd*`)` —

`remote_eval(`*host*`, `*cmd*`)` —

id = remote_async(*host*, *cmd*) —

remote_poll(*id*, "status") —

remote_poll(*id*) —

encode(*format*, *obj* [, ...]) — Returns a string representation of the spec data objects *obj* ... in the specified *format*.

decode(*format*, *str* — Returns a spec data object obtained from the string *str* in *format*.

Hardware Functions and Commands

Controlling Motors

move_all — This command sets motors in motion. The sequence of events is as follows. For some motor controllers, spec first examines the controller registers of all nonbusy motors and makes sure the contents agree with the current positions in program memory. If there is a discrepancy, the user is asked to choose the correct position. Next, spec prepares to move all motors that are not already at the positions specified in the built-in A[] array, interpreted in user units. A motor will not be moved if it is currently moving or if it is marked as protected (via the configuration file) or unusable (due to failure of a hardware presence test). If the target position of any of the motors is outside the software limits, the entire move is canceled, and the program resets to command level. Otherwise, the motors are started, and the command returns.

The sequence of commands when using move_all should almost always be,

```
wait(1)         # Wait for moving to finish
read_motors(0)  # Put current positions of all motors in A[]
(Assign new values to elements of A[] to be moved)
move_all        # Move to those positions
```

If read_motors() is called before the motors have stopped, the values in A[] will reflect the motor positions before they stopped. If read_motors() is not called at all, or if you do not explicitly assign a value to each element of A[], then you will not know for sure where some motors will be going when move_all is called.

A **^C** halts moving, as does the sync command.

move_cnt — This command is similar to move_all, described above, but with the following differences. Just before the motors are started, the clock/timer is enabled and programmed to gate the scalers with a longer-than-necessary count time. The motors are then started at the base rate set in the *config* file, but are

not accelerated to the steady-state rate. No backlash correction is done at the end of the move. When the move is completed, the clock/timer is stopped. The `move_cnt` command is used in powder-averaging scans. (See the powder-mode macros on page 178.)

`sync` – If any motors are moving, they are halted. The motor positions maintained by the motor controller are then compared with the motor positions currently set in the program. If there is a discrepancy, the user is asked which should be changed. The `sync` command is used to place the motor hardware in a known state and is supposed to fix any problems in communicating with the controllers.

`motor_mne(`*`motor`*`)` – Returns the string mnemonic of motor number *`motor`* as given in the configuration file. (Mnemonics are, at most, 7 characters long.) Resets to command level if not configured for *`motor`*.

`motor_name(`*`motor`*`)` – Returns the string name of motor number *`motor`* as given in the configuration file. (Names are, at most, 15 characters long.) Returns `"?"` if not configured for *`motor`*.

`motor_num(`*`mne`*`)` – Returns the motor number corresponding to the motor mnemonic *`mne`*, or –1 if there is no such motor configured. As of SPEC release 6.05.01, *`mne`* can be a variable or an expression. If *`mne`* is an uninitialized variable, –1 is returned.

`motor_par(`*`motor`*`, `*`par`*` [, `*`val`*` ])` – Returns or sets configuration parameters for motor *`motor`*. Recognized values for the string *`par`* follow. Note, not all parameters are meaningful for all motor controllers.

`"step_size"` – returns the current step-size parameter. The units are generally in steps per degree or steps per millimeter. If *`val`* is given, then the parameter is set to that value, but only if changes to the step-size parameter have been enabled using `spec_par("modify_step_size", "yes")`.

`"acceleration"` – returns the value of the current acceleration parameter. The units of acceleration are the time in milliseconds for the motor to accelerate to full speed. If *`val`* is given, then the acceleration is set to that value.

`"base_rate"` – returns the current base-rate parameter. The units are steps per second. If *`val`* is given, then the base rate is set to that value.

`"velocity"` – returns the current steady-state velocity parameter. The units are steps per second. If *`val`* is given, then the steady-state velocity is set to that value.

`"backlash"` – returns the value of the backlash parameter. Its sign and magnitude determine the direction and extent of the motor's backlash

correction. If *val* is given, then the backlash is set to that value. Setting the backlash to zero disables the backlash correction.

`"config_step_size"` – returns the step-size parameter contained in the *config* file.

`"config_acceleration"` – returns the acceleration parameter contained in the *config* file.

`"config_velocity"` – returns the steady-state velocity parameter contained in the *config* file.

`"config_base_rate"` – returns the base-rate parameter contained in the *config* file.

`"config_backlash"` – returns the backlash parameter contained in the *config* file.

`"controller"` – returns a string containing the controller name of the specified motor. The controller names are those used in spec's *config* files.

`"unit"` – returns the unit number of the specified motor. Each motor controller unit may contain more than one motor channel.

`"channel"` – returns the channel number of the specified motor.

`"responsive"` – returns a nonzero value if the motor responded to an initial presence test or appears otherwise to be working.

`"active"` – returns a nonzero value if the motor is currently moving.

`"disable"` – returns a nonzero value if the motor has been disabled by software. If *val* is given and is nonzero, then the motor is disabled. If *val* is given and is zero, the motor becomes no longer disabled. A disabled motor channel will not be accessed by any of spec's commands, and, of course, cannot be moved. Any `cdef()`-defined macros will automatically exclude the portions of the macro keyed to the particular motor when the motor is software disabled.

`"slop"` – returns the value of the slop parameter. If *val* is given, sets the slop parameter. When this parameter is present, discrepancies between hardware and software motors positions are silently resolved in favor of the the hardware when the number of steps in the discrepancy is less than the value of the slop parameter. (Not yet implemented for all motor controllers.)

`"home_slew_rate"` – returns the value of the home-slew-rate parameter. If *val* is given, sets the parameter. This parameter is the steady-state velocity used during a home search. (Only available for selected controllers.)

`"home_base_rate"` – returns the value of the home-base-rate parameter. If *val* is given, sets the parameter. This parameter is the base-rate velocity used during a home search. (Only available for selected controllers.)

"home_acceleration" – returns the value of the home-acceleration parameter. If *val* is given, sets the parameter. This parameter is the acceleration used during a home search. (Only available for selected controllers.)

"dc_dead_band" – returns the value of the dead-band parameter for certain DC motors. Sets the parameter if *val* is given.

"dc_settle_time" – returns the value of the settle-time parameter for certain DC motors. Sets the parameter if *val* is given.

"dc_gain" – returns the value of the gain parameter for certain DC motors. Sets the parameter if *val* is given.

"dc_dynamic_gain" – returns the value of the dynamic-gain parameter for certain DC motors. Sets the parameter if *val* is given.

"dc_damping_constant" – returns the value of the damping-constant parameter for certain DC motors. Sets the parameter if *val* is given.

"dc_integration_constant" – returns the value of the integration-constant parameter for certain DC motors. Sets the parameter if *val* is given.

"dc_integration_limit" – returns the value of the integration-limit parameter for certain DC motors. Sets the parameter if *val* is given.

"dc_following_error" – returns the value of the dc-following parameter for certain DC motors. Sets the parameter if *val* is given.

"dc_sampling_interval" – returns the value of the sampling-interval parameter for certain DC motors. Sets the parameter if *val* is given.

"encoder_step_size" – returns the value of the encoder step size parameter. Sets the parameter if *val* is given.

"step_mode" – returns the value of the step-mode parameter. Sets the parameter if *val* is given. A zero indicates full-step mode, while a one indicates half-step mode.

"deceleration" – returns the value of the deceleration parameter. Sets the parameter if *val* is given.

"torque" – returns the value of the torque parameter. Sets the parameter if *val* is given.

Rereading the *config* file resets the values of all the motor parameters to the values in the *config* file. Little consistency checking is done by spec on the values programmed with `motor_par()`. You must be sure to use values meaningful to your particular motor controller.

In addition, device-dependent values for *par* are available for specific motor controllers. See the *Hardware Reference* for values for specific controllers.

`dial(`*motor*`, `*user_angle*`)` – Returns the dial angle for motor *motor* corresponding to user angle *user_angle* using the current *offset* between user and dial angles for *motor*. The value returned is *(user_angle – offset)* / *sign*, where *sign* is ±1

and is set in the *config* file. The value is rounded to the motor resolution as set by the step-size parameter in the *config* file. Resets to command level if not configured for motor *motor*.

`read_motors(`*how*`)` — Reads the current motor positions from the motor controllers and places the values in the `A[]` array, depending on the value of the argument *how*. If bit 1 is set, the function returns dial values, otherwise user values are returned. If bit 2 is set, a forced read of all hardware takes place. (For efficiency, normally most motor controllers are not read if the position hasn't been changed by a move.) If bit 3 is set and if there is a discrepancy between the software and hardware, the software will be silently corrected to match the hardware. Note, the forced-read and "silent-sync" features are not yet implemented for all motor controllers. Check the *Hardware Reference* or contact CSS for hardware-specific information.

`move_info([`*motor*`|`*keyword*`|`*motor*`, `*keyword*`])` — The `move_info()` function returns information about what would happen on a subsequent `move_all` command given the current motor positions and current values in the `A[]` array. Such a function might be called, for example, within the `user_premove` macro to determine which motors will be moved to allow extra limit checking involving the relative postions of motors.

If called with no arguments, returns a two-dimensional associative array containing the move information. The array is indexed by motor number and information keyword. The keywords are:

`"to_move"`	nonzero if motor will move
`"error"`	reason the motor will not move
`"commanded"`	the commanded position
`"magnitude"`	magnitude of the move in user units
`"current"`	current position in user units
`"current_dial"`	current position in dial units
`"target"`	target position in user units
`"target_dial"`	target position in dial units
`"backlash"`	backlash for this move in user units
`"leftover"`	remainder due to motor resolution

If called with a single argument that is one of the above keywords, the function returns a one-dimensional associative array indexed by motor number containing values for that keyword for each motor. If called with a motor number or mnemonic as a single argument, the function returns a one-dimensional associative array, indexed by the above keywords containing values for the one motor. If called with two arguments, motor number and keyword, the function

returns the corresponding single value.

No matter how the function is called, the internal code will calculate values for all motors. Thus, if multiple values are needed, it is most efficient and recommended to call the function once selecting arguments that will return all the needed values.

The "to_move" element will be 1 or 0, indicating whether the motor would move or not. If there is condition that prevents the move, the "error" element will contain one of these strings:

"low limit"	move exceeds low limit
"high limit"	move exceeds high limit
"busy"	motor is busy
"read only"	motor is configured as read only
"protected"	motor configuration does not allow moving
"disabled"	motor has been disabled
"externally disabled"	shared motor has been disabled
"unusable"	motor did not respond to presence test

Otherwise, there will be no "error" array element.

The "target" and "target_dial" values are the final position after backlash. The "magnitude" value contains the distance to the target position and does not include the magnitude of the backlash.

The "leftover" value is the fractional value that is the difference between the requested position and the achievable position given the finite resolution of the motor. For example, if a motor has 1000 steps per degree, each step corresponds to 0.001 degrees. If one asks to move the motor to a position of 1.0004 degrees, the motor will move to 1 degree and the leftover value will be 0.0004 degrees.

The "commanded" value is the target position in user units to the full precision requested. The other postion-related values are rounded to the motor resolution. The "commanded" position is saved after a move and is available using special arguments to the built-in read_motors() functions.

As mentioned above, if multiple values are needed, it is better to make a single call of move_info() saving the return values in an array, rather than making multiple calls, as each call involves calculations for all the motor positions and values, even if only selected values are returned. For example,

```
{
   local i, m[]
   m = move_info()
   for (i = 0; i < MOTORS; i++)
        if (m[i]["to_move"]) {
                ...
        }
}
```

For the most part, the `move_info()` function will reflect what will happen on the next `move_all` command. However, for shared motors that can be moved by other processes or for motors that have positions that drift or have jitter, the status and position may change between the `move_info()` call and the `move_all` call.

`chg_dial(`*`motor`*`, `*`dial_angle`*`)` — Sets the dial position of motor *`motor`* to *`dial_angle`*. Returns nonzero if not configured for *`motor`* or if the protection flags prevent the user from changing the limits on this motor. Resets to command level if any motors are moving.

`chg_dial(`*`motor`*`, `*`cmd`*`)` — Starts motor *`motor`* on a home or limit search, according to the value of *`cmd`* 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. (At present, most motor controllers do not implement the home or limit search feature.) Returns –1 if not configured for motor *`motor`* or if the motor is protected, unusable or moving, else returns zero.

`get_lim(`*`motor`*`, `*`flag`*`)` — Returns the dial limit of motor number *`motor`*. If *`flag`* > 0, returns the high limit. If *`flag`* < 0, returns the low limit. Resets to command level if not configured for *`motor`*.

`user(`*`motor`*`, `*`dial_angle`*`)` — Returns the user angle for *`motor`* corresponding to dial angle *`dial_angle`* using the current *offset* between user and dial angles for *`motor`*. The value returned is $sign \times dial\_angle + offset$, where *sign* is ±1 and is set in the *config* file. The value is rounded to the motor resolution as set by the step-size parameter in the *config* file. Resets to command level if not configured for *`motor`*.

`chg_offset(motor, user_angle)` — Sets the offset between the dial angle and the user angle, using the current dial position and the argument *user_angle* for motor *motor* according to the relation *user_angle* = *offset* + *sign* × *dial_angle* where *sign* is ±1 and is set in the *config* file. Returns nonzero if not configured for *motor*. Resets to command level if any motors are moving.

`set_lim(motor, low, high)` — Sets the low and high limits of motor number *motor*. *low* and *high* are in dial units. It does not actually matter in which order the limits are given. Returns nonzero if not configured for *motor* or if the protection flags prevent the user from changing the limits on this motor. Resets to command level if any motors are moving.

Counting

`mcount(counts)` — Starts the timer/clock counting for *counts* monitor counts. Returns zero.

`tcount(t)` — Starts the timer/clock counting for *t* seconds, where *t* may be nonintegral. Returns zero.

`getcounts` — Loads the built-in array `S[]` with the contents of the scalers.

`cnt_mne(counter)` — Returns the string mnemonic of counter number *counter* as given in the configuration file. (Mnemonics are, at most, 7 characters long.) Resets to command level if not configured for *counter*.

`cnt_name(counter)` — Returns the string name of counter number *counter* as given in the configuration file. (Names are, at most, 15 characters long.) Returns `"?"` if not configured for *counter*.

`cnt_num(mne)` — Returns the counter number corresponding to the counter mnemonic *mne*, or −1 if there is no such counter configured. As of SPEC release 6.05.01, *mne* can be a variable or an expression. If *mne* is an uninitialized variable, −1 is returned.

`counter_par(counter, par [, val ])` — Returns or sets parameters associated with counter number *counter* as given in the configuration file. The following *par* arguments are supported for all counters:

`"unit"` — returns the unit number of the indicated counter.

`"channel"` — returns the channel number of the indicated counter.

`"scale"` — returns the value of the scale factor set in the *config* file for the indicated counter.

`"responsive"` — returns nonzero if the hardware appears to be working for the indicated counter.

`"controller"` — returns a string that indicates the controller type of the indicated counter.

`"disable"` — returns a nonzero value if the counter has been disabled by software. If *val* is given and is nonzero, then the counter is disabled. If *val* is given and is zero, the counter becomes no longer disabled. A disabled counter channel will not be accessed by any of spec's counting commands. Any `cdef()`-defined macros will automatically exclude the portions of the macro keyed to the particular counter when the counter is software disabled.

In addition, device-dependent values for *par* are available for specific counter models. See the *Hardware Reference* for values for specific controllers.

The counting functions `mcount()` and `tcount()` both program the timer/clock for a specified count time. Before the count period begins, both functions clear and enable all configured scalers and MCAs. The routines return immediately, as counting is asynchronous. Use `wait()`, described below, to determine if counting has been completed. A **^C** will halt the timer/clock.

When the count time has expired, or counting is aborted by a **^C**, the scalers and MCAs are disabled. Normally, enable signals from the timer/clock are used to directly gate the scalers or MCAs. Software gating takes place whether or not hardware gating is in place and can be used in lieu of hardware gating, although the interval over which the gating occurs will not be as precisely controlled.

Miscellaneous

`reconfig` — Reconfigures the hardware. This command obtains any modified configuration information, including hardware devices and types, CAMAC slot assignments and motor parameters and settings, from the *config* and *settings* files. The sequence of events is as follows:

First, spec waits for all asynchronous activity (moving and counting) to finish. It then does a `sync` of the motor controller registers, comparing them with the internal program positions. Next, all open devices are closed. The *config* file is then read to obtain the configuration information, and the program opens and possibly does hardware presence tests on the selected devices. Finally, the internal program motor positions are updated from the *settings* file and then resynchronized with the motor hardware.

`set_sim(`*how*`)` — If *how* is 0, simulate mode is turned off. If *how* is 1 (or positive), simulate mode is turned on. In either case the program waits for moving and counting to finish before changing the mode, and the function returns the previous mode (0 or 1). If *how* is –1 (or negative) the function returns the current

value of simulation mode without changing it. Whenever simulation mode is turned off, the motor settings file is reread to restore the motor positions. Simulation mode cannot be turned off if spec was invoked with the **–s** flag.

`wait()` – Waits for all active motors, timers, counters, MCA- and image-type activity to complete. Returns **true**.

`wait(`*mode*`)` – Waits for specified activity to complete or returns status indicating whether specified activity is active. The function `wait()` is used to synchronize the flow of commands in spec with moving, counting and other activity. Since the built-in commands and functions `move_all`, `move_cnt`, `tcount()` and `mcount()` return immediately after starting moving or counting, macros need to include some form of `wait()` if the next command requires the previous move or count to complete.

Bits set in the *mode* argument affect the behavior as follows:

Bit	Value	Activity Waited For Or Other Action
	0	Moving, counting and other acquisition
0	1	Moving
1	2	Counting (by the master timer)
2	4	Other acquisition (MCAs, CCDs, etc.)
3	8	Remote connections and remote asynchronous events
4	16	Shared motors started by other clients
5	32	Return zero or nonzero to indicate if busy

If *mode* is a negative number, `wait()` will behave as for *mode* = 0, but a message will be printed showing what is being waited on.

For acquisition devices with `"auto_run"` mode enabled (such devices are started automatically during counting), waiting for counting will also include waiting for those devices.

When spec is running as client to a spec server, bit 3 checks if `remote_async()` replies have all arrived. In addition, bit 3 also checks if all configured spec servers have connected and if all spec server and EPICS remote motors have connected.

Waiting for spec server and remote motor connections is mainly an issue on start up or after `reconfig`. One might use `wait(8)` or `wait(0x28)` in the built-in special macro `config_mac` if it is important to delay until all connections are up. Note, until remote spec server and EPICS motors are fully connected and usable, the positions reported for those motors will be the last saved positions from spec's *settings* file.

When spec is configured with shared motors either on a spec server or using EPICS channel access, if those motors are started by a different client, setting bit 4, as in `wait(16)` will cause spec to wait until those motors have completed their move. Waiting can be interrupted with a **^C**, but that will not stop the motors.

Also, note that `wait(0)` does not check for the events flagged by bits 3 or 4. To wait for remote events or externally busy motors requires explicitly setting bits 3 or 4 in *mode*. Also, a **^C** interrupts a `wait(8)` or `wait(16)` but doesn't change the conditions that caused `wait(8)` or `wait(16)` to block. That is, the next wait(8) will still block if there are still pending connections, and the next wait(16) will still block if the external motors are still moving.

If bit 5 (0x20) in *mode* is set, `wait()` returns **true** (1) if the activities flagged by bits 0, 1, 2, 3 or 4 are still going on, otherwise `wait()` returns **false** (0).

`stop()` — Stops all asynchronous activity. Returns **true**.

`stop(`*mode*`)` — If *mode* has bit one set (1), stops all motors that are moving. If *mode* has bit two set (2), stops the timer, counters and any other data acquisition (multi-channel scaling, for example).

MCA (1D) Data Acquisition

`mca_sel(`*n*`)` — Selects which MCA-type device to use with subsequent `mca_get()`, `mca_put()` and `mca_par()` commands. The numbering of MCA-type devices is set 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. The selected MCA-type device does not change when reading the *config* file with the `reconfig` command.

`mca_sel("?")` — Lists the configured MCA devices, with the currently selected device marked with an asterisk.

`mca_par(`*par* `[,` *val* `])` — A device-dependent function to access various features and parameters of the currently selected MCA-type device. The string *par* selects an option. The argument *val* contains an optional numeric value. See the help file for the particular device for implemented options and return values.

`mca_get(`*grp, elem*`)` or `mca_get(`*array*`)` — Gets data from the currently selected MCA-type device, and transfers it to element *elem* of data group *grp* or to the elements of the data array *array*. Generally returns the number of points read or –1 for failure.

`mca_put(grp, elem)` or `mca_put(array)` — Sends data from data group *grp*, element *elem* or from the data array *array* to the currently selected MCA-type device. Generally returns the number of points written or –1 for failure.

`mca_spar(sel, par [, val ])` — As `mca_par()` above, but selects which MCA device with the *sel* argument.

`mca_sget(sel, grp, elem)` or `mca_sget(sel, array)` — As `mca_get()` above, but selects which MCA device with the *sel* argument.

`mca_sput(sel, grp, elem)` or `mca_sput(sel, array)` — As `mca_put()` above, but selects which MCA device with the *sel* argument.

Image (2D) Data Acquisition

`image_par(sel, par [, val ])` —

`image_get(sel, array)` —

`image_put(sel, array)" —`

Socket Functions

spec user-level socket functions connect and communicate with sockets created by other processes on the local platform or on a remote host. Most of the function calls require a string in the form "*host*:*port*" as the first argument to specify the socket. The *host* can be specified by a symbolic name or by an IP address. The *port* number is assigned by the process that created the socket.

`sock_get("host:port")` — Reads and returns as many characters as are already available. If no characters are immediately available, waits for input and returns the first character(s) that show up, or returns a null string if no characters arrive before the time-out expires. The maximum number of characters that can be read at a time is this mode is 8191 characters.

`sock_get("host:port", n)` — Reads up to *n* characters or until a timeout. If *n* is zero, the routine reads up to a newline or the maximum of 8191 characters, whichever comes first. In any case, if the read is not satisfied before a timeout, the routine returns the null string.

`sock_get("host:port", eos)` — Reads characters until a portion of the input matches the string *eos* and returns the string so obtained, including the end-of-string characters. If no match to the end-of-string characters is found within the timeout period, the null string is returned.

`sock_get("host:port", d)` — Reads incoming bytes into the data array *d*. The size of *d* determines how many bytes are to be read. Sub-array syntax can be used to limit the number of bytes. The function returns the number of array elements read, or zero if the read times out. Note, no byte re-ordering is done for short- or long-integer data, and no format conversions are done for float or double data.

`sock_get("host:port", mode)` — If *mode* is the string `"byte"`, reads and returns one unsigned binary byte. If *mode* is the string `"short"`, reads two binary bytes and returns the short integer so formed. If *mode* is the string `"long"`, reads four binary bytes and returns the long integer so formed. The last two modes work the same on both *big-endian* and *little-endian* platforms. On both, the incoming data is treated as *big*endian If the incoming data is *little*endian use `"short_swap"` or `"long_swap"`.

`sock_put("host:port", s)` — Writes the string *s* to the socket described by "*host*:*port*. Returns the number of bytes written.

`sock_put("host:port", d, [, cnt])` — Writes the contents of the data array *d* to the socket described by "*host*:*port*. By default, the entire array (or subarray, if specified) will be sent. The optional third argument *cnt* can be used to specify the number of array elements to send. For short and long integer arrays, the data will be sent using native byte order. The `"swap"` option of the `array_op()` function can be used to change the byte order, if necessary. No format conversions are available for float or double data. Returns the number of bytes written.

`sock_par("host:port", cmd [, arg ])` — Accesses various features for the given socket with values for *cmd* as follows:

`"?"` — Lists the available commands.

`"show"` — Lists the existing sockets along with additional information, such as whether the socket is UDP type, whether the socket is internal (as opposed to a user-level socket created by one of the functions described in this document) and whether the socket is set up for listening. The command does not check whether the connection is still alive.

`"info"` — Returns a string that contains the information displayed by the `"show"` command described above.

`"connect"` — Opens a socket to the specified host and port. Returns **true** for success and **false** for failure. With the string `"silent"` as optional argument, error messages won't be shown (as of spec release 6.00.02).

`"connect_udp"` — Opens a socket to the specified host and port using the UDP protocol. Returns **true** for success and **false** for failure. With the string `"silent"` as optional option, error messages won't be shown (as of spec

release 6.00.02).

`"listen"` – Sets up a socket for listening, allowing another instance of spec or some other program to make a connection.

`"close"` – Closes the socket associated with the specified host and port.

`"flush"` – Flushes spec's input queue for the socket at "*host*:*port*. The input queue may contain characters if a `sock_get()` times out before the read is satisfied, or if more characters arrive than are requested.

`"ignore_sim"` – With *arg* set to 1 or 0, turns *ignore-simulate* mode on or off, respectively. Otherwise, returns the current state. When *ignore-simulate* mode is on, the `sock_get()`, `sock_put()` and `sock_par()` commands will work even when simulate mode is on. Note, simulate mode must be off to create a new socket connection.

`"queue"` – Returns the number of characters in the socket's input queue. The input queue may contain characters if a `sock_get()` times out before the read is satisfied, or if more characters arrive than are requested. `"timeout"` 1i Returns or sets the read timeout for the socket described by "*host*:*port*. The units for *arg* are seconds. A value of zero indicates no timeout – a `sock_get()` will wait until the read is satisfied or is interrupted by a `^C`. The smallest allowed value of 0.001 will cause the `sock_get()` to return immediately. A negative value resets the timeout to the default of five seconds.

`"nodelay"` – A value for *arg* of 1 or 0 sets or clears the state of the *TCP_NODELAY* socket option. With no argument, the current state is retruned. Normally, the underlying TCP protocol sends data along as it is made available. However, if the previous data packet has not yet received acknowledgment from the client, the protocol holds onto and gathers small amounts of data into a single packet which will be sent once the pending acknowledgment is received or the size of the packet exceeds a threshold. This algorithm increases network efficiency. For some clients that send a stream of short packets that receive no replies, this algorithm may cause unwanted delays. Set the `"nodelay"` option to 1 to turn off the algorithm, which corresponds to setting the *TCP_NODELAY* option at the system level.

All the `sock_get()` calls will store leftover bytes in a queue. Contents from the queue will be returned on a subsequent `sock_get()` call. Bytes are leftover if the read finishes with a timeout, if more bytes have arrived than are asked for or if more bytes are available after an end-of-string match. Use the `"flush"` option of `sock_par()` to clear the input queue, if needed.

To transfer binary byte streams containing null bytes, use the data-array versions of `sock_get()` and `sock_put()` with byte arrays. Null bytes mark the

end of a normal string.

Note, the "connect" command isn't required to open a TCP connection, as the sock_get() and sock_put() functions will automatically open the connection if it doesn't already exist. The return value from the "connect" command, however, may be useful as a test on whether a given socket can be created. To create a UDP connection, however, the "connect_udp" command must be used.

Connections remain open until closed with the "close" option to "spec_par()". Sockets created at user level are not closed on a hardware "reconfig".

The following example connects to the *echo* service on port 7 of the local host:

```
24.FOURC> sock_put("localhost:7", "This is a test.\n")

25.FOURC> print sock_get("localhost:7")
 This is a test.

26.FOURC>
```

RS-232 Serial Interfaces

Generic user-level access to the serial ports is through the ser_get(), ser_put() and ser_par() functions described in the following sections. The generic serial devices are configured on the Interfaces screen of the configuration editor. Each serial device is numbered, starting from zero, and that number is the first parameter *addr* in the functions below. Up to 21 serial devices can be configured, numbered from 0 to 20.

Do not configure a generic serial device when the associated device node is for a motor controller, counter/timer or other acquisition device that uses spec's built-in support. The serial device associated with such controllers is specified as part of the controller configuration

The default serial interface is through the built-in standard UNIX serial driver. However, the same user functions can access serial devices configured to use EPICS, TACO, or SOCKET interfaces.

Supported baud settings are 300, 600, 1200, 1800, 2400, 4800, 7200, 9600, 14400, 19200, 28800, 38400, 57600, 115200, 230400, 460800, 921600, 1000000, 1152000, 1500000, 2000000, 2500000, 3000000, 3500000 and 4000000 baud. However, not all baud rates are supported by all platforms and by all serial interface hardware.

The baud rate setting is ignored for the SOCKET type of interface. See the documentation associated with a particular Ethernet-to-serial device for procedures to set the

serial port parameters.

Serial interfaces have associated modes, some of which can be set in spec's configuration editor and some with the `ser_par()` function. There are many more mode parameters than those described below. Only the parameters that have been found to be needed by spec users can be configured.

The standard serial ports can be configured in either *raw* mode or several flavors of *cooked* mode. In *raw* mode, the kernel does minimal processing of the bytes transmitted and received, generally passing all of the 256 possible values through. Also, the received bytes are available to spec as soon as they are received by the kernel. For transferring binary data, *raw* mode is essential. On some platforms, a seven-bit raw mode is available, where the eighth bit is used for parity.

In *cooked* mode, the kernel buffers the incoming data. The input data only becomes available to be read by spec when a newline or carriage return is received. Also, the kernel may do some processing of the data, such as converting tabs to spaces on output or processing delete or line-erase characters on input. The character processing makes *cooked* mode inappropriate for receiving binary data. The various flavors of *cooked* mode implemented in spec set whether to use even or odd parity or no parity, whether to disable software flow control and whether to ignore carriage returns on input.

spec does turn off input echoing in both *raw* and *cooked* modes.

Note, the `TACO`, `EPICS` and `SOCKET` interface types only support *raw* mode.

`ser_get(`*`addr`*`)` — If the serial device *addr* is in *cooked* mode, reads and returns a string of bytes, up to and including a newline character, or returns the null string if the read times out. If the device is in *raw* mode, the function reads and returns as many characters as are already available in the queue. If no characters are available, waits for a character and returns it, or returns a null string if no characters become available within the time-out period. The maximum string length in this mode is 8191 characters.

`ser_get(`*`addr, n`*`)` — If the serial device *addr* is in *cooked* mode, reads up to a newline, but no more than *n* bytes and returns the string so obtained. In *cooked* mode, no characters can be read until a newline is received. In *raw* mode, reads up to *n* characters or until a timeout. If *n* is zero, the routine reads up to a newline or the maximum of 8191 characters, whichever comes first. In both cases, if the read is not satisfied before a timeout, the routine returns the null string.

`ser_get(`*`addr, eos`*`)` — Reads characters until a portion of the input matches the string *eos* and returns the string so obtained, including the end-of-string characters. If no match to the end-of-string characters is found within the timeout

period, the null string is returned.

`ser_get(`*`addr, d`*`)` — Reads incoming bytes into the data array *d*. The size of *d* determines how many bytes are to be read. Sub-array syntax can be used to limit the number of bytes. The function returns the number of array elements read, or zero if the read times out. Note, no byte re-ordering is done for short- or long-integer data, and no format conversions are done for float or double data.

`ser_get(`*`addr, mode`*`)` — If *mode* is the string `"byte"`, reads and returns one unsigned binary byte. If *mode* is the string `"short"`, reads two binary bytes and returns the short integer so formed. If *mode* is the string `"long"`, reads four binary bytes and returns the long integer so formed. The last two modes work the same on both *big-endian* and *little-endian* platforms. On both, the incoming data is treated as *big*endian If the incoming data is *little*endian use `"short_swap"` or `"long_swap"`.

`ser_put(`*`addr, s`*`)` — Writes the string *s* to the serial device with address *addr*. Returns the number of bytes written.

`ser_put(`*`addr, d`*`, [, `*`cnt`*`])` — Writes the contents of the data array *d* to the serial device with address *addr*. By default, the entire array (or subarray, if specified) will be sent. The optional third argument *cnt* can be used to specify the number of array elements to send. For short and long integer arrays, the data will be sent using native byte order. The `"swap"` option of the `array_op()` function can be used to change the byte order, if necessary. No format conversions are available for float or double data. Returns the number of bytes written.

`ser_par(`*`addr, cmd`* `[, `*`arg`* `])` — Accesses various features for the given socket with values for *cmd* as follows:

`"device_id"` — Returns the name of the associated serial device or –1 if there is no serial device configured as *addr*.

`"responsive"` — Returns 1 if the associated serial device is open, 0 if the device could not be opened and –1 if there is no serial device configured as *addr*.

`"drain"` — Waits for pending output on the associated serial device to be transmitted, but can be interrupted with **`^C`**. Use the `"flush"` option, described next, to empty the output queue.

`"flush"` — Flushes the input and/or output queues for the serial device with address *addr*. If the optional argument *arg* is zero or absent, the input queue is flushed. If *Otherwise, both queues are flushed. The input queue may contain characters if a* `ser_get()` *times out before the read is satisfied, or if more characters arrive than are requested.*

`"queue"` — Returns the number of characters in the serial device's input queue. The input queue may contain characters if a `ser_get()` times out before the read is satisfied, or if more characters arrive than are requested.

`"timeout"` — Returns or sets the read timeout for the serial device with address *`addr`*. The units for *`arg`* are seconds. A value of zero indicates no timeout – a `ser_get()` will wait until the read is satisfied or is interrupted by a **`^C`**. The smallest allowed value of 0.001 will cause the `ser_get()` to return immediately. A negative value resets the timeout to the default of two seconds.

`"baud"` — Returns or sets the baud rate for the serial device with address *`addr`*. Valid baud rates are from 300 to 4000000. The function returns the device's baud rate. If *`arg`* isn't valid or if there was an error, the function returns –1. Reading the hardware *config* file resets the baud rate to the value in the file. spec cannot set the baud rate on `SOCKET` interfaces.

`"stop_bits"` — Returns or sets the stop-bits value for the serial device with address Normal values are one or two. The default value of one is appropriate for nearly every serial device, and this command should very rarely be needed. Note, to set the non-default value, this command will need to be issued each time after reading the hardware *config* file. This mode is not supported on `SOCKET` interfaces.

`"data_bits"` — Returns or sets the data-bits value for the serial device with address *`addr`*. Accepted values are 5, 6, 7 and 8. The default values of seven if parity is enabled and eight if parity is disabled should work for nearly every serial device, and this command should very rarely be needed. Note, to override the default value, this command needs to be issued after reading the hardware *config*file `reconfig`). This mode is not supported on `SOCKET` interfaces.

`"dtr"` — Returns the current setting or sets or clears the Data Terminal Ready (DTR) control line. Only available on standard serial interfaces. Reset on hardware reconfiguration.

`"rts"` — Returns the current setting or sets or clears the Request To Send (RTS) control line. Only available on standard serial interfaces. Reset on hardware reconfiguration.

`"dsr"` — Returns the current setting of the Data Set Ready (DSR) control line. Only available on standard serial interfaces.

Values for any combination of the parameters `"timeout"`, `"baud"`, `"stop_bits"`, `"data_bits"`, `"dtr"`, and `"rts"` can be set in one call of `ser_par()` by combining assignments in a comma-separated list, as in

```
ser_par(addr, "timeout=1.5,baud=28800,stop_bits=2,data_bits=8")
```

All the `ser_get()` calls will store leftover bytes in a queue. Contents from the queue will be returned on a subsequent `ser_get()` call. Bytes are leftover if the read finishes with a timeout, if more bytes have arrived than are asked for or if more bytes are available after an end-of-string match. Use the `"flush"` option of `ser_par()` to clear the input queue, if needed.

To transfer binary byte streams containing null bytes, use the data-array versions of `ser_get()` and `ser_put()` with byte arrays. Null bytes mark the end of a normal string. Reads a string of characters, up to and including a newline, from serial device *dev_num* and returns the string so read. The routine will not return until the read is satisfied. Use `ser_get(`*dev_num*`, 0)` to read up to a newline with a timeout.

GPIB (IEEE-488) Hardware Functions

GPIB functions are available if the appropriate hardware and software drivers have been installed on the computer. Information in the *config* file describes the particular GPIB hardware in use. Refer to the *Administrator's Guide* for information on the supported GPIB controllers and how to install the corresponding drivers.

spec allows up to four GPIB controllers to be in use at once. The controller unit numbers are set in the *config* file. For the functions below, there are two methods by which the unit number can be specified. If no unit number is specified, the default, unit 0, is used. The first method of addressing is of the form `"unit:addr"` where the quotes are required. Alternatively, the unit number can be coded in the GPIB address as *unit* × 100 + *addr*.

`gpib_cntl(`*addr*`, `*cmd*`)` — Performs the selected GPIB command on the device with address *addr*. The string *cmd* is one of the following:

- `"gtl"` — Go to local.
- `"llo"` — Local lockout.
- `"sdc"` — Selected device clear.
- `"dcl"` — Device clear (sent to all devices).
- `"get"` — Group execute trigger (sent to addressed device).
- `"ifc"` — Interface clear. This command resets the GPIB bus by sending the IFC message. The address *addr* is ignored. spec runs the same code sequence with the `"ifc"` command as it does when it initializes the GPIB controller on start up or on the `reconfig` command. For most controllers, spec sleeps for some fraction of a second after resetting the bus. Also, for most controllers, spec asserts the REN (remote enable)

command after sending IFC.

`"responsive"` — Not a GPIB command, but returns 1 or 0 indicating whether the associated controller is configured and working. Note, the test is on the controller, not the device. To test controllers other than unit 0, the address syntax for *addr* is `"1:1"` or `101` for unit 1, etc. The device address isn't looked at for this option.

`gpib_get(`*addr*`)` — Returns a string from the GPIB device with address *addr*. The device must terminate the string with either a newline (`\n`) or a carriage return and a newline (`\r\n`). In either case, the terminator is removed before the string is returned. At most, 8,192 characters can be read at a time.

`gpib_get(`*addr, n*`)` — As above, but reads *n* bytes and does not look for or remove the terminator.

`gpib_get(`*addr, s*`)` — As above, but tries to read up to the terminator given by the first character of the string *s*, except for the special cases described below. The terminator is removed.

`gpib_get(`*addr, mode*`)` — If *mode* is the string `"byte"`, reads and returns one unsigned binary byte. The following modes read short or long integers and work the same on both *big-endian* and *little-endian* platforms. If *mode* is the string `"int2"` reads two binary bytes and returns the short integer so formed. If *mode* is the string `"int4"` reads four binary bytes and returns the long integer so formed. By default, the incoming data is treated as *big endian*. If the incoming data is *little endian*, use `"int2_swap"` or `"int4_swap"`.

`gpib_put(`*addr, string*`)` — Writes the string *string* to the device with GPIB address *addr*. Returns the number of bytes written. The length of the string is not limited, but null bytes cannot be sent.

`gpib_poll(`*addr*`)` — Returns the serial-poll status from the device with address *addr*.

Returns zero if command is sent successfully, Otherwise returns –1.

VME Hardware Functions

The type of data access and/or VME address modifier for the following functions can be selected with the optional argument *dmode* as follows (if more than one option is needed, make a comma-separated list in the single string argument):

`"D8"` — byte access

`"D16"` — short-word access

`"D32"` — long-word access, but only available with `vme_get32()` and `vme_put32()`.

"DPRT" — use the address modifier appropriate for dual-port memory access, on adapters that support it.

"amod=0x*XX*" — specify the hexadecimal value for the address modifier.

The default mode for the A16 access functions vme_get() and vme_put() is D8. The default mode for the A32 access functions vme_get32() and vme_put32() is D32. Not all VME adapters supported by spec support A32 access. There are currently no functions for A24 access.

vme_get(*addr* [, *dmode*]) — Returns the data at *addr* in the 64K A16 address space.

vme_put(*addr*, *data* [, *dmode*]) — Writes *data* to *addr* in the 64K A16 address space.

vme_get32(*addr* [, *dmode*]) — Returns the data at *addr* in the A32 address space.

vme_put32(*addr*, *data* [, *dmode*]) — Writes *data* to *addr* in the A32 address space.

vme_move(*from*, *to* [, *cnt* [, *dmode*]]) — Copies data between a spec data array and VME A32 address space. One of the *from* and *to* arguments must be the name of a spec data array while the other must be a VME address. If the optional argument *cnt* is present, it designates how many data items (not bytes) to copy. If missing or zero, the number of elements in the array is copied.

PC Port I/O

The port I/O functions allow arbitrary access to I/O ports on a PC computer, and therefore, should be used with caution. To lesson the chance of writing data to a port that might damage the computer or corrupt data, valid addresses for the following functions must be explicitly assigned in the *config* file.

port_get(*addr*) — Reads one byte from the PC I/O port with the address *addr*. Resets to command level if the port has not been selected in the *config* file. Otherwise, returns the byte read.

port_getw(*addr*) — As above, but reads and returns a two-byte value.

port_put(*addr*, *byte*) — Writes the byte *byte* to the PC I/O port with the address *addr*. Resets to command level if the port has not been selected for writing in the *config* file. Otherwise, returns zero.

port_putw(*addr*, *word*) — As above, but writes the two-byte value *word* to the I/O port.

CAMAC functions are available if the appropriate hardware devices and software drivers have been installed on the computer. The *config* file describes which CAMAC hardware is installed. Refer to the *Administrator's Guide* for information on the supported CAMAC controllers and how to install the corresponding drivers.

CAMAC modules are programmed with *FNA* codes where *F* is a function code, *N* is the slot number and *A* is a subaddress number. Slot numbers are assigned in the *config* file. Built-in code for the specialized CAMAC devices used for controlling motors, clocks and scalers is accessed through commands such as `mcount()`, `tcount()` and `move_all`. However, simple devices, such as input or output registers, can be accessed directly by the user. These modules are also assigned slot numbers in the *config* file. They are also given device numbers, starting at 0, that are used in the following functions.

`ca_get(`*`device`*`, `*`address`*`)` — The CAMAC module having device number *`device`*, as set in the *config* file, is read using *F* = 0 and *A* = *`address`* with the 24-bit value so obtained returned. Resets to command level if not configured for *`device`*.

`ca_put(`*`x`*`, `*`device`*`, `*`address`*`)` — This function is similar to `ca_get()` above, except the 24-bit value *`x`* is written using *F* = 16. The actual number written is returned, which is the 24-bit integer representation of *`x`*. Resets to command level if not configured for *`device`*.

`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 the selected CAMAC crate command according to the parameter *`cmd`*, as follows:

`"Z"` or `"init"` — performs a crate initialize (reset).

`"C"` or `"clear"` — performs a crate clear.

`"inhibit"` — set crate inhibit if *`arg`* is 1 and clears crate inhibit if *`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.

.

STANDARD MACRO GUIDE

Introduction

The standard macros included with the spec distribution are an integral part of the spec package. For some sites, the standard macros are sufficient for performing experiments. For others, the standard macros provide a starting point for custom development. This reference presents some of the standard macros, grouped by functionality. For many macros, the complete definition is printed. At the end of this reference, the construction of the scan macros is discussed in depth.

The following files, found in the *macros* subdirectory of the distribution, contain the definitions for all the macros in the standard library. If it is not obvious in which file a particular macro is stored, you can always change to the *macros* directory and type `grep` *`macro_name`* `*`, where *grep* is the standard UNIX file searching utility.

File	Contents
count.mac	Counting macros (`ct`, `uct`, `count`, `show_cnts`, `...`).
cplot.mac	The `cplot_plot` macro.
energy.mac	For an energy-selecting monochromator (`Escan`, `set_E`, `...`).
file.mac	The `newfile` macro.
getscan.mac	The `getscan` macro.
hkl.mac	General reciprocal space macros (`br`, `mk`, `ca`, `wh`, `...`).
motor.mac	Motor moving and status (`mv`, `umv`, `wa`, `set`, `set_lm`, `...`).
plot.mac	Data plotting (`plot`, `plot_res`, `rplot`, `splot`, `ansiplot`, `...`).
powder.mac	Powder-mode macros (`setpowder`, `_pmove` and `_pcount`).
region.mac	Macros to define a series of scans(`setreg` and `doreg`).
scans.mac	Basic scan macros (`ascan`, `a2scan`, `hklscan`, `...`).
scans1.mac	Scan helper macros (`_head`, `_loop`, `setscans`, `...`).
start.mac	The `startup` macro.
temper.mac	Temperature control (`settemp`, `measuretemp`, `te`, `...`).
util.mac	Misc. utility macros (`do`, `qdo`, `savmac`, `comment`, `u`, `help`, `...`).
fivec.src	Fivec-circle geometry macros.
fourc.src	Four-circle geometry macros.
sixc.src	Sixc-circle geometry macros.
spec.src	Version for no diffractometer.
surf.src	Special liquid surface diffractometer macros.
twoc.src	Two-circle geometry macros.

zaxis.src Z-axis geometry macros.

When installed, the *.mac* files above are combined into one file and placed (assuming the default auxiliary file directory) in */usr/local/lib/spec.d/standard.mac*. A file formed from the first four letters of the geometry configuration contains the geometry macros from one of the *.src* files above. For example, */usr/lib/spec.d/four.mac* is created for the four-circle geometry and contains the macros from *fourc.src*.

The macro definitions are the least stable part of the spec package. The macros are easy to change — no recompilation of C code is necessary — and the intent of the designers of the spec package was to put its flexibility in the macros. Thus, the definitions presented below may differ with the macro definitions in your current version of spec.

You may find the existing macros do not accomplish what you want. A simple procedure for modifying a standard macro is to use the macro `savmac` to copy the definition of the existing macro into a file. You then edit the macro definition in that file and read it back in using the `do` macro. You can gather your customized macros into a file named *spec.mac* in your current directory or into the file */usr/lib/spec.d/site.mac*. Both of these files are read every time you start spec, whether or not you are starting fresh.

If you have made generally useful modifications to the standard macros, or if you have developed your own macros, please send copies to Certified Scientific Software. We include many user-contributed macros in each new release of the spec package.

Some Tips

The syntax rules for defining macros are given in the *Reference Manual* on page 101. The suggestions that follow offer some additional guidance for writing macros that will fit in well with the standard library.

When a macro requires arguments, it is a good idea to check that the right number of arguments have been given, and if not, print a usage message and exit to command level. The symbol `$#` will be set to the number of arguments when the macro is run. For example,

```
def ascan '
      if ($# != 5) {
              print "Usage:  ascan  motor start finish  intervals time"
              exit
      }
      ...
'
```

If an argument is supposed to be a motor number or mnemonic, use the _check0 macro before operating on the motor. The _check0 macro exits to command level if the argument is not a valid mnemonic. For instance, to check the first argument of a macro, use

```
_check0 "$1"
```

A mistyped mnemonic might otherwise become a variable with an arbitrary value (zero for a new variable) resulting in an operation on the wrong motor (usually motor zero).

It is good practice to refer to arguments just once when writing macros to avoid side effects that occur, for example, if the macro is invoked as mymac i++. Here the variable i would be incremented each time $1 is used in the macro. In the scan macros, the arguments are assigned to global variables just after the usage check:

```
def ascan '
      ...
      { _m1 = $1; _s1 = $2; _f1 = $3; _n1 = int($4); _ctime = $5 }
      ...
'
```

When a macro changes a parameter or mode that affects later data, it is a good idea to note that change in the data file and on the printer. Macros such as comment, qcomment and gpset are available for that purpose.

If possible, declare local variables local to avoid conflicts with other variables, especially when macros are nested or parsed together.

Watch out for name conflicts when naming new macros and using variables. You can prevent most conflicts by using the local keyword to explicitly declare local names within a statement block. Names declared that way can be used as symbols within the statement block even if they are already in use as macros. Otherwise, if you construct commands using a variable name that is really a macro name, when that intended variable is encountered, it will be replaced by the macro, making a mess of things.

Note that several one-letter names such as d, h, p and l are already in use as macro names. Don't use these names as variables, unless they are declared local inside a statement block. Typing lsdef ? will list all one letter macro names. Typing lsdef _? will list all two letter macro names that begin with an underscore.

Command files that define macros often assign default values to related global variables. You should always check if these global variables have already had a value assigned before assigning default values. If the user had assigned a new value to a variable, you do not want that assignment undone if the macro file is reread. The built-in whatis() function can be used to see if a variable has been assigned a value

(see page 82 for an explanation of the whatis() return values),

```
if ((whatis("DATAFILE")>>16)&0x0800) {
      print "Warning:  No open data file.  Using \"/dev/null\".\n"
      open(DATAFILE = "/dev/null")
}
```

When writing macros that move motors, be careful with the move_all command. When moving motors, always do a waitmove and getangles first. Then assign new values to A[], and finally call move_all (or move_em).

When obtaining input from the user, the functions getval() and yesno() are useful. For example,

```
_update = yesno("Show updated moving and counting", _update)
g_mode = getval("Geometry mode", g_mode)
```

results in the following screen output:

```
Show updated moving and counting (NO)?
Geometry mode (3)?
```

You can also use the input() built-in function to obtain user input. Remember, though, that input() returns a string. If the string contains a valid number, the automatic string-to-number conversion will take place, if context requires it. However, no expression simplification is done on the string, so a response of 2+2 will not have a number value of 4 when returned by input().

When using on() and off() to control output, do the operations on "tty" last. Since "tty" is always turned back on if everything else is turned off, the commands

```
off("tty");on(PRINTER);print "hello world";on("tty");off(PRINTER)
```

will not have the desired effect. The first off() turns off everything, so "tty" is automatically turned back on, and the message goes to both PRINTER and "tty".

Use existing UNIX utilities if they can be of help. For example, if you manipulate UNIX file names in your macros you can use the return value of the *test* utility to check for existence of a file. For example, the function unix("test -r $1") will return zero if the file specified by the argument exists and is readable.

Utility Macros

UNIX Commands

These simple macros are for commonly used UNIX commands.

```
def cd    'chdir("$*")'          # Change working directory
def pwd   'print CWD'            # Print working directory
def u     'unix("$*")'           # Execute arbitrary shell commands
def ls    'unix("ls $*")'        # List files
def l     'unix("ls -l $*")'     # Long listing of files
def cat   'unix("cat $*")'       # Show file contents
def less  'unix("less $*")'      # Peruse files with handy utility
def mail  'unix(sprintf("%s $*", MAIL))' # Send mail
def ed    'unix("ed $*")'        # Invoke an editor
def ned   'unix("ned $*")'       # Invoke another editor
def vi    'unix("vi $*")'        # Invoke another editor
```

The u macro, without arguments, spawns an interactive subshell, using your *SHELL* environment variable.

Note how the above macros supply parentheses and quotation marks around the arguments, as required by the parser's grammar rules.

Basic Aliases

The main purpose of these macros is to provide a shorthand way of typing some useful commands.

```
def d         'print date()'        # Print current time and date
def p         'print'               # Shorthand for print
def h         'help'                # Shorthand for help, below
def hi        'history'             # Shorthand for history
def beep      'printf("\a")'        # Sound the bell
def cl        'tty_cntl("cl")'      # Clear the screen
def com       'comment "$*"'        # Shorthand for comment, below
def ond       'if (DATAFILE)        # Send output to data file
              { on(DATAFILE) }'
def offd      'if (DATAFILE)        # Stop sending
              { off(DATAFILE)'
def onp       'if (PRINTER)         # Send output to the printer
              { on(PRINTER) }'
def offp      'if (PRINTER != "")   # Stop sending
              { off(PRINTER) }'
def ont       'on("tty")'           # Send output to the terminal
def offt      'off("tty")'          # Stop sending
def fon       'if ($# == 1) on("$1")
               else { print "Usage: fon filename"; on(); }'
def foff      'if ($# == 1) off("$1")
```

```
                else { print "Usage: foff filename";on(); }'
def waitall   '{ user_waitall; wait(0) }'   # Wait for async activity
def waitmove  '{ user_waitmove; wait(1) }'  # Wait for moving to end
def waitcount '{ user_waitcount; wait(2) }' # Wait for counting to end
def chk_move  '(wait(0x21) || USER_CHK_MOVE)'
def chk_count '(wait(0x22) || USER_CHK_COUNT)'
def chk_acq   '(wait(0x24) || USER_CHK_ACQ)'
def w         '{ waitall; beep }'   # Wait, and be audible when done
```

Basic Utility Macros

These straightforward macros combine a number of built-in functions and commands to provide a higher level of functionality with minimal input. First, here is their usage:

```
help [topic]                        # Display help files
config                              # Edit hardware configuration
onsim                               # Turn on simulate mode
offsim                              # Turn off simulate mode
debug [value]                       # Select debugging categories
bug                                 # Mail a bug report
whats object                        # Identify the object
gpset variable value                # Comment if a variable has changed
```

Here are the definitions for some:

```
# Examine help file, use default if no argument.
def help  '
      if ($#)
          gethelp("$1");
      else {
          local t
          for (t="help";;)
              if (gethelp(t) || (t = input("\nSubject?  ")) == "")
                  break
      }
'
```

```
# View (and modify), then reread configuration file.
# Use -s flag if in simulate mode.  Re-order motor numbers
# with _assign.  Check for monochromator mnemonics.
def config ’
      wait(-1)
      user_waitall
      sync
      unix(sprintf("%s/edconf %s %s/%s",\
              SPECD, set_sim(-1)? "-s":"", SPECD, SPEC))
      reconfig
      user_config
      _assign
      _assign_mono
’
def user_config ’’

# Turn simulate mode on.  Comment on printer and file if changed.
def onsim ’{
      local t

      if (!(t = set_sim(1))) { qcomment "Simulate mode ON" }
      printf("Simulate was %s, is now %s.\n", t? "on":"off",\
                                      set_sim(-1)? "ON":"OFF")
}’
# Turn simulate mode off.
def offsim ’{
      local t

      if (t = set_sim(0)) { qcomment "Simulate mode OFF" }
      printf("Simulate was %s, is now %s.\n", t? "on":"off",\
                                      set_sim(-1)? "ON":"OFF")
}’

# Easy way to set the debug level.
# +arg adds bits to DEBUG.  -arg removes them.
def debug ’{
      local t

      if ($# == 0) {
              gethelp("debug")
              t = input(sprintf("\nDebug value (%d)? ", DEBUG))
      } else
              t = "$*"
      if (index(t, "+"))      DEBUG |= 0+t
      else if (index(t, "-")) DEBUG &= ~(0-t)
      else                    DEBUG  = 0+t
}’
```

```
# Send a bug report to the administrator.
def bug ’
      print "The mail utility will be run for you.  Describe your"
      print "problem to the administrator.  When you are done, type ^D."
      {
        local s
        s = unix(sprintf("%s -s \"Bug from %s\" %s", MAIL, USER, ADMIN))
        printf("Bug report %ssent to %s.", s? "not ":"", ADMIN)
      }
’


# Set something and comment if it has changed.
def gpset ’
      if ($1 != $2) {
              comment "$2 reset from %g to %g" "$2,$1"
              $2 = $1
      }
’
```

Reading From Command Files

```
do command_file                    # Run a command file
qdo command_file                   # Run a command file without echo
newmac                             # Reread standard command files
```

Since the do and qdo macros have nearly identical functionality, the commands for both are placed in a single macro named _do. This macro implements special functions, such as letting a dot stand for the previous command file or searching for a command file first in the current directory and then in a special command file directory.

```
# "do" a command file.
def do  ’_do $* do’

# Quietly "do" a command file.
def qdo ’_do $* qdo’

# Run a command file.  If not in current directory, look for
# file in DO_DIR.  Save file name in DOFILE so "do ." works.
def _do ’
      if ($# != 2 || ("$2" != "do" && "$2" != "qdo")) {
              print "Usage:  do file"
              print "       qdo file"
              exit
      }
      if ("$1" == "." && DOFILE == "") {
              print "No previous do file."
              exit
      }
```

```
        if ("$1" != ".") {
                DOFILE = "$1"
                if (DO_DIR != "." && unix(sprintf("test -r %s", DOFILE))) {
                        local t

                        t=sprintf("%s/%s", DO_DIR, DOFILE)
                        if (!unix(sprintf("test -r %s", t)))
                                DOFILE = t
                }
        }
        if (!unix(sprintf("test -r %s", DOFILE))) {
                qcomment "do %s" DOFILE
                $2file(DOFILE)
        } else {
                printf("Can\'t read command file \"%s\".\n", DOFILE)
                exit
        }
,
```

The newmac macro rereads the standard macro files that reside in SPECD the auxiliary file directory (usually */usr/lib/spec.d*). Invoking newmac is useful if a new version of the standard macros has been installed but you do not want to start fresh, or if you have somehow corrupted the definition of a standard macro and want to get back the original definition.

Saving To Output Devices

```
comment format args                 # Send a comment to output devices
qcomment format args                # Send comment to file and printer
prcmd command                       # Print the output of a command
savcmd command file                 # Save a command to a file
savmac macro_name file              # Save a macro to a file

# Print a comment on the screen, printer and data file.
def comment ’
      printf("\n%s.  $1.\n", date(), $2)
      qcomment "$1" "$2"
’
# Print a comment on the printer and data file.
def qcomment ’
      if (PRINTER != "")
              fprintf(PRINTER,"\n%s.  $1.\n", date(), $2)
      if (DATAFILE != "")
              fprintf(DATAFILE, "#C %s.  $1.\n", date(), $2)
’


# Have output of any command sent to the printer.
# Commands are all on one line to avoid outputting prompts.
def prcmd ’onp;offt;printf("\n%s\n","$*");ont;$*
offp’

# Have output of any command sent to a file.
def savcmd ’
      if ($# != 2) {
              print "Usage:  savcmd command filename"
              exit
      }
      on("$2");offt;printf("\n%s\n","$1");ont;$1
      close("$2")
’


# Save a macro definition to a file.
def savmac ’
      if ($# != 2) {
              print "Usage:  savmac macro_name filename"
              exit
      }
      on("$2"); offt
      prdef $1
      ont; close("$2")
’
```

Start-up Macros

These macros ask for all the initialization information that may be needed by the other macros.

```
startup                 # Sets things up to start experiment
newsample               # Gets title and data file for experiment
newfile [name [num]]    # Sets up a data file
starttemp               # Asks for temperature control parameters
setscans                # Asks for scan options
setplot                 # Asks for plotting options
startgeo                # Queries user for all geometry parameters
save [file]             # Saves important variables to a file
savegeo                 # Used by "save", saves geometry parameters
saveusr                 # Used by "save", user defined
```

The `startup` macro prompts for values for a number of global variables and also invokes all the other initialization macros, leading to more than a screenful of questions. You can, however, always jump back to command level by typing the interrupt character (**^C**) if you do not need to change items at the bottom of the list. Be careful, though, since some of the initialization macros, (`setplot`, for example) don't save the entered information until all their questions are answered.

```
def startup '
      printf("\n(newsample)")
      newsample
      {
       local t
       t = PRINTER != "" && PRINTER != "/dev/null"
       if (yesno("\nUse a printer for scan output", t)) {
              PRINTER = getval("Printer device",PRINTER)
              if (index(PRINTER,"/")==0)
                     PRINTER = sprintf("/dev/%s",PRINTER)
              if (open(PRINTER))
                     PRINTER = "/dev/null"
       } else
              PRINTER = "/dev/null"
      }
      if (substr(PRINTER,1,5) != "/dev/")
              PRINTER = sprintf("/dev/%s",PRINTER)
      if (open(PRINTER))
              PRINTER = "/dev/null"
      newfile
      DO_DIR = getval("\nCommand file directory",DO_DIR)
      COUNT = getval("Default count time for ct and uct",COUNT)
      UPDATE = getval("Update interval for umv, uct, etc. in seconds",UPDATE)
      if (whatis("starttemp")>>16)
              printf("0arttemp)")
      starttemp
```

```
        setscans
        setplot
        startgeo
'
```

In the standard distribution, `starttemp` has a null definition.

The `newfile` macro creates, opens or reopens standard spec data files. The filename and scan number may be given as arguments. Otherwise, you are asked for the information. If you have a directory named *data* in you current directory, and there are no / characters in the file name you give, the data file will be placed in the *data* directory. If the file already exists, new scans will be appended to the file. The existing file is not removed.

The `startgeo` macro is defined differently for various geometries, but should query the user for values for all the relevant geometry parameters.

The `save` macro is not really an initialization macro, but it creates a file that can be used for initialization. The purpose of the macro is to save all the important global variables in a file that can be run as a command file at a later time to restore the values of those variables. For example, if the user anticipates starting fresh with a new version of the software, having a file created by the `save` macro will simplify creating a new program state.

```
# Save current globals to a save file
def save '{
        local f

        if ($# == 0)
                f = getval("File for saving globals", "saved")
        else if ($# == 1)
                f = "$1"
        else {
                print "Usage:  save [filename]"
                exit
        }
        unix(sprintf("if test -s %s ; then mv %s %s.bak ; fi", f, f, f))
        on(f); ofti
        printf("PRINTER=\"%s\"\n",PRINTER)
        savegeo
        saveusr
        ont; close(f)
        qcomment "Globals saved in \"%s\"" "f"
        printf("Type \"do %s\" to recover.\n", f)
}'
```

The macro `savegeo` saves all the geometry parameters for the particular configuration. You can define the macro `saveusr` to save whatever else is desired.

Motor Macros

```
mv motor pos            # Move a motor
mvr motor pos           # Move a motor, relatively
mvd motor dial_pos      # Move a motor to a dial position
tw motor inc            # Tweak a motor, interactively
umv motor pos           # Move while updating screen
umvr motor pos          # Move while updating screen
wa                      # Show positions of all motors
lm                      # Show limits of all motors
wm m1 m2 ...            # Show positions and limits of motors
uwm m1 m2 ...           # Show positions while motors are moving
set motor pos           # Set user angle for a motor
set_dial motor pos      # Set dial angle for a motor
set_lm motor low high   # Set user limits for a motor
an tth_pos th_pos       # Move two theta and theta
pl chi_pos phi_pos      # Move chi and phi (four-circle)
uan tth_pos th_pos      # Move while updating screen
upl chi_pos phi_pos     # Move while updating screen
```

The following macro moves a single motor, adding a comment to the printer that the motor was moved:

```
# Move a single motor
def mv  '_mv $*; move_poll'
def umv '_mv $*; _update1 $1'   # "update" version of mv
def _mv '
      if ($# != 2) {
              print "Usage:  mv motor position"
              exit
      }
      _check0 "$1"
      waitmove; getangles; A[$1]=$2
      if (PRINTER != "")
              fprintf(PRINTER,"\nmv $1 %g\n", A[$1])
      move_em
'
```

In `mv`, as in all the macros that move motors, the `move_em` macro is invoked, rather than the `move_all` command. Normally, `move_em` is defined as

```
def move_em '
      user_premove
      move_all
      user_postmove
```

One can define the `user_premove` and/or `user_postmove` macros to take into account special conditions. For example, to check for limits that depend on the relative position of motors, one could define `user_premove` as

```
def user_premove ’
      if (fabs(A[tth] - A[th]) > 10) {
              print "Move exceeds Theta - Two Theta relative limit."
              exit
      }
      move_all
’
```

The `set` macro changes the offset between user and dial units.

```
# Define a new motor position
def set ’
      if ($# != 2) {
              print "Usage:  set motor new_user_value"
              exit
      }
      {
        local old
        _check0 "$1"
        waitmove; getangles
        old = A[$1]
        if (chg_offset($1, $2))
          exit
        getangles
        if (old != A[$1]) {
          comment "%s reset from %g to %g" "motor_name($1), old, A[$1]"
        } else
          print "No change."
      }
’
```

The `set_dial` macro changes the dial position of the motor, which means a change to the contents of the motor controller register. `set_dial` refuses to set the dial beyond the current software limits for the motor. `set_dial` also changes the offset to maintain the prior value of the user angle. These two macros document the change in the data file and on the printer.

The `set_lm` macro converts the user-unit arguments to dial units for the call to `set_lim()`.

```
Change a motor limit
def set_lm '
      if ($# != 3) {
              print "Usage:  set_lm motor low high"
              exit
      }
      {
          _check0 "$1"
          if (!set_lim($1, dial($1, $2), dial($1, $3))) {
              onp
              printf("\n%s limits set to %g %g (dial units).\n",\
                      motor_name($1), get_lim($1, -1), get_lim($1, +1))
              offp
          }
      }
'
```

The macros in the above list that begin with a u continuously read motor positions from the controller and show the positions on the screen. The frequency of screen updates is set by the global variable UPDATE, which is used as an argument to the sleep() function. Setting UPDATE=.25 places a 1/4 second pause between updates. The umv macro first calls _mv and then calls the internal _update1 macro. The other updated-move macros are defined similarly.

```
def umv _'mv $*; _update1 $1 '   # "update" version of mv

# Displays updated position of 1 motor while it is moving
def _update1 '
      if (chk_move)) {
              printf("\n%10.9s\n", motor_name($1))
              while (wait(0x22)) {
                      getangles
                      printf("%10.4f\r", A[$1])
                      sleep(UPDATE)
              }
              getangles
              printf("%10.4f\n", A[$1])
      }
'
```

The technique for displaying status information about all the motors is a little complicated. spec places no restriction on what order the motors are assigned to the controller, but does recognize that there is a preferred order for displaying motor information. To this end, the macros use an array mA[] which contains reordered motor numbers. The four-circle macro source file contains the following code, which is executed when the command file is read and when the config macro is run.

```
# Conventionally, the first four motors are tth, th, chi, phi.
# The following code guarantees this.
def _assign '{
      local   i j
      mA[0]=tth
      mA[1]=th
      mA[2]=chi
      mA[3]=phi
      for (i = 4, j = 0; i < MOTORS; j++) {
              if (j == tth || j == th || j == chi || j == phi)
                      continue
              mA[i++] = j
      }
}'
```

Similar code is contained in the macro source files for the other geometries.

An internal macro named _mo_loop exists to loop through all the motors printing selected fields. Its use is best illustrated by example. First here is its definition:

```
# Looping routine used in many macros.
# Normally k is set to MOTORS, but can be set to something else, e.g., 4
# (Kludge with printf(" ") avoids auto linefeed on 80th column.)
def _mo_loop '{
      local s
      for (j = i; j < i + 8 && j < k; j++)
              if (motor_name(mA[j]) != "unused") {
                      s = s sprintf("%$1", $2)
                      if (j < i + 7)
                              s = s " "
              }
      print s
}'
```

It is within this macro that motors named unused are not used in printing motor information.

The wa macro that displays information for all motors is typical of a macro that calls the _mo_loop macro.

```
# Where - all motors
def wa ’
      waitmove; get_angles
      onp
      printf("\nCurrent Positions  (user, dial)\n")
      {
              local i j k
              for (i = 0, k = MOTORS; i < k; i += 8) {
                      _mo_loop 9.9s "motor_name(mA[j])"
                      _mo_loop 9.9s "motor_mne(mA[j])"
                      _mo_loop 9.4f "A[mA[j]]"
                      _mo_loop 9.4f "dial(mA[j], A[mA[j]])"
              }
      }
      offp
’
```

The first argument for `_mo_loop` is a `printf()` field specification, the second argument is the field value. The field values use the `mA[]` array to reorder the motor numbers.

Counting Macros

```
ct [time]               # Count, then show_cnts
count [time]            # Count for time
show_cnts               # Count, then display results
uct [time]              # Updated counting
```

When *time* is positive, counting is to seconds. When *time* is negative, counting is to monitor counts. If the counting macros are invoked without an argument, the count time used is that contained in the global variable COUNT.

Counting in spec combines timing generators and scalers. Three scaler channels are normally used in the standard macros. The first channel takes an accurate clock input (normally at 1 KHz), the second takes the monitor input, and the third takes the detector.

The scaler channels identifying the various inputs are determined by the values of the global constants, sec, mon and det. Their default values are 0, 1 and 2, respectively, matching the recommended hardware cabling. If you do not connect your counting sources to the default scaler channels, you must explicitly assign new values to sec, mon and det for the standard counting macros to work properly.

The additional global variables MON and DET are then set to the channels that are to be treated as the monitor and detector for particular scans, normally mon and det. If, while doing a line-up scan of a motor on which the monitor is mounted, you would want the monitor counts to be plotted as a function of the motor position, enter DET=mon before doing the scan. Do not forget to reassign DET=det at the end of the scan.

The global variable COUNT is set to a default count time (normally 1 second). If the macro ct is invoked without arguments, counting will last for the duration given by COUNT.

```
# A user calls "ct" to count for some interval and display results
def ct '{
      rdef cleanup \'
              undef cleanup
              onp; show_cnts; offp
      \'
      waitmove
      count_em $*
      waitcount
      undef cleanup
      onp; show_cnts; offp
}'
```

```
# "count" is the basic macro to count to monitor or to time.
# It runs the clock and reads the scalers.
def count '{
      waitmove
      if ($1) for (;;) {
              count_em $1
              waitcount
              get_counts
              chk_beam
      }
      if (S[sec] && MON >= 0)
              MON_RATE = S[MON]/S[sec]
}'
# The macro "show_cnts" reads the scalers and displays the results.
def show_cnts '{
      local   i

      get_counts
      printf("\n%s\n\n", date())
      for (i=0;i<COUNTERS;i++)
              if (cnt_name(i) != "unused")
                      printf("%12s = %g%s\n", cnt_name(i), S[i], \
                 i != sec && S[sec]? sprintf(" (%g/s)", S[i] / S[sec]):"")
}'
```

Updated counting is done with the `uct` macro,

```
def uct '{
      waitmove
      count_em $*
      if (chk_count) {
              local i
              printf("\n")
              for (i=0;i<COUNTERS;i++)
                      if (cnt_name(i) != "unused")
                              printf("%12.12s ",cnt_name(i))
              printf("\n")
              while (chk_count) {
                      get_counts
                      for (i=0;i<COUNTERS;i++)
                              if (cnt_name(i) != "unused")
                                      printf("%12g ",S[i])
                      printf("\r")
                      sleep(UPDATE)
              }
              get_counts
              for (i=0;i<COUNTERS;i++)
                      if (cnt_name(i) != "unused")
                              printf("%12g ",S[i])
              printf("\n")
      }
}'
```

Plotting Macros

At present, spec's plotting is done entirely at the macro level and does only character plots on the screen and printer. Several screen plotting macros are defined for various types of terminals.

```
setplot [mode]          # Select plotting options
plot                    # Plot data on printer
rplot                   # Plot updated data at each point of scan
splot                   # Plot data on screen
pts                     # List current data on the screen
lp_plot                 # Primitive 132-column wide plot for printers
plot_res                # Show results after scans
splot_res               # Show results on screen plot
rplot_res               # Show results on updated plot during scans
```

The `setplot` macro defines the `plot` macro, depending on your choices of plot modes.

The `scan_plot` macro is invoked within the looping portion of all the scans.

The `setplot` macro assigns values to the global variable, `PLOT_MODE`, according to the values defined in the following table:

Bit Value	Description
1	Do updated plotting during scans.
2	Do screen plot after scan.
4	Do printer plot after scan.
8	Scale *x*-axis of screen plots to fit width of scan.
16	Force *y*-axis minimum to be zero.
32	Use logarithmic *y*-axis.
64	Do simple background subtraction
128	Use high-resolution plotting device
256	With high-res, don't use large dots
512	With high-res, don't connect points with lines
1024	With high-res, don't draw error bars

The `scan_plot` macro is called for each point of a scan, while `plot` is called at the end of each scan.

The `splot` macro draws a screen plot. The `rplot` macro is called to redraw the plot with minimal updating during data accumulation.

Reciprocal Space Macros

The following macros are general and applicable to most geometry configurations. Macros special to the four-circle geometry configuration are described in the *Four-Circle Reference* that follows this guide.

```
ca H K L                # Display calculated positions for H K L
cal H K L               # As above, but don't reset positions
ci tth th chi phi       # Display calculated H K L for angles
br H K L                # Move to H K L
mk H K L                # Move to H K L
ubr H K L               # Move to H K L while updating screen
umk H K L               # Move to H K L while updating screen
mi ALPHA BETA           # Move to ALPHA BETA
wh                      # Display H, K, L, tth, th, chi, phi, etc.
pa                      # Display geometry parameters
```

The difference between `ca` and `cal` is that the first macro restores the `A[]` angles and `H`, `K` and `L` to the current diffractometer position, while the second macro leaves them at the calculated values.

There is no difference between the `br` and `mk` macros, except their names.

The `ubr` and `umk` macros continuously read the motor positions from the controller and show the positions on the screen. The frequency of updates is set by the global variable `UPDATE`.

```
# Go to a Bragg position
def br '_br $*; move_poll'
def _br '
      if ($# != 3) {
               print "Usage:  br H K L"
               exit
       }
       waitmove; { H=$1; K=$2; L=$3 } getangles; calcA
       onp; offt; printf("\nbr %g %g %g\n", H, K, L); offp; ont
       move_em
'
```

```
# Calculate motor positions for a given H, K, and L
def cal ’
      if ($# != 3) {
              print "Usage:  cal H K L"
              exit
      } ;
      {H = $1; K = $2; L = $3 } calcA; calcHKL
      onp
      printf("\nCalculated Positions:\n")
      _var
      offp
’
# As above but reset positions to diffractometer positions
def ca ’
      if ($# != 3) {
              print "Usage:  ca H K L"
              exit
      } ;
      {H = $1; K = $2; L = $3 } calcA; calcHKL
      onp
      printf("\nCalculated Positions:\n")
      _var
      offp
      waitmove; getangles; calcHKL
’

# Where - reciprocal and real space
def wh ’
      waitmove; getangles; calcHKL
      onp
      _var
      offp
’

# A macro called by "wh", "ca" and "ci" to display important
# geometry quantities.  (Four-circle version.)
def _var ’
        printf("\nH K L =  %.5g  %.5g  %.5g\n", H, K, L)
        printf("ALPHA = %.5g  BETA = %.5g", ALPHA, BETA)
        printf("  AZIMUTH = %.5g  LAMBDA = %g\n\n", AZIMUTH, LAMBDA)
        _mot 4
’
```

Scan Macros

The following sections summarize the usage of the standard scans in spec. This discussion is followed by a description of macros to customize the scan output sent to the printer and data file. Refer to page184 for a detailed discussion of a single-motor scan.

All the scans use the same basic invocation syntax. For example, the single motor, absolute-position scan is invoked as

```
ascan  motor  start  finish  intervals  time
```

The range of a scan is specified by the starting position *start*, the final position *finish* and the number of intervals *intervals* of the scanned parameters. Thus

```
ascan  tth  26  28  20  60
```

would start with the tth motor at 26°, and move the motor to 28°, using 20 intervals of 0.1°. The total number of points scanned is one more than the number of intervals, in this case, 21 points. The time per point *time*, if a positive number, indicates counting to seconds. In the above example, each point takes 60 seconds. Using a negative time indicates counting to monitor counts.

Scan Miscellany

```
resume                          # Restart an aborted scan
setscans                        # Set scan-mode options
```

If a scan is halted by typing the interrupt character (**^C**), or because of some other error such as hitting a motor limit, you can normally restart the scan by typing resume. You must not have changed the value of any of the internal scan variables in the meantime. If you use resume to continue a relative position scan, such as lup or dscan, the scanned motors will not be returned to the center point when the scan ends, as they otherwise would be.

You also have the option to control how the scan data is displayed on the screen as it is taken. An option to the setscans macro allows the motor positions and scalers contents to be displayed while they are changing during a scan. The rate of updates is set by the UPDATE global variable, just as with the umv and uct macros. An option to the setplot macro, presented earlier, allows real-time plots of the data points to be displayed as they are measured.

Another option selected in the setscans macro lets you choose whether to do prescan motor limit checks with reciprocal space scans. The purpose of these checks is to avoid running into a software motor limit in the middle of a scan. For regular motor

scans, the limit checks are done only at the scan endpoints. For reciprocal space scans, the motor positions do not necessarily change monotonically and so the motor limits must be checked at each scan point. Since this checking requires a time-consuming loop at the macro level, you may choose to disable the feature if you are confident your reciprocal space scans will not send a motor outside the ranges defined by software limits.

Motor Scans

```
ascan motor start finish  intervals time
a2scan m1 s1 f1  m2 s2 f2  intervals time
a3scan m1 s1 f1  m2 s2 f2  m3 s3 f3  intervals time
mesh m1 s1 f1  intervals1  m2 s2 f2  intervals2 time
lup motor start finish intervals time
dscan motor start finish intervals time
d2scan m1 s1 f1  m2 s2 f2  intervals time
d3scan m1 s1 f1  m2 s2 f2  m3 s3 f3  intervals time
th2th tth_start_rel tth_finish_rel intervals time
```

The argument *motor* (and *m1*, *m2* and *m3*) is a motor number or mnemonic, such as th, tth, chi, or phi. ascan, a2scan and a3scan are single-, two- and three-motor absolute-position scans. mesh is a nested two-motor scan, where the first motor scans through its range at each point of the second motor's scan. lup (or equivalently dscan), d2scan and d3scan are single- and two- and three-motor relative position scans. The starting and finishing positions are given relative to the current position, and the motors are returned to their starting position at the end of the scan. These relative position scans are defined in terms of the absolute-position scans.

The th2th macro is a special case of the d2scan that will scan the tth and th motors, with th moving half the range of tth. Its definition is,

```
def th2th '
      if ($# != 4) {
          print "Usage:  th2th tth_start_rel tth_finish_rel intervals time"
          exit
      }
      d2scan tth $1 $2 th ($1)/2 ($2)/2 $3 $4
'
```

Basic Reciprocal Space Scans

```
hscan start_H finish_H intervals time
kscan start_K finish_K intervals time
lscan start_L finish_L intervals time
hklscan s_H f_H  s_K f_K  s_L f_L  intervals time
hklmesh Q1 s_Q1 f_Q1 intervals1  Q2 s_Q2 f_Q2  intervals2 time
```

The first three scans are special cases of `hklscan`, as in

```
def hscan '
      if ($# != 4) {
              print "Usage:  hscan start finish intervals time"
              exit
      }
      waitall; getangles; calcHKL
      hklscan $1 $2 K K L L $3 $4
'
```

`hklmesh` does a grid scan of two reciprocal coordinates, where *Q1* and *Q2* are literally `H`, `K` or `L`, and *Q1* ≠ *Q2*.

A `waitmove`, `getangles` and `calcHKL` are done at the start of the scan to obtain the current diffractometer position to determine the values of the unspecified coordinates in `hscan`, `kscan`, `lscan` and `hklmesh`.

Special Reciprocal Space Scans

```
klradial angle start_radius finish_radius intervals time [H=expr]
hlradial angle start_radius finish_radius intervals time [K=expr]
hkradial angle start_radius finish_radius intervals time [L=expr]
klcircle radius start_angle finish_angle intervals time [H=expr]
hlcircle radius start_angle finish_angle intervals time [K=expr]
hkcircle radius start_angle finish_angle intervals time [L=expr]
```

The first three scans trace a path that would form a radial cut if projected on to the *K-L*, *H-L* or *H-K* planes at the specified angle in degrees from the positive *K*, *H* or *H* axes, respectively. *start_radius* and *finish_radius* specify the radial distance from the origin. The unscanned coordinate will be reevaluated at each point according to the optional expression in the last argument, which can be a function of the other coordinates, for example, `H=L/300`. Otherwise, the unscanned coordinate will remain constant.

The second three scans differ only in that they trace out a circular arc in the projected plane at the radius given by the first argument. *start_angle* and *finish_angle* are the endpoints in degrees of the arc of the scan.

Temperature Scans

```
# Temperature scan
tscan start finish intervals time [sleep]
# Delta temperature scan
dtscan start finish intervals time [sleep]
```

These two macros scan the temperature setpoint. The macro `settemp` (see the temperature control macros, described below) is called to change the setpoint. A call of the `sleep()` function is done after calling `settemp`, but before counting, if the optional argument *`sleep`* is greater than zero.

Powder Mode

```
setpowder                       # Select "powder" mode
setpowder off                   # Turn "powder" mode off
setpowder motor full_width      # Specify powder motor and width
```

Powder mode enables you to measure intensities while a motor is scanned through some range. When turned on, powder mode affects all scans where motors are moved. If invoked without arguments `setpowder` prompts for the powder motor and for the full width of the rocking movement to take place at each point of the scan. Invoked with the `off` argument, scans return to their normal mode.

Customizing Scan Output

To allow you to customize the scan headers and the information saved with each data point, several macros are available for redefinition. Their default definitions are:

```
def Pheader ''          # Printer header
def Fheader ''          # File header
def Plabel  '""'        # Printer/Video column labels
def Pout    '""'        # Printer/Video output at each point
def Flabel  '""'        # File column labels
def Fout    '""'        # File output at each point
```

Four of these must be defined as strings — in the default case, null strings. Here are examples of how these macros might save temperature set point and measurement information on the output devices.

```
def Pheader 'printf("\n Setpoint = %g (%g C)\n", TEMP_SP,DEGC_SP)'
def Fheader '_cols++;printf("#X %gKohm (%gC)\n",TEMP_SP,DEGC_SP)'
def Plabel  'sprintf("%7.7s %7.7s ","T-set","T-degC")'
def Pout    'sprintf("%7.5g %7.5g ",TEMP_SP,DEGC)'
def Flabel  '"DegC  "'
def Fout    'sprintf("%g ",DEGC)'
```

The `Pheader` and `Fheader` macros must print newline-terminated lines. More than one line is permitted, however. Besides adding scan header information to the data file, `Fheader` also adjusts the value of the global variable `_cols`, which is used to indicate the number of data columns in the data file. In the example shown, the `Flabel` and `Fout` definitions add one column to the data file, so `_cols` is incremented by one. The `Plabel` and `Pout` macros add columns to the printer (and screen) output. The columns in `Flabel` should be separated by double spaces (the data file convention). The columns in the other headers should be separated by single spaces. In each case, the spaces come after the label.

The `Ftail` macro is available for adding scan results to the data file at the conclusion of a scan. By default `Ftail` is defined as nothing,

```
def Ftail   ''          # File tail
```

You might define it to be

```
def Ftail   'printf("#R %d %g %g %g %g %g %g\n", \
               SCAN_N, pl_xMAX, pl_MAX, pl_FWHM, pl_CWHM, pl_COM, pl_SUM);'
```

where the values being printed are from the `pl_anal()` function described on page ##. The `#R` characters begin the data file control line for scan results.

Temperature Control Macros

```
te                              # Read or set the temperature
settemp                         # Set the temperature
measuretemp                     # Measure the temperature
showtemp                        # Show temperature parameters
teramp                          # Ramp the temperature
```

Methods for handling temperature control and other experimental parameters are likely to vary greatly from lab to lab and experiment to experiment. You may be able to modify these standard macros to suit your specific needs.

The temperature control model assumed by these macros uses two independent instruments: one instrument to control the temperature and one instrument to measure the temperature. The following global variables are used by the macros:

TEMP_SP	The set point of the controller in ohms, volts, etc.
T_LO_SP	The lower limit for the controller set point.
T_HI_SP	The upper limit for the controller set point.
DEGC_SP	The temperature from which the set point is derived.
TEMP_CS	The value of the temperature sensor in ohms, volts, etc.
DEGC	The measured temperature.

The macro below displays the current set point and measured temperature.

```
# Display temperature parameters
def showtemp ’
      measuretemp
      printf("Temperature Setpoint = %g (%gC)\n",TEMP_SP,DEGC_SP)
      printf("            Measured = %g (%gC)\n",TEMP_CS,DEGC)
’
```

You must supply the macro measuretemp. It should read TEMP_CS from the temperature sensor and convert it to DEGC. Sample measuretemp macros are given below.

The te macro is the one you would use most often to display or set the temperature set point.

```
# Simple read or set temperature
def te ’
      if ($# == 1) {
            settemp $1
            qcomment "Temperature Setpoint at %g" "TEMP_SP"
      }
      onp; showtemp; offp
’
```

If invoked without arguments, it simply displays the current temperature parameters. Otherwise it invokes the settemp macro. The settemp macro checks its argument against the set point limits and then calls the _settemp macro, which you must supply.

```
# Assign the temperature setpoint
def settemp ’
      if ($# != 1) {
            print "Usage:  settemp set_point"
            exit
      } else {
            local _1
            _1 = $1
            if (_1 < T_LO_SP || _1 > T_HI_SP) {
                  printf("Temp limits are %g to %g.\n",T_LO_SP,T_HI_SP)
                  exit
            }
            TEMP_SP = _1
            _settemp
```

```
        }
'
```

Here are examples of _settemp macros from several installations (the symbol _1 is defined in settemp):

```
# Write setpoint to a Lakeshore 82C Controller on GPIB bus
def _settemp '
        gpib_put(12, sprintf("S%6.4f", _1))
'
# Write setpoint to home-made GPIB device used at MIT
def _settemp '{
        local _s
        _s = int(32767*_1/10)
        gpib_put(4, sprintf("%c%c%c%c\160\200",\
                0x80|( _s     &0xF), 0x90|((_s>>4 )&0xF),\
                0xA0|((_s>>8 )&0xF), 0xB0|((_s>>12)&0xF)))
}'
# Write setpoint to a home-made device used with CAMAC at Harvard
def _settemp '
        ca_put(bcd(10000*_1), 0, 0)
'
```

Here are examples of different measuretemp macros:

```
# Read parameters from a Lakeshore 82C Controller on GPIB bus
def measuretemp '{
        local _s
        gpib_put(12,"W0")
        _s=gpib_get(12)
        TEMP_SS=substr(_s,1,6)
        TEMP_CS=substr(_s,9,6)*100
        RtoT_0 DEGC TEMP_CS
        TEMP_SP=substr(_s,17,6)
        RtoT_0 DEGC_SP TEMP_SP
}'
# Read setpoint from CAMAC and temperature from GPIB device
def measuretemp '
        TEMP_CS = gpib_get(1)/1000
        RtoT_0 DEGC TEMP_CS
        TEMP_SP = dcb(ca_get(0, 0))/10000
        RtoT_0 DEGC_SP TEMP_SP
}'
```

Keep in mind that measuretemp is also called at each iteration of the standard scan macros.

The macro `RtoT_0`, used above, is one of several in the standard package that convert between degrees C and kilohms for common thermistors:

```
#  Temperature to kohms
def TtoR_0 ’
      local _k        # YSI 44011 (100kohm @ 25C)  20 to 120 C
      $1 = exp(-11.2942             +5.3483e3   /(_k = ($2) + 273.15)\
                -1.42016e5  /(_k*_k) -1.172e7    /(_k*_k*_k))
’
#  Kohms to temperature
def RtoT_0 ’
      local _l        # YSI 44011 (100kohm @ 25C)  20 to 120 C
      $1 = (1/(+2.2764e-3           +2.20116e-4 *(_l = log($2))\
               +2.61027e-6 *_l*_l    +9.02451e-8 *_l*_l*_l) - 273.15)
’
```

(The four parameters in each equation were obtained by fitting a table of values supplied by the manufacturer of the thermistors. No guarantees are made about the accuracy of the fitted parameters.)

The following macro will gradually change (or ramp) a temperature controller to a new set point. If the ramp time is greater than 500 seconds, the temperature is changed every 10 seconds, otherwise the temperature is changed every 2 seconds.

```
# Read or set or ramp the temperature
def teramp ’{
      if ($# == 1) {
              te $1
      } else if ($# == 2) {
              local _i _s1 _f1 _d1 _rtime _stime
              _f1 = $1
              _rtime = $2
              _stime = _rtime < 500? 2:10
              _s1 = TEMP_SP
              _d1 = (_f1 - _s1) / _rtime * _stime
              qcomment "Ramp Temp Setpoint from %g to %g" "_s1,_f1"
              for (_i=0; _i<=_rtime; _i+=_stime, _s1 += _d1) {
                      settemp _s1
                      measuretemp
                      printf("Set=%7.4f  Meas=%7.4fC\r",TEMP_SP,DEGC)
                      sleep(_stime)
              }
              showtemp
      } else {
              print "Usage:  teramp set_point  or  teramp set_point time"
              exit
      }
}’
```

Printer Initialization Macros

These macros send out the particular character sequences that put various printers into compressed mode to fit 132 columns of text on 8½" wide paper.

```
# Put DecWriter into compressed mode
def initdw ’onp; offt; printf("\033[4w"); offp; ont’

# Put Epson printer into compressed mode
def initfx ’onp; offt; printf("\017"); offp; ont’

# Put Okidata printer into compressed mode
def initoki ’onp; offt; printf("\035"); offp; ont’

# NEC P6/P7 printer, put into compressed mode
def initnec ’onp; offt; printf("\033!\004"); offp; ont’
```

The Scan Macros In Detail

All the scan macros in the standard package share a similar structure. To keep the format of the output sent to the data file, printer and screen consistent, common parts of each scan are defined as macros that are called by all the scans. For example, the scan_head macro is called by each scan to write scan headers on all the output files and devices. Certain macros are shared by all the scans for another reason. Special operating modes or options are implemented by redefining shared macros. For example, the scan_move macro, called within the looping portion of the scans, is normally defined as _move , which is:

```
def _move  'move_em; waitmove; getangles; calcHKL'
```

In powder mode, scan_move is defined as _pmove , a slightly more complicated macro, designed to move the designated powder averaging motor some width on alternating sides of the center trajectory of the scan,

```
def _pmove '
      if (_stype&2)
              _cp = A[_pmot]
      A[_pmot] = _cp + _pwid/2
      _pwid = -_pwid
      move_em; waitmove; getangles; A[_pmot] = _cp; calcHKL
'
```

The following paragraphs explain in detail the construction of the scan macros, using the single-motor scan, ascan , as an example. Here is its definition:

```
def ascan '
      if ($# != 5) {
              print "Usage:  ascan motor start finish intervals time"
              exit
      }
      _check0 "$1"
      { _m1 = $1; _s1 = $2; _f1 = $3; _n1 = int($4); _ctime = $5 }

      if (_n1 <= 0) {
              print "Intervals <= 0"
              exit
      }

      _bad_lim = 0
      _chk_lim _m1 _s1
      _chk_lim _m1 _f1
      if (_bad_lim) exit

      HEADING = sprintf("ascan %s %g %g %g %g","$1",$2,$3,$4,$5)
      _d1 = (_f1 - _s1) / _n1++
      _cols=4
```

```
        X_L = motor_name(_m1)
        _sx = _s1 ; _fx = _f1
        _stype = 1|(1<<8)
        FPRNT=sprintf("%s  H  K  L", motor_name(_m1))
        PPRNT=sprintf("%8.8s", motor_name(_m1))
        VPRNT=sprintf("%9.9s", motor_name(_m1))
        scan_head
        def _scan_on \'
         for (; NPTS < _n1; NPTS++) {
                A[_m1] = _s1 + NPTS * _d1
                scan_move
                FPRNT=sprintf("%g %g %g %g",A[_m1],H,K,L)
                PPRNT=sprintf("%8.4f",A[_m1])
                VPRNT=sprintf("%9.4f",A[_m1])
                scan_loop
                pl_put(NPTS, A[_m1], S[DET])
                scan_plot
         }
         scan_tail
        \'
        _scan_on
'
```

In `ascan`, as in all scans, the first thing to do is to check the number of arguments, `$#`, and if incorrect, print a usage message:

```
        if ($# != 5) {
                print "Usage:  ascan motor start finish intervals time"
                exit
        }
```

Next, the `_check0` macro is called,

```
        _check0 "$1"
```

as it is whenever a motor mnemonic is used as an argument in the standard macros. The macro checks its argument against all valid motor mnemonics and motor numbers. The purpose is to prevent unintentionally sending motors into motion if the user mistypes a mnemonic. The definition of `_check0` is

```
def _check0 '{
      local _i

      for (_i = 0; _i <= MOTORS; _i++)
              if (_i == MOTORS) {
                      print "Invalid motor name:  $1"
                      exit
              } else if ($1 == _i) {
                      if ("$1" != motor_mne(_i) && "$1" != _i) {
                              print "Invalid motor name:  $1"
                              exit
                      } else
                              break
              }
'
```

Next in ascan, the global variables used in the scan are initialized from the arguments.

```
        { _m1 = $1; _s1 = $2; _f1 = $3; _n1 = int($4); _ctime = $5 }
```

The global variables being assigned are shared by all the scans.

Next in ascan, a check is made to ensure the number of intervals is positive.

```
        if (_n1 <= 0) {
                print "Intervals <= 0"
                exit
        }
```

The next four lines do a motor limit check before the start of the scan.

```
        _bad_lim = 0
        _chk_lim _m1 _s1
        _chk_lim _m1 _f1
        if (_bad_lim) exit
```

The _chk_lim macro sets the flag _bad_lim if the position given by the second argument is outside the limits of the motor given by the first argument.

```
def _chk_lim '{
      local   _u _t

      if ((_u = dial($1, $2)) < (_t = get_lim($1, -1))) {
              printf("%s will hit low limit at %g.\n",motor_name($1),_t)
              _bad_lim++
      } else if (_u > (_t = get_lim($1, 1))) {
              printf("%s will hit high limit at %g.\n",motor_name($1),_t)
              _bad_lim++
      }
}'
```

The prescan limit check is straightforward for simple motor scans. For reciprocal

space scans, the limit check must loop through all the points of the scan since the motor positions are not necessarily monotonic functions of the scan variables.

Next in `ascan`, the global variable `HEADING` is initialized.

```
HEADING = sprintf("ascan %s %g %g %g %g","$1",$2,$3,$4,$5)
```

It is used in the scan headers written to the file, screen and printer, and records the arguments with which the scan was invoked.

Next, some global scan variables are initialized.

```
_d1 = (_f1 - _s1) / _n1++
_cols=4
X_L = motor_name(_m1)
_sx = _s1 ; _fx = _f1
_stype = 1|(1<<8)
```

The `_d1` variable is set to the step size for the scan. The number of intervals in `_n1` is incremented so its value will be the actual number of points. The `_cols` global variable is set to the number of extra columns this scan will use in the data file. Here it is four, for the motor position and values of H, K and L at each point.

`X_L` is set to the x-axis label to use on the plot of the scan. The globals `_sx` and `_fx` are set to the endpoints of the x axis to be used in plotting the data on the screen during the scan.

The variable `_stype` is treated as a two byte integer and holds a code representing the current scan type. The low-order byte is a bit flag, while the high order byte contains a number value. The expression `1|(1<<8)` use the bitwise-or and the bitwise-shift operators to put values in each byte. Currently, the following codes are used:

Code	Type Of Scan	High-Order Byte
1	motor	number of motors
2	HKL	nothing
4	temperature	nothing

Next in `ascan`, the global variables `FPRNT`, `PPRNT` and `VPRNT` are given string values to be used for file, printer and video-screen column labels particular to this scan.

```
FPRNT=sprintf("%s  H  K  L", motor_name(_m1))
PPRNT=sprintf("%8.8s", motor_name(_m1))
VPRNT=sprintf("%9.9s", motor_name(_m1))
```

Each label contains the name of the motor being scanned, although printed with a different field width. Different widths are used to fit the widths and number of fields on the target devices. A challenge in constructing the scan macros is to fit all the desired columns of information within a single line. All the scans must limit the line

length to 132 columns for output sent to the printer. (80-column printers must be operated in compressed mode to make their carriages effectively 132 columns wide.) The video screen is 80 columns wide. For the data file, there is no restriction on width. Also for the data file, no attempt is made to line up items in columns.

Next in ascan is the scan_head macro, called to do the general initialization. All scan macros call scan_head. The default definition of scan_head is

```
def scan_head  '_head'
```

where _head is defined as,

```
def _head '
      _scan_time
      waitall; get_angles; calcHKL
      NPTS = T_AV = MT_AV = 0
      DATE = date()
      TIME = TIME_END = time()
      _cp = A[_pmot]
      rdef cleanup "_scanabort"

      # DATA FILE HEADER
      ond; offt
      printf("\n#S %d  %s\n#D %s\n",++SCAN_N,HEADING,DATE)
      if (_ctime < 0)
              printf("#M %g  (%s)\n", -_ctime, S_NA[MON])
      else
              printf("#T %g  (%s)\n", _ctime, S_NA[sec])
      printf("#G")
      for (_i=0; _i<NPARAM; _i++) printf(" %g", G[_i])
      printf("\n")
      printf("#Q %g %g %g\n", H, K, L)
      {
              local _i _j _k
              for (_i = 0, _k = MOTORS; _i < _k; _i += 8) {
                      printf("#P%d ", _i/8)
                      _mo_loop .6g "A[mA[_j]]"
              }
      }
      Fheader
      printf("#N %d\n", _cols + 3)
      printf("#L %s%s  Epoch  %s  %s\n",FPRNT,Flabel,\
              S_NA[_ctime < 0? sec:MON],S_NA[DET])
      offd

      # PRINTER HEADER
      onp; offt
      printf("\n\f\nScan %3d   %s   file = %s  %s  user = %s\n%s\n\n",\
              SCAN_N,DATE,DATAFILE,TITLE,USER,HEADING)
      {
              local _i _j _k
              for (_i = 0, _k = MOTORS; _i < _k; _i += 8) {
```

```
                        printf("  ")
                        _mo_loop 9.9s "motor_name(mA[_j])"
                        printf("  ")
                        _mo_loop 9.6g "A[mA[_j]]"
                }
        }
        Pheader
        printf("\n  # %11.9s %11.9s %11.9s %8.8s %8.8s %8.8s %s%s\n",\
                "H","K","L",S_NA[sec],S_NA[MON],S_NA[DET],PPRNT,Plabel)
        offp

        # TTY HEADER
        ont
        printf("\nScan %3d   %s   file = %s  %s  user = %s\n%s\n\n",\
                SCAN_N,DATE,DATAFILE,TITLE,USER,HEADING)
        printf("  # %s %8.8s %8.8s %10.10s%s\n",\
                VPRNT,S_NA[DET],S_NA[MON],S_NA[sec],Plabel)
'
```

The commands at the beginning of _head,

```
        waitall; get_angles; calcHKL
```

insure the motors are stopped and positions current before proceeding. Next, _head initializes some variables. NPTS is the loop variable in the scans that will run from 0 to _n1. T_AV and MT_AV maintain the average temperature (from the global variable DEGC) and the average monitor counts or time per point during the scan. DATE and TIME are set to the current date and time. TIME_END is updated at each point with the current time. The _cp variable is used in powder mode and is set to the center position of the powder-average motor.

Next in the header macro, the real space motor positions and the reciprocal-space position are made current with getangles and calcHKL. The cleanup macro is defined to be the standard macro _scanabort. The macro named cleanup is special as spec automatically invokes that macro when a user types **^C** or on any other error, such as hitting motor limits, trying to go to an unreachable position or encountering a syntax error in a macro. The definition of _scanabort is,

```
def _scanabort '
      _cleanup2
      _cleanup3
      comment "Scan aborted after %g points" NPTS
      sync
      undef cleanup
'
```

The _cleanup2 macro is defined for delta scans to move motors back to their starting positions. The _cleanup3 macro is available to users for defining some kind of private clean up actions.

Finally, the headers are written to the file, printer and screen in turn. Included in the headers are the user-defined `Fheader`, `Flabel`, `Pheader` and `Plabel`.

Returning back to `ascan`, the next part of the macro is the loop:

```
def _scan_on \'
 for (; NPTS < _n1; NPTS++) {
        A[_m1] = _s1 + NPTS * _d1
        scan_move
        FPRNT=sprintf("%g %g %g %g",A[_m1],H,K,L)
        PPRNT=sprintf("%8.4f",A[_m1])
        VPRNT=sprintf("%9.4f",A[_m1])
        scan_loop
        pl_put(NPTS, A[_m1], S[DET])
        scan_plot
 }
 scan_tail
\'
_scan_on
```

The loop is implemented as a macro to enable the scan to be continued with the `resume` macro. The relevant global variables are initialized outside the loop, so that invoking `_scan_on` continues the scan where it had left off when interrupted. Here is the `resume` macro.

```
def resume '
      if (NPTS >= (index(HEADING, "mesh")? _n1*_n2 : _n1)) {
              print "Last scan appears to be finished."
              exit
      }
      def cleanup "_scanabort"
      comment "Scan continued"
      _scan_on
'
```

The `scan_move`, `scan_loop` and `scan_plot` macros are invoked by all the scans. In the loop, the motor array `A[]` is set to the target position for the scanned motor and the motor is moved using the `scan_move` macro, normally defined as `_move`:

```
def _move  'move_em; waitmove; getangles; calcHKL'
```

String variables are then assigned to values that will be written to the output devices using the `scan_loop` macro. The `scan_loop` macro is generally defined as `_loop` which has the definition,

```
# The loop macro, called by all the scans at each iteration
def _loop '
      scan_count _ctime
      measuretemp
      calcHKL
      z = _ctime < 0? S[sec]/1000:S[MON]
      T_AV += DEGC; MT_AV += z
      printf("%3d %s %8.0f %8.0f %10.6g%s\n",\
              NPTS,VPRNT,S[DET],S[MON],S[sec]/1000,Pout)
      onp; offt
      printf("%3d %11.5g %11.5g %11.5g %8.6g %8.0f %8.0f %s%s\n",\
              NPTS,H,K,L,S[sec]/1000,S[MON],S[DET],PPRNT,Pout)
      offp; ond; offt
      printf("%s%s %d %g %g\n",FPRNT,Fout,(TIME_END=time())-EPOCH,z,S[DET])
      offd; ont
'
```

This macro first counts by calling the `scan_count` macro, normally defined as `_count`, which is, in turn, defined as `count`. (In powder mode, or when using updated counting during scans, `_count` is defined differently.) The `_loop` macro then calls `measuretemp`. With this macro, you can have any per-point action done, not limited to, nor necessarily even including, measuring the temperature of the sample. Next in `_loop` the sums for computing the average temperature and monitor count rate are adjusted. Finally the video screen, printer and data file are updated with the results of the current iteration.

The last thing in `_scan_on` is a call to `scan_tail`, normally defined as `_tail`:

```
# The tail macro, called by all the scans when they complete
def _tail '
      undef cleanup
      TIME_END = time()
      if (!(_stype&8)) {
              ond; offt
              Ftail
              offd; ont
              plot
      }
'
```

This macro removes the definition of `cleanup`, since it is no longer needed, and if not a *mesh* scan, adds the user defined results to the file and calls the `plot` macro.

Standard Data-File Format

The data files created by the macros have a simple format. The files are ASCII. Control lines in the file begin with a # followed by a upper-case letter. Other lines are blank or contain scan data.

The control conventions are:

Code	Parameters	Description
`#C`	*comment*	Comments inserted by many of the standard macros.
`#D`	*date*	A string representing the current date, in the format `Wed May  4 23:59:49 1988`.
`#E`	*seconds*	The UNIX epoch at the time the file was created.
`#F`	*filename*	The name by which the file was created.
`#G1`	*parameters*	Array `G[]` (geometry mode, sector, etc.)
`#G2`	*parameters*	Array `U[]` (lattice constants, orientation reflections)
`#G3`	*parameters*	Array `UB[]` (orientation matrix)
`#G4`	*parameters*	Array `Q[]` (LAMBDA, frozen angles, cut points, etc.)
`#I`	*factor*	A normalizing factor to apply to the data.
`#j`*N*	*mnemonics*	Counter mnemonics (*N* = 0,1,2,... eight per row)
`#J`*N*	*names*	Counter names (*N* = 0,1,2,... eight per row, each separated by two spaces)
`#L`	*labels*	Labels for the data columns (each separated by two spaces)
`#M`	*counts*	If counting to monitor counts, the number of counts.
`#N`	*num [num2]*	The number of columns of data that follow (num2 sets per row)
`#o`*N*	*mnemonics*	Motor mnemonics (*N* = 0,1,2,... eight per row)
`#O`*N*	*names*	Motor names (*N* = 0,1,2,... eight per row, each separated by two spaces)
`#P`*N*	*positions*	Postions of motors corresponding to #O/#o above.
`#Q`	*H K L*	A reciprocal space position.
`#R`	*results*	User-defined results from a scan.
`#S`	*number*	A new scan having scan number *number* follows, normally preceded by a blank line.
`#T`	*seconds*	If counting to time, the time used.
`#U`		Reserved for user.
`#X`	*setpoint*	The temperature setpoint.
`#@MCA`	*fmt*	This scan contains MCA data (array_dump() format, as in `"16C"`)
`@A`	*data*	MCA data formatted as above

`#@CALIB`	*a b c*	Coefficients for x[*i*] = a + b *i* + c i^2 for MCA data.
`#@CHANN`	*n f l r*	MCA channel information (number_saved, first_saved, last_saved, reduction coef)
`#@CTIME`	*p l r*	MCA count times (preset_time, elapsed_live_time, elapsed_real_time)
`#@ROI`	*n f l*	MCA ROI channel information (ROI_name, first_chan, last_chan)

FOUR-CIRCLE REFERENCE

Introduction

When invoked by the name *fourc*, spec runs with code appropriate for a four-circle diffractometer. This section of the *Reference Manual* focuses on the features of spec unique to the *fourc* version.

The four circles of the standard four-circle diffractometer are: 2θ, the angle through which the beam is scattered, and θ, χ, and ϕ, the three Euler angles, which orient the sample. Of these three, θ is the outermost circle with its axis of rotation coincident with that of 2θ. The χ circle is mounted on the θ circle, with its axis of rotation perpendicular to the θ axis. The ϕ circle is mounted on the χ circle such that its axis of rotation lies in the plane of the χ circle.

From the keyboard and on the screen, the angles are named `tth`, `th`, `chi` and `phi`, respectively, and conventionally referred to in that order. For *fourc* to work properly, angles with these names must be configured.

In describing the operation of a four-circle diffractometer, it is convenient to consider three coordinate systems: 1) a frame fixed in the laboratory, 2) a frame fixed on the spectrometer and 3) the natural axes of the sample. Note that *fourc* uses right-handed coordinate systems. All rotations are right-handed except for the χ rotation.

(1) The x-y plane of the laboratory coordinate system is called the scattering plane and contains the sample and the points reached by the detector as it rotates on the 2θ arm. A counter-clockwise rotation of the 2θ axis corresponds to increasing 2θ, with the 2θ rotation axis defining the positive z direction in the laboratory. The zero of 2θ is defined as the setting at which the undeflected X-ray beam hits the detector.

The positive y axis is along the line from the sample to the X-ray source. The position at which θ rotates the χ circle to put the χ rotation axis along the y axis defines the zero of θ. A clockwise rotation of χ corresponds to increasing χ.

The zero of χ is the position which puts the ϕ rotation axis along the positive z axis. The positive x axis direction is determined by the cross product of the y and z axes ($\hat{\mathbf{x}}=\hat{\mathbf{y}}\times\hat{\mathbf{z}}$), which completes the definition of the right-handed coordinate system.

It is important to note that the zeroes of 2θ, θ and χ and the direction of positive rotation of all the circles must be set as described above and cannot be freely redefined.

(2) The spectrometer coordinate system is defined as a right-handed system fixed on the ϕ rotation stage at the sample position such the coordinate system is aligned with the laboratory coordinate system when all four spectrometer angles are zero. This definition determines the zero of ϕ.

(3) The third coordinate system is aligned with specific directions in the sample. A common and useful example are coordinates defined as the lattice vectors of a crystalline sample. When placing a sample in the spectrometer, it is unlikely that its axes will line up with the spectrometer axes. Nevertheless, *fourc* allows the sample orientation to be fully specified by finding the angles at which two Bragg peaks are detected and giving the corresponding reciprocal lattice indices. This process is described fully in the section on the *Orientation Matrix*.

To orient a sample so as to measure the intensity at a particular reciprocal lattice position requires that the reciprocal lattice vector of interest is aligned with the scattering vector of the spectrometer. Since any rotation about the scattering vector does not change the diffraction condition, there is a high degree of degeneracy that must be resolved in order for *fourc* to determine unique angle settings. How the degeneracy is lifted described in the section on *Four-Circle Modes*.

Diffractometer Alignment

This section presents a guide on how to set up a four-circle spectrometer. This summary applies whether the scattering plane is horizontal or vertical.

The first step, which should need only be done the first time *fourc* is used with the diffractometer, is to ensure each diffractometer motor is set up with the correct name, mnemonic, rotation sense, steps-per-degree, etc. The `config` macro is generally used for this purpose. For stepping motors, the rotation sense of an axis depends on the details of the motor controller and cable connections. If the rotation sense isn't as described in the *Introduction*, change the *sign_of_user_×_dial* parameter in the motor configuration file.

For each motor, *fourc* keeps track of a both a dial and a user position. The dial position is meant to agree with the readout of the physical dial on the spectrometer. The value and the sign of the *steps_per_unit* parameter should be chosen so that the dial position and its direction in the computer agree with the physical dial reading. Use the `set_dial` macro to set the dial positions. The user positions should correspond to the underlying "true" orientation angles of the spectrometer that satisfy the constraints given above. Use the `set` macro to set the user positions.

Once properly configured, diffractometer alignment proceeds as follows.

(1) Arrange for the X-ray beam to go through the center of rotation. Generally, the center of rotation is found with a pin and a telescope.

(2) Arrange for the X-ray beam to be perpendicular to the 2θ axis. This condition is typically verifyed by comparing X-ray burns made on X-ray sensitive paper with 2θ near the undeflected beam direction and with 2θ offset by 180°.

(3) Set 2θ so that the undeflected X-ray beam direction corresponds to the zero of 2θ.

(4) Align the χ rotation axis with the laboratory y axis to set the zero of θ.

(5) Align the ϕ rotation axis with the θ rotation axis to set the zero of χ.

One way to do (4) and (5) is as follows:

(i) Mount a Si(111) wafer so that the (111) direction is (approximately) along the ϕ axis.

(ii) Find the (111) Bragg reflection. Note the values of θ and χ. Call them θ_1 and χ_1.

(iii) Rotate ϕ by 180°.

(iv) Find the Bragg reflection again. Note thes values of θ and χ. Call them θ_2 and χ_2.

(v) $\frac{1}{2}(\chi_1 + \chi_2)$ corresponds to $\chi = 90°$ in a correctly aligned spectrometer; $\frac{1}{2}(\theta_1 + \theta_2)$ corresponds to $\theta = \frac{1}{2}2\theta$.

The Huber four-circle diffractometer is an example of an instrument that works with *fourc*.[1] Another common spectrometer configuration has two crossed ± 20° tilt stages on top of full θ and 2θ circles. This configuration is compatible with the four-circle code if the tilt stage immediately adjacent to the θ circle is χ and the other is ϕ. When the first tilt stage is zero, χ is at 90°.

[1]The four-circle Huber has dial readings with all right-handed rotations, so the χ circle should have dial readings and user readings that are in opposite senses. If the θ circle is offset by 180° then the dial readings and the user readings of all angles can have the same sense.

Orientation Matrix

Angle calculations for the four-circle diffractometer are described in detail in Busing and Levy.[2] You may also refer to that paper to learn how to calculate the *orientation matrix*. The orientation matrix, **UB**, describes the sample orientation with respect to the diffractometer angles. Given **UB**, it is possible to calculate the diffractometer angles $(2\theta, \theta, \chi, \phi)$ necessary to rotate a particular scattering vector **Q**, indexed by (H, K, L), into the diffraction position. The matrix **B** transforms the given (H, K, L) into an orthonormal coordinate system fixed in the crystal. The matrix **U** is the rotation matrix that rotates the crystal's reference frame into the spectrometer's.

The first step in constructing an appropriate orientation matrix is to enter the sample crystal lattice parameters a, b, c, α, β and γ.[3] These are real-space parameters, as might be found in Wychoff[4] or Pearson.[5] Use the macro `setlat` to assign values:

```
1.FOURC> setlat 3.61 3.61 3.61 90 90 90

2.FOURC>
```

Next, you must specify the sets of values of $(2\theta, \theta, \chi, \phi)$ at which two Bragg reflections are in the diffracting position. One of these is called the primary reflection. *Fourc* ensures that the values of (H, K, L) reported for the primary reflection agree (to within a scale factor) with the values entered. However, because of experimental errors and/or uncertainties in the unit cell parameters, the values of (H, K, L) reported for the other Bragg reflection, called the secondary reflection, may not agree perfectly with the entered values (although they should be close).

You can use the `or0` and `or1` macros to enter the parameters for the primary and secondary reflections, respectively. However, the `or0` and `or1` macros require that the diffractometer be moved to the associated reflections, as these macros use the current angles and the entered (H, K, L) in the calculation of the orientation matrix. Alternatively, you can use the `setor0` and `setor1` macros, which prompt for both (H, K, L) and the angles that define the orientation matrix, without moving the spectrometer to the given settings.

---

[2] W. R. Busing and H .A. Levy, *Acta Cryst.* **22**, 457 (1967). Note however, that this paper uses right-handed coordinates systems and left-handed rotations for all rotations except for χ, which is right-handed.

[3] The conventional symbols for the crystal lattice angles include α and β. These angles are unrelated to the orientation angles α and β defined in the introduction. The different meanings should be clear from context.

[4] R. W. G. Wychoff, *Crystal Structures* (Wiley, New York, 1964).

[5] P. Villars and L.D. Calvert, *Pearson's Handbook of Crystallographic Data for Intermetallic Phases* (American Society for Metals, Metals Park, Ohio, 1985).

Four-Circle Modes

As noted above, because there are three Euler angles (θ, χ, ϕ) while the direction of the scattering vector $\mathbf{Q}$ is specified by only two angles, there is a degeneracy associated with the transformation from (H, K, L) to $(2\theta, \theta, \chi, \phi)$. The degeneracy is resolved in *fourc* by providing a constraint. In *fourc*, the different constraints are called *modes*. The value of the g_mode geometry parameter determines the prevailing mode.

Fourc defines several angles in order to specify certain of the modes.

The angle ω is defined as $\omega = \theta - (2\theta)/2$, and is referred to as OMEGA.

The angle ψ (referred to as AZIMUTH) specifies a clockwise rotation about the diffraction vector. The zero of ψ is determined by a reference vector, different from the diffraction vector. The azimuthal angle ψ is defined to be zero when this vector is in the diffraction plane.

The angles α (ALPHA) and β (BETA) are defined such that the angles between the azimuthal reference vector and the incident and scattered X-rays are $90°-\alpha$ and $90°-\beta$, respectively. One commonly used azimuthal reference vector is the sample's surface normal, which then makes α and β correspond to the incident and exit angles of the X-rays on the surface.

Omega Equals Zero (g_mode = 0)

The simplest constraint is $\omega = 0$. Any (H, K, L) with a small enough 2θ can be reached in this mode.

Omega Fixed (g_mode = 1)

In this mode ω is fixed to a finite value. Suppose you want to go the the $(0, 0, 2)$ Bragg reflection but with $\omega = 10$. You would then type

```
2.FOURC> setmode 1

Now using Omega-Fixed mode.

3.FOURC> OMEGA=10

4.FOURC> br 0 0 2

5.FOURC>
```

Zone or χ *and* ϕ *Fixed (*g_mode = *2)*

A zone of reciprocal space is a plane passing through the origin. A zone axis is the direction of the normal to this plane. The zone axis can be specified by the vector product of any two non-colinear points in the zone. **Zone** mode forces the specified zone axis to be normal to the scattering plane. In other words, a zone of the crystal is leveled into the scattering plane of the spectrometer. Any point in this plane can then be reached with θ and 2θ (i.e., χ and ϕ are fixed). The geometry code provides for the calculation of the χ and ϕ necessary to put any two reciprocal space positions in the scattering plane via the cz, sz or mz macros. These macros are explained later.

Phi Fixed or *Three Circle (*g_mode = *3)*

The angle ϕ is fixed at some arbitrary value.

*Azimuth Fixed (*g_mode = *4)*

This mode fixes the value of the rotation angle ψ of a reference vector about the scattering vector **Q**.

Azimuth-fixed mode provides a degree of control that is particularly useful in a surface-diffraction experiment. If the reference vector is chosen to be the surface normal, setting ψ to 90° rotates the surface normal into the plane defined by the scattering vector and the diffractometer θ-2θ axis. This means that the incidence angle α will equal the exit angle β.

The reference vector is defined by the geometry parameters g_haz, g_kaz, and g_laz. Use the macro setaz to specify the (*H, K, L*) of the reference vector. For example, to set the reference vector to (0, 0, 2), use

```
5.FOURC> setaz 0 0 2

6.FOURC>
```

Azimuth-fixed mode will fail if you try to make measurements with **Q** parallel to the reference vector, since there is then no way to define a rotation about **Q**. The remedy is either to switch to another mode (usually the best choice) or to switch to another reference vector.

*Alpha Fixed (*g_mode = *5)*

This mode allows you to hold the value of α constant while moving to various values of (*H, K, L*). This is useful in experiments in which it is necessary to control the X-ray penetration depth into a sample. More generally, if looking at a weak signal from

a surface, keeping α small will keep the background small with no restriction (in principle) on the momentum transfer normal to the surface.

Suppose you are studying a sample of copper and you want to fix the incidence angle to be equal to the critical angle for total external reflection. Set the reference vector to the surface normal of the crystal and then type the following commands:

```
6.FOURC> setmode 5
Now using Alpha-fixed mode.

7.FOURC> ALPHA=0.4126

8.FOURC> br 0 2 2

9.FOURC>
```

To implement **alpha-fixed** mode, *fourc* calculates the value of ψ needed to fix the angle between the incident wave vector and the reference vector.[6]

*Beta Fixed (*g_mode = *6)*

This is the same as **alpha-fixed** mode except that β, rather than α, may be fixed.

[6]S. G. J. Mochrie, *J. Appl. Cryst.* **21**, 1-3 (1988).

Freezing Angles

For the **omega-fixed**, **phi-fixed**, **zone**, **azimuth-fixed**, **alpha-fixed** and **beta-fixed** modes you may *freeze* the value of the associated angle (or angles), so when calculating motor positions corresponding to an arbitrary (*H, K, L*) using `calcA` (within `br`, for example), the angle (or angles) will be reset to the frozen value before the calculation no matter what the current position of the diffractometer.

The macro `freeze` waits until all motors have stopped moving, then sets a variable (`g_frz`) indicating frozen mode is on and saves the current position of the frozen angle in another variable. Usage might be:

```
9.FOURC> setmode 1
Now using Omega-Fixed mode.

10.FOURC> freeze 5

Sun Jan 28 12:16:23 1990.  Freezing Omega at 5.

11.FOURC>
```

If the value to freeze for the current mode is not given as an argument to the macro, the current value of the related angle or parameter is used to set the frozen value. In **zone** mode, both the χ and ϕ values need to be given as arguments.

The macro `unfreeze` sets `g_frz` to zero. Subsequent angle calculations will use whatever the current value of the associated constrained angle or angles for the current mode.

Sectors

Sectors correspond to different symmetry transformations of $(2\theta, \omega, \chi, \phi)$ and may be of help in avoiding blind spots. Sectors can also be useful for samples in cryostats or ovens. All modes can have the positions chosen for the motors further influenced by the choice of sector. However, the modes **azimuth-fixed**, **alpha-fixed** and **beta-fixed** only allow sectors numbered zero through three, below.

For a given pair of incident and scattered X-ray beams, $\mathbf{k}_i$ and $\mathbf{k}_f$, there are eight orientations of the crystal in the spectrometer that give the same scattering since they present equivalent projections to the incident and scattered beams. The eight orientations are labeled as sectors 0 through 7.

Four of the orientations come from the symmetries of a pair of vectors. They correspond to the identity operation (i.e., the current diffraction angles), a rotation of 180° about $\mathbf{k}_i$, a rotation of 180° about the bisector of $\mathbf{k}_i$ and $\mathbf{k}_f$, and a rotation about z by 180° – 2θ. The last two symmetries are based on interchanging the role of the entrance angle and the exit angle of the X-rays on the sample. These four symmetries give sectors 0, 2, 4 and 6 in *fourc*. For each of these four positions another position can be obtained by increasing θ by 180°, decreasing ϕ by –180° and changing the sign of χ. Since a rotation of 180 and of –180 are the same, both θ and ϕ can be increased by 180. These orientations give sectors 1, 3, 5 and 7. Studying these operations shows that sectors 2, 3, 6 and 7 have opposite signs of 2θ from the current position. Sectors 2, 3, 4 and 6 have flipped the *up* direction of the sample (normal to the scattering plane) to the *down* direction.

The value of `g_sect` determines in which sector of reciprocal space the diffractometer operates.

The actual transformations of the angles are:

	0	1	2	3	4	5	6	7
$2\theta \rightarrow$	2θ	2θ	-2θ	-2θ	2θ	2θ	-2θ	-2θ
$\omega \rightarrow$	ω	$\omega - 180°$	$-\omega$	$180° - \omega$	$-\omega$	$180° - \omega$	ω	$\omega - 180°$
$\chi \rightarrow$	χ	$-\chi$	$\chi - 180°$	$180° - \chi$	$180° - \chi$	$\chi - 180°$	$-\chi$	χ
$\phi \rightarrow$	ϕ	$\phi - 180°$	ϕ	$\phi - 180°$	$\phi - 180°$	ϕ	$\phi - 180°$	ϕ

In addition, a sector 8 is defined to minimize $|\chi - 90°|$ and $|\phi|$. It can be used when the χ and ϕ circles of the diffractometer are arc segments rather than a complete circle.

Cut Points

The angles –90° and +270° are at the same position on a circle, but if a motor in spec is sitting at 0°, it will move counter-clockwise by 90° to get to the –90° position and clockwise by 270° to get to the +270° position.

By setting a lower cut point for a particular motor, you can choose what value on the circle the four-circle angle calculations will produce. That is, the calculations will place the angle between the lower cut point and that value plus 360°.

Cut points can be set for the θ, χ and ϕ four-circle angles. For the 2θ angle, the lower cut point is fixed at –180°. For the **azimuth-fixed** modes, there is pseudo-cut point for the azimuthal angle. If the cut point is less than zero, the calculated azimuth will be between –180° and 0°, otherwise the azimuth will be between 0° and +180°.

The default cut points are shown below.

Angle	Default Cut Point
θ	–180
χ	–180
ϕ	–180
ψ	0

The macro `cuts` can be used to set or display the cut points. With no arguments, it displays the current cut points. Two arguments are used to set a single cut point. Four arguments set all cut points.

```
11.FOURC> cuts

Cut Points:

    th   chi   phi  azimuth
  -180  -180  -180        0

12.FOURC> cuts phi 90
```

or

```
13.FOURC> cuts -180 -180 90 0

14.FOURC>
```

Four-Circle Files

Most questions regarding the behavior of spec when used with a four-circle diffractometer can ultimately be resolved by consulting the appropriate source code. The file *geo_fourc.c* in the standard distribution of spec contains the code for all the four-circle calculations. The file *u_hook.c* contains a few lines of code that connect the code in *geo_fourc.c* with the rest of the program. Finally, the files *macros/fourc.src* and *macros/ub.mac* in the standard distribution contains the definitions for all the four-circle and orientation matrix macros.

Four-Circle Variables

The four-circle coordinate variables (H, K and L, in particular) are stored in a built-in array named `Q[]`. The four-circle geometry calculations either use the motor positions contained in the `A[]` array to calculate values for the `Q[]` parameters or place motor positions in `A[]` based on the current values in `Q[]`. Each four-circle variable has a descriptive macro definition as an alias, such as `def OMEGA 'Q[6]'`.

Variable	Alias	Description
`Q[0]`	`H`	x component of the scattering vector.
`Q[1]`	`K`	y component of the scattering vector.
`Q[2]`	`L`	z component of the scattering vector.
`Q[3]`	`LAMBDA`	Incident X-ray wavelength λ.
`Q[4]`	`ALPHA`	Incident angle α.
`Q[5]`	`BETA`	Exiting angle β.
`Q[6]`	`OMEGA`	$\omega = \theta - (2\theta)/2$.
`Q[7]`	`AZIMUTH`	Azimuthal angle.
`Q[8]`	`F_ALPHA`	Frozen value of α for **alpha-fixed** mode.
`Q[9]`	`F_BETA`	Frozen value of β for **beta-fixed** mode.
`Q[10]`	`F_OMEGA`	Frozen value of ω for **omega-fixed** mode.
`Q[11]`	`F_AZIMUTH`	Frozen value of ψ for **azimuth-fixed** mode.
`Q[12]`	`F_PHI`	Frozen value of ϕ for **phi-fixed** mode.
`Q[13]`	`F_CHI_Z`	Frozen value of χ for **zone** mode.
`Q[14]`	`F_PHI_Z`	Frozen value of ϕ for **zone** mode.

The geometry parameters in the table below affect the geometry calculations in various ways. Although the parameters can be changed by assignment, the preferred method is to use the indicated macro for setting the parameters.

Variable	Alias	Related Macro	Description
G[0]	g_mode	setmode	Specifies the four-circle *mode*.
G[1]	g_sect	setsector	Specifies the *sector*.
G[2]	g_frz	freeze	Nonzero when an angle is *frozen*.
G[3]	g_haz	setaz	H of the azimuthal reference vector.
G[4]	g_kaz	setaz	K of the azimuthal reference vector.
G[5]	g_laz	setaz	L of the azimuthal reference vector.
G[6]	g_zh0	mz, sz	H of first **zone**-mode vector.
G[7]	g_zk0	mz, sz	K of first **zone**-mode vector.
G[8]	g_zl0	mz, sz	L of first **zone**-mode vector.
G[9]	g_zh1	mz, sz	H of second **zone**-mode vector.
G[10]	g_zk1	mz, sz	K of second **zone**-mode vector.
G[11]	g_zl1	mz, sz	L of second **zone**-mode vector.
U[0]	g_aa	setlat	a lattice constant in Angstroms.
U[1]	g_bb	setlat	b lattice constant.
U[2]	g_cc	setlat	c lattice constant.
U[3]	g_al	setlat	α lattice angle.
U[4]	g_be	setlat	β lattice angle.
U[5]	g_ga	setlat	γ lattice angle.
U[6]	g_aa_s	setrlat	a^* reciprocal lattice constant.
U[7]	g_bb_s	setrlat	b^* reciprocal lattice constant.
U[8]	g_cc_s	setrlat	c^* reciprocal lattice constant.
U[9]	g_al_s	setrlat	α^* reciprocal lattice angle.
U[10]	g_be_s	setrlat	β^* reciprocal lattice angle.
U[11]	g_ga_s	setrlat	γ^* reciprocal lattice angle.
U[12]	g_h0	or0, setor0	H of primary reflection.
U[13]	g_k0	or0, setor0	K of primary reflection.
U[14]	g_l0	or0, setor0	L of primary reflection.
U[15]	g_h1	or1, setor1	H of secondary reflection.
U[16]	g_k1	or1, setor1	K of secondary reflection.
U[17]	g_l1	or1, setor1	L of secondary reflection.
U[18]	g_u00	or0, setor0	Observed 2θ of primary reflection.
U[19]	g_u01	or0, setor0	Observed θ of primary reflection.
U[20]	g_u02	or0, setor0	Observed χ of primary reflection.

U[21]	g_u03	or0, setor0	Observed ϕ of primary reflection.
U[24]	g_u10	or1, setor1	Observed 2θ of secondary reflection.
U[25]	g_u11	or1, setor1	Observed θ of secondary reflection.
U[26]	g_u12	or1, setor1	Observed χ of secondary reflection.
U[27]	g_u13	or1, setor1	Observed ϕ of secondary reflection.

The first three parameters select modes and set flags. The next three parameters describe the components of the azimuthal reference vector. The six after that describe the *zone*-mode vectors.

The next sets of parameters describe the orientation matrix, including the lattice constants of the sample and the parameters of the primary and secondary orientation reflections. Remember that the calcG macro, described below, must be called to make sure the orientation matrix is recalculated after changing any of these related values above. The or0, setor0, or1, setor1, or_swap and setlat macros do just that.

Four-Circle Functions

You can access the four-circle calculations through spec's user-hook routine calc(). The table below summarizes the calculations available.

Function	Alias	Description
calc(1)	calcA	Calculate motor positions for current H K L.
calc(2)	calcHKL	Calculate H K L for motor positions in A[].
calc(4)	calcG	Recalculate orientation matrix.
calc(4,1)	USER_UB	Returns 0, 1 or 2 if **UB** is from reflections, entered direcly or a result of fitting.
calc(5)	calcZ	Calculate χ and ϕ for zone feature.
calc(7,0)	calcD	Calculate direct lattice from reciprocal parameters.
calc(7,1)	calcR	Calculate reciprocal lattice from direct parameters.
calc(8)	calcE	Calculate λ for current monochromator positions.
calc(9)	calcM	Calculate monochromator position for current λ.
calc(10)	_begUB	Initialize sums for fitting **UB**.
calc(11)	_addUB	Add a reflection to fitting sums.
calc(12)	_fitUB	Fit **UB**.
calc(13)	calcL	Calculate lattice parameters from **UB**.

Four-Circle Macros

The macros below are used in setting the parameters and selecting modes.

Name	Arguments	Description
`setmode`	1 optional	Choose geometry mode.
`setsector`	1 optional	Choose sector.
`setlat`	6 optional	Set lattice parameters.
`setaz`	3 optional	Set azimuthal reference vector.
`setmono`	2 optional	Set beam-line monochromator parameters.
`or0`	3 optional	Set primary orientation reflection.
`or1`	3 optional	Set secondary orientation reflection.
`setor0`	none	Alternative to set primary orientation reflection.
`setor1`	none	Alternative to set secondary orientation reflection.
`or_swap`	none	Swap values for primary and secondary vectors.
`freeze`	none	Turn on *freeze* mode.
`unfreeze`	none	Turn *freeze* mode off.
`cz`	6	Calculate zone.
`sz`	6	Set zone parameters.
`mz`	6	Move to zone.
`cuts`	2/6 optional	Show or set cut points.

Most of the macros with optional arguments will prompt for the required values if invoked without arguments. Using any of these macros to change a parameter will produce a comment on the printer and in the data file.

The `or0` macro is a typical example of these macros.

```
14.FOURC> prdef or0
def or0 '{
      local   _1 _2 _3

      if ($# == 3) {
              _1 = $1; _2 = $2; _3 = $3
      } else if ($# == 0) {
              print "\nEnter primary-reflection HKL coordinates:"
              _1 = getval(" H", g_h0)
              _2 = getval(" K", g_k0)
              _3 = getval(" L", g_l0)
      } else {
              print "Usage:  or0  or  or0 H K L"
              exit
      }
      waitmove; get_angles
      gpset _1       g_h0       # gpset documents the change
      gpset _2       g_k0
```

```
        gpset _3       g_l0
        gpset A[mA[0]] g_u00
        gpset A[mA[1]] g_u01
        gpset A[mA[2]] g_u02
        if (_numgeo > 3) { gpset A[mA[3]] g_u03 }
        if (_numgeo > 4) { gpset A[mA[4]] g_u04 }
        if (_numgeo > 5) { gpset A[mA[5]] g_u05 }
        if (_numgeo > 6) { gpset A[mA[6]] g_u06 }
        gpset LAMBDA g_lambda0
        calcG
}'

15.FOURC>
```

Zone Macros

Zone mode is controlled with the `cz` (calculate zone), `sz` (set zone) and `mz` (move zone) macros. Given two Bragg reflections, `cz` will calculate and display the values of χ and ϕ necessary to put both of these reflections in the scattering plane. To find the angles needed to put $(0, 0, 2)$ and $(0, 2, 2)$ in the scattering plane, type

```
15.FOURC> cz 0 0 2 0 2 2
Chi = 45  Phi = 90

16.FOURC>
```

Once appropriate values of χ and ϕ have been calculated, the scattering plane can be set using the `pl` (plane) macro, which moves the 2θ and θ motors together,

```
16.FOURC> pl 45 90

17.FOURC>
```

Alternatively, you can use the `mz` macro, which calculates the necessary χ and ϕ, moves there, sets **zone** mode, if not already in it, and saves the values of the zone vectors in the `G[]` geometry parameter array.

```
17.FOURC> mz 0 0 2 0 2 2

18.FOURC> p A[chi], A[phi]
45 90

19.FOURC>
```

The `sz` macro calculates and displays the χ and ϕ values, sets **zone** mode, if not already in it, saves the values of the zone vectors, sets the frozen values of **zone**-mode χ and ϕ, but does not move the diffractometer.

The cz, sz and mz macros make use of the Z[] array variables to pass the zone vectors to the geometry code.

```
19.FOURC> prdef cz
def cz '
      if ($# != 6) {
              eprint "Usage:  cz h0 k0 l0 h1 k1 l1"
              exit
      }
      Z[0]=$1; Z[1]=$2; Z[2]=$3; Z[3]=$4; Z[4]=$5; Z[5]=$6
      calcZ
      printf("Chi = %g  Phi = %g\n", A[chi], A[phi])
      waitmove; get_angles; calcHKL
'

20.FOURC>
```

Least-Squares Refinement of Lattice Parameters

In the previous sections, the procedure described for determining the orientation matrix required a knowledge of the lattice parameters of the crystal and the position of two reflections. When such information is unknown, the orientation matrix can be fit to an unlimited number of observed peak positions using a least-squares procedure.[7] Lattice parameters derived from the fitted orientation matrix can then be calculated, although such lattice parameters are not constrained to exhibit any symmetry whatsoever.

In spec's implementation of least square refinement, three macros are used to create a file that contains the observed peak positions. That file is eventually run as a command file, and the least squares analysis is performed.

[7] J. Matthews and R. L. Walker, *Mathematical Methods of Physics*, (Benjamin, Menlo Park, 1970), p. 391.

The `reflex_beg` macro initializes the reflections file:

```
global REFLEX   # Variable for file name
# Open the file, save old one as .bak and write header
def reflex_beg '{
      if ("$1" == "0") {
              if (REFLEX == "")
                      REFLEX = "reflex"
              REFLEX = getsval("Reflections file", REFLEX)
      } else
              REFLEX = "$1"

      if (open(REFLEX))
              exit
      close(REFLEX)
      if (file_info(REFLEX, "-s"))
              unix(sprintf("mv %s %s.bak", REFLEX, REFLEX))
      fprintf(REFLEX,"# %s\n\n_begUB\n\n",date())
}'
```

The `reflex` macro adds lines to the file that contain the (H, K, L) and $(2\theta, \theta, \chi, \phi)$ of each reflection:

```
# Add reflection to the file
def reflex '
      if ($# != 3) {
              print "Usage:  reflex H K L"
              exit
      }
      if (REFLEX == "") {
              REFLEX = getsval("Reflections file", "reflex")
              if (REFLEX == "")
                      exit
      }
      waitmove; get_angles; calcHKL
      fprintf(REFLEX,"H = %g;  K = %g;  L = %g\n",$1,$2,$3)
      {
        local i

        for (i=0; i<_numgeo; i++)
              fprintf(REFLEX,"A[%s]=%9.4f;  ",motor_mne(mA[i]),A[mA[i]])
        fprintf(REFLEX,"\n")
      }
      fprintf(REFLEX,"# counts = %g\n", S[DET])
      fprintf(REFLEX,"_addUB\n\n")
'
```

Finally, the `reflex_end` macro puts the proper trailer on the file:

```
# Add trailer to file
def reflex_end '
      fprintf(REFLEX,"_fitUB\n")
      printf("Type \"qdo %s\" to calculate new orientation matrix\n",\
              REFLEX)
'
```

When you are ready to calculate the orientation matrix, simply run the command file. There is no limit to the number of reflections contained in the file. You can also edit the file by hand to add or subtract reflections.

The calculated orientation matrix will remain valid until you type `calcG`, or invoke a macro that calls `calcG`. Those macros are `or0`, `or1`, `or_swap` and `setlat`.

The six original lattice parameters will remain unchanged when using the above macros to fit the orientation matrix to the reflections. The `calcL` macro can be invoked to calculate the lattice parameters derived from the fitted orientation matrix and place their values in the appropriate elements of the parameter array. The old lattice parameters will be lost.

Here is a sketch of the commands you use to perform the least squares refinement of the lattice parameters.

```
20.FOURC> reflex_beg
Reflections file (reflex)? <return>

21.FOURC> (find and move to a reflection ...)

22.FOURC> reflex 2 2 0

23.FOURC> (find and move to another reflection ...)

24.FOURC> reflex 2 0 2

25.FOURC> ...

26.FOURC> reflex_end
Type "qdo reflex" to recalculate orientation matrix.

27.FOURC> qdo reflex
Opened command file ‘reflex’ at level 1.

28.FOURC>
```

At least three reflections must be used for the least-squares fitting to work.

To calculate the new lattice parameters, use the `calcL` macro:

```
28.FOURC> calcL

29.FOURC> pa

Four-Circle Geometry, Omega fixed (mode 1)
Frozen values:  Omega = 5
Sector 0

  Primary Reflection (at lambda 1.54):
          tth th chi phi = 74.212 42.106 89.898 -80.2141
                   H K L = 2 2 0

  Secondary Reflection (at lambda 1.54):
          tth th chi phi = 24.631 17.3155 135.315 -82.9029
                   H K L = 0 1 0

  Lattice Constants (lengths / angles):
              real space = 4.127 4.123 4.111 / 90.04 89.98 90.12
        reciprocal space = 1.523 1.524 1.528 / 89.96 90.02 89.88

  Azimuthal Reference:
                   H K L = 0 0 1

 Cut Points:
      tth   th  chi  phi
     -180 -180 -180 -180

30.FOURC>
```

Here is a typical reflections file created by the above macros:

```
# Wed Jan 31 21:55:01 1990

_begUB

H = 2;  K = 2;  L = 0
A[tth]=  63.8185;  A[th]=  36.9320;  A[chi]=  89.8765;  A[phi]= -80.0815
# counts = 3456
_addUB

H = 2;  K = 0;  L = 2
A[tth]=  63.8335;  A[th]=  36.8920;  A[chi]= 145.4185;  A[phi]=  42.8145
# counts = 6345
_addUB

H = 0;  K = 2;  L = -1
A[tth]=  49.4100;  A[th]=  29.6725;  A[chi]=  35.8180;  A[phi]=  45.9070
# counts = 5634
_addUB

H = 0;  K = 2;  L = -1
```

```
A[tth]=  49.3550;  A[th]=  29.7225;  A[chi]=  35.9380;  A[phi]=  45.9070
# counts = 4563
_addUB

_fitUB
```

Within this file, the `calc()` function codes defined by the macros `_begUB`, `_addUB` and `_fitUB` are used to access the C code that performs the least squares operations.

ADMINISTRATOR'S GUIDE

Introduction

The first section of this guide outlines the procedure for installing spec on your computer. Later sections describe the format of some of the installed files.

Quick Install

For those who need little explanation, here are minimal installations instructions based on the customary configuration:

```
# First time, create a specadm user account, then
cd ~specadm
mkdir spec6.05.03              # choose name based on release
cd spec6.05.03
tar xvf .../spec_XXX.tar       # use pathname of distribution
# If an update, start with last version's parameters
cp ../spec_YYY/install_data .  # use pathname of previous distribution
./Install                      # as root
```

The *Install* program will display the current installation parameters and prompt for changes. Use the command `./Install -d` to use the parameters from an existing *install_data* file, skipping the interactive prompts. Once the software is installed, run the spec executable and type `config` to invoke the hardware configuration editor.

For those needing more detailed instructions, read on.

Steps For Installing spec

To install spec on your computer, be sure you have the software development tools available. spec's requirements include the *make* utility, a compatible C compiler, and compatible runtime libraries for linking.

Before installing spec for the first time, you need to make several decisions:

Decide who will own the files. Most sites create a special user account, usually with the name *specadm*, that has a home directory used to hold the spec distribution files and is the user account assigned to own the installed spec files. Having a special *specadm* account allows you or other users to configure and edit spec files without invoking super-user powers, thus lowering the risk of making catastrophic errors.

Decide where to put the distribution. The distribution files are what come directly from CSS, usually as a *tar* file sent via internet. (In olden times, the files were on a floppy disk, a magnetic tape or a CD-ROM.) These distribution files need to be

extracted into a distribution directory that serves as a “staging” area for the installation. The files needed while running spec will be copied elsewhere during installation, so the distribution directory need not be always accessible to the spec users. However, it is usually advised to keep the distribution files available for reference. You will likely receive updated distributions from time to time, and it is a good idea to keep each distribution in a separate directory. The usual choice is to put the distribution in the *specadm* home directory in a subdirectory named after the spec version number, such as *~specadm/spec6.05.03*.

Decide where the installed files will go. Two directories are needed for the installed files. One is a directory for the executable programs. Most sites choose */usr/local/bin*. This directory needs to be in each spec user's search path. The second directory is for spec's auxiliary files. The usual choice is */usr/local/lib/spec.d*.

Once these decisions are made and you have created a spec administrator user account (if used) and the directories mentioned above (if needed), you are ready to perform the installation. In brief, to install spec you:

- extract the distribution from the supplied media or *tar* file,
- run the installation program to install the spec files,
- enter the hardware configuration for your particular experimental set-up.

In addition to the information presented below, be sure to consult any *README* files in the spec distribution directories for up-to-date information on installation procedures.

Extracting the Distribution

If you have made a spec administrator's account, you should become that user (or the root user), either by logging in or by using the set-user id command `su specadm`. You can then change to the spec administrator's home directory and use it as a place to hold the distribution files.

Make a subdirectory to receive the current version of spec. If the distribution is numbered release 6.05.03, you might make a directory called *spec6.05.03* using the command `mkdir spec6.05.03`. Change to the new directory with `cd spec6.05.03`.

The distribution will be in *tar* format. Usually the distribution is obtained as a *tar* file via internet. If the distribution arrives via magnetic media or a CD-ROM, the command to extract the *tar* file should be printed on the distribution label. For a *tar* file, the command to extract the files is

```
tar xvf specdist.tar
```

Installing the spec Program Files

To install the spec files, particularly for a first time installation, you may need to install as root. Use the `su` command to gain super-user privileges. If you are updating from previous distributions, you can copy the most recent *install_data* file containing your default installation parameters to the new distribution directory. Then, from the current spec distribution directory, type `./Install` to run the installation program. If updating an existing installation and using the same installation parameters as before, type `./Install -d` (for default) to skip the interactive portion described next.

The *Install* program will first indicate the current installation parameters. You may either accept those or enter new parameters. When entering new parameters, the default response to each question is given in parenthesis. Most questions present a number of choices. You can either type the number of the choice or you can type out the literal selection. If one of the options is the word “other”, such as for the name of a directory, you can directly type your selection when prompted. For most questions, the first choice listed is probably the best response. For example,

```
Choices for binaries directory are:

      1) /usr/local/bin
      2) /usr/local
      3) /usr/local/spec/bin
      4) /u/bin
      5) /LocalLibrary/Spec/bin
      6) other

Choose binaries directory (/usr/local/bin)?
```

Entering a single minus sign (`-`) will move back to the question for the previous parameter, allowing you to enter a different value.

The installation questions ask for the following parameters:

geometry – Selects from the supported diffractometer configurations.

installed name – Selects a name for the installed program. For configurations with special geometry code, the first four letters of the name must match the first four letters of the geometry configuration. Thus, *fourcL* and *fourcR* might be the names for the left- and right-hand sides of a rotating anode lab with two four-circle diffractometers.

additional geometries – Asks if you want to enter more combinations of the previous two items.

file ownership – Selects the name of the owner of the spec files.

binaries directory – Selects where the programs that users run directly from the shell will go. (It's better to put spec files in some place other than the standard */bin* or */usr/bin*, in order to be able to distinguish files that are standard UNIX from those that have been added locally.) This directory should be in each spec user's search path.

auxiliary directory – Selects where spec puts its auxiliary files. This directory is preferably on a local disk, as it contains files that are frequently updated, particularly the motor *settings* file.

TACO library directory – Gives the location of the TACO device server libraries if installing on a TACO device server platform. Gives the location of the TACO device server libraries, if using. There should be a subdirectory named "lib" in this directory. If using TANGO, enter "no" here and configure TANGO on next screen.

TANGO library directory – Gives the location of the TANGO device server libraries.

EPICS library directory – Gives the location of the EPICS channel access libraries if installing on an EPICS platform. The location can be the directory that has a subdirectory named *base*, the path name to the base directory that has a subdirectory named *lib*, or the complete path to the directory containing the EPICS libraries for this platform.

HDF5 library directory – Gives the location of the HDF5 libraries. Enter a value to enable spec's HDF (Hierarchical Data Format) data commands. Including HDF5 support with the default static libraries nearly doubles the size of the spec executable image.

config file permissions – Selects who can change the hardware configuration file. On a low-security site, select the first choice, which lets all spec users make changes as needed. See security notes on page 237 for additional considerations.

command-line editing options – Selects an alternative library that will be linked with spec to provide a more powerful history recall syntax than the minimal built-

in spec history mechanism. The alternative library also includes command line editing features. spec includes a prebuilt version of the Berkeley *libedit* library.

Extra compiler flags – Allows you to add extra compiler flags for both compiling the site-dependent source files and linking.

Extra object files – Allows you to specify extra site-dependent object files to include when linking the spec executable.

Extra library flags – Allows you to specify extra site-dependent libraries to be searched during the link phase when producing the spec executable.

After answering the questions, the installation should then continue automatically, producing output similar to the following:

```
This program will install version 6.05.03 of the spec package.
Type "Install -" to see invocation options.

Checking if u_hook.c needs compiling ...
Compiling u_hook.c ...
Compiling u_hdw.c ...
Checking if geo_fourc.c needs compiling ...
Checking if spec needs to be linked ...
Linking spec ...
Installing config auxiliary files ...

Installing fourc ...
cp fourc /usr/local/bin/fourc
 Checking "fourc" config and settings file permissions ...
Installing macros ...
 Installing /usr/local/lib/spec.d/standard.mac ...
 Installing /usr/local/lib/spec.d/four.mac ...

Installing the high res graphics filters ...
Installing help files ...
 Clearing out old help files ...
 Making the "help" help file ...
 Copying help files ...
 Changing ownership of help files to gerry ...
Installing the chelp program ...
Installing the show_state program ...
Installing data_pipe auxiliary files ...
 Building data_pipe utility "dp_cplot" ...
Installing auxiliary "include" files ...
cp spec_shm.h /usr/local/lib/spec.d/include
cp spec_server.h /usr/local/lib/spec.d/include
Installing the "showscans" package ...
cp show.awk /usr/local/lib/spec.d
cp scans /usr/local/bin
Installing the "contents" program ...
```

```
cp contents /usr/local/bin
Installing the "tidy_spec" program ...
cp tidy_spec /usr/local/bin
Installing the "wiz_passwd" program ...
cp wiz_passwd /usr/local/bin
Installing splot utility ...
 Clearing out old splot files ...
mkdir /usr/local/lib/spec.d/splot
 Copying splot files ...
 Changing ownership of splot files to gerry ...
Creating spec.conf file ...

Done with spec Install!
```

If you change certain parameters that require relinking spec and don't see the `Linking spec ...` message when rerunning the *Install* program, simply remove the *spec* file and run *Install* again.

Selecting the Hardware Configuration

The final step in the initial spec installation is to set the hardware configuration specific to your site. You can do that either by starting the spec program and typing `config`, which is macro that executes the configuration editor, or by running the configuration editor directly. For the latter, first change to */usr/local/lib/spec.d* (or to the auxiliary file directory specified when you did the installation). If you are installing the normal four-circle version of the program, type the command `edconf fourc`. If you installed a different geometry, give the name of that geometry as the argument. This command starts a spread-sheet styled program that lets you select motor parameters, devices names, etc.

Refer to the notes on the configuration editor that follow for instructions on using *edconf*.

Adding Site-Dependent Help Files

If a file named *.local* in the *spec_help* subdirectory of the auxiliary file directory exists and contains a list of file names, those names will be added to the topics contained in the *help* help file when spec is installed.

When spec starts up, the help file *news*, which is provided by CSS, and the file *local*, if it exists, will be displayed. The help file format is described on page 81 in the *Reference Manual*.

Adding Site-Dependent C Code

This step applies only to sophisticated end users of spec who understand the C language and need to customize spec for specific, site-dependent uses. Most readers can skip to the next section. Note also, local code can be accessed using the *data-pipe* facility explained on page 124 in the *Reference Manual*.

spec has provisions for end users to add their own C code to the program. User-added code is accessed using the built-in `calc()` function. If you wish to incorporate non-standard calculations within the spec program, you can do so by adding hooks for the code in the *u_hook.c* source file. C code that you add should, in general, be limited to calculations. You should avoid I/O, signal catching, etc. Consult CSS for specific information about what is appropriate for including in user-added C code. The *geo_*.c* files in the standard spec distribution that contain the X-ray diffractometer geometry code are examples of site-dependent code.

Within *u_hook.c* there is a routine called `init_calc()`. This routine is called once when spec starts up. Within `init_calc()`, calls to the routine

```
ins_calc(int num, int (*func)())
```

insert the C routine *func* in a table of functions. These functions are called when `calc(`*num*`)` or `calc(`*num*`, `*arg*`)` is typed as a command to spec. The routine *func*`()` should be specified as either

```
func(int num)
```

or

```
func(int num, double arg)
```

depending on whether `calc()` is to be invoked with one or two arguments.

Any return value from *func*`()` is ignored. However, you can have the `calc()` routine return a value by assigning a number to the variable

```
extern  double  calc_return;
```

in *func*`()`. If no explicit assignment is made to `calc_return`, `calc()` returns zero.

The argument *num* can be from 0 to 63, but must be chosen not to conflict with any of the other `ins_calc()` entries already existing in *u_hook.c*.

You can also create built-in arrays of double precision, floating point numbers that can be used to communicate values between your C code and the user of the program.

The routine

```
ins_asym(double **x, int n, char *s)
```

inserts the array *x* consisting of *n* elements into the table of built-in symbols. The character pointer *s* points to a string containing the name used to refer to the array from spec command level. For example,

```
#define N_PARAM 28
double  *gparam[N_PARAM];
init_calc() {
      ...
      ins_asym(gparam, N_PARAM, "G");
      ...
}
```

inserts the 28-element array referred to as `G[]` into the program. Since the array `gparam[]` is an array of pointers, you must use the indirection operator (`*`) when referring to the values of the floating point numbers in your C code, as in

```
...
*gparam[3] = 1.54;
...
if (*gparam[2] == 0)
        ...
```

If you make any changes to *u_hook.c*, you must relink and reinstall the spec binary.

Updating spec

spec updates are normally extracted into a directory named after the spec version number, as in */usr/specadm/spec6.05.03* for version 6.05.03 of the software. The instructions above for extracting the distribution and installing the files also apply to updates.

Existing *settings* and *config* files from a previous installation will not be disturbed during an update. Each user's state file will also remain intact, although it is recommended that users either start out fresh (by typing `fourc -f`) or run the macro command `newmac` after new versions are installed to incorporate improvements to the standard macros in their state files. Occasionally, new versions will not be compatible with previous state files, and spec will automatically throw out the old state files, print a message and start fresh anyway. Use the *tidy_spec* utility to clean out old state files to free up disk space, especially if the old state files are obsolete with the newer version of spec.

The help file *changes* will contain summaries of the significant bug fixes and improvements to spec included in the update.

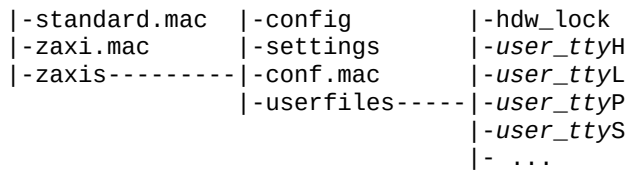
Installed Files

File Hierarchy

After installation and site configuration on a computer running, for example, both a four-circle and a *z*-axis diffractometer, the spec file hierarchies would appear something like the following,

```
                        |-chelp
                        |-contents
                        |-dpmake
                        |-fourc
/usr/local/bin----------|-scans
                        |-showscans
                        |-show_state
                        |-specfile
                        |-splot
                        |-tidy_spec
                        |-wiz_passwd
                        |-zaxis

                        |-README
                        |-data_pipe-----|-data_pipe.mak
                        |               |-data_pipe.o
                        |               |-pipe_test.c
                        |               |-user_pipe.h
                        |
                        |-edconf        |-config        |-hdw_lock
                        |-four.mac      |-settings      |-user_ttyH
                        |-fourc---------|-conf.mac      |-user_ttyL
                        |               |-userfiles-----|-user_ttyP
                        |                                 |-user_ttyS
                        |-hgr-----------|-x11filt       |- ...
                        |
                        |-include-------|-spec_server.h
                        |               |-spec_shm.h
/usr/local/lib/spec.d-|-passwd
                        |-show.awk
                        |-site.mac      |-help          |-angles
                        |-site_f.mac    |-help.tmac     |-ackno
                        |-spec.conf     |-help_man------|-changes
                        |-spec_help-----|               |- ...
                        |               |
                        |               |               |-angles
                        |               |-help_pre------|-ackno
                        |               |               |-changes
                        |               |               |- ...
                        |               |-news
                        |               |-nroff_help.tmac
                        |               |-old_help.tmac
                        |               |-rst2man
                        |
                        |-splot---------|-many splot files
                        |
```



where */usr/local/bin* is the installation directory, configured as `INSDIR` in the *Makefile*, and */usr/local/lib/spec.d* is `SPECD`, the auxiliary file directory. Of the programs installed in */usr/local/bin*, *contents* and *showscans* are described in the *User Manual*, while the *camac* utility program is described below. The *chelp* utility is a standalone help file viewer that allows browsing of the spec help files without having to run spec.

The subdirectory *fourc* contains the files specific to the four-circle diffractometer. The name *fourc* matches the name by which the program is invoked. The first four letters of the name determine the geometry configuration. If a single computer is to control two spectrometers, they could be called *fourcL* and *fourcR*, and the *Install* program would create separate subdirectories called by those names for each.

Within the diffractometer directory is the associated configuration file, *config*, which specifies the hardware and the motor parameters to be used. Also associated with each diffractometer is the *settings* file that tracks changes in the motor position and limit settings. The *edconf* program can be used to to modify the contents of these two files.

The subdirectory *userfiles* contains each user's state files on a per terminal basis. These files allow the user to exit spec and restart at a later time, retaining macro definitions, variable assignments, etc. The spec administrator may, from time to time, delete old state files for users not expected back again, especially if disk space is a problem. The *tidy_spec* program reports on the disk usage of all the *userfiles* directorys and provides options for removing files by age, user, tty or spectrometer geometry. Type `tidy_spec -` from the shell for usage options.

Accessing Protected I/O Ports On PC Platforms Running *linux*

In order to enable spec's built-in support for certain instrument control and data acquisition devices, spec needs access to resources that are generally off limits to normal users. For some hardware, spec needs access to */dev/mem* to map PCI/PCIe memory space to the spec process ("dac override" capability, which overrides discretionary access control, that is, bypasses file access permissions). On some recent *Linux* platforms, PCI/PCIe cards used by spec are disabled on boot by the *Linux* kernel, and spec needs escalated privileges to enable the cards ("sys admin" capability). If *udev* rules are not in place to set permissions for USB devices supported by spec,

spec will use escalated permissions to open the device nodes (also "dac override" capability). For some older PC cards, spec needs direct access to I/O ports, which it gains using the *iopl()* system call ("sys rawio" capability).

Up until spec release 6.05.01, access was achieved by making spec a set-user-id root executable. With current releases, the *Linux capabilities* facility is used. If the *Linux* platform lacks the *setcap* command, spec will use set-user-id root mode, as before. The command to set capabilities is the following:

```
setcap "cap_dac_override=ep cap_sys_rawio=ep cap_sys_admin=ep" spec
```

The command to make spec set-user-id root is:

```
chown root spec && chmod u+s spec
```

The spec *Install* script will automatically execute one or the other of the above commands on *Linux* platforms if the script is run by the root user.

To protect the system whether using *setcap* or set-user-id mode, spec turns off all special privileges immediately on program start up and only enables the capabilities around the few lines of code used to gain access to the otherwise unavailable resources.

The Configuration Editor

The *edconf* program is the primary means for maintaining the hardware configuration. When running spec, *edconf* is usually run by invoking the `config` macro. Without arguments, *edconf* will use the *config* and *settings* files in the current directory. If given a directory name as an argument, it will use the files in that directory. If invoked with the **`-s`** flag, *edconf* will run in *simulate* mode, allowing you to view but not modify the files. If you do not have write permission for the *config* file, *edconf* will automatically run in *simulate* mode.

To get a list of the available commands while running *edconf*, type a question mark (**`?`**). Note, the available commands are different on different screens. The following commands are available:

Arrow keys	Move around.
`h j k l`	Move left, down, up, right (just like in the *vi* editor).
<return>	Enter data or move down one row.
<space>	Move right.
`'` or **`"`**	Enter to change string-valued cell (e.g. motor names).
`^A`	Edit value from start of string

`^E`	Edit value from end of string
`+ - > <`	Step through list of choices (if <> appears in label).
`r`	**R**eread *settings* and *config* files.
`w`	**W**rite *settings* and *config* files.
`R`	**R**ead from backup *settings* and *config* files.
`c`	**C**hange to next screen.
`M`	Change to **M**otor screen.
`m`	Step through **m**otor parameter screens.
`C`	Change to **C**AMAC screen, page through additional CAMAC screens.
`I`	Change to **I**nterfaces screen.
`D`	Change to **D**evices screen.
`A`	Change to MCA/Image **A**cquisition screen.
`S`	Change to **S**calers (counters) screen.
`s`	Switch between main and optional parameter **s**caler screens.
`p`	Change to and from custom optional **p**arameters screen.
`a`	**A**ppend an entry on the custom parameter screen.
`i`	**I**nsert a motor at current position or an entry on the custom parameter screen.
`d`	**D**elete the motor at current position or an entry on the custom parameter screen.
`G`	Toggle all-motors mode with linked **g**eometry configurations.
`^F`	Scroll forward through motors on the motor screen, counters on the scalers screen, configured controllers on the devices screen, module choices on the CAMAC screen, items on drop-down menus, etc.
`^B`	Scroll backwards, as above.
`^G`	Go to first item in scrollable list.
`^D`	Blank out optional motor parameter fields.
`^L`	Refresh the screen.
`^W`	Gain wizard access to set protections.
`H`	Print help information for current screen.
`?`	Print command help window.
`q`	**Q**uit.
`^C`	Exit.

There are five types of data cells in the configuration spread sheet. For number-valued cells, simply enter the number value when the cell is highlighted. For string-valued cells, such as motor names, you must first type a single quote (`'`) before entering the string. For `YES/NO` cells, type a `y` or an `n`. The fourth type of cell has the

characters <> before the cell label. For these cells, use the **+**, **-**, **<**, or **>** keys to step through the possible choices. For all types of cells, the <*return*> key enters the values. The fifth type of cell is used for entering motor unit/[module]/channel values. For these cells, type just the channel number for unit zero, or type two or three numbers separated by a literal **/**.

Use the w command to write out the configuration, and use **^C** or q to exit the program.

The Settings File

The *settings* file is a binary file that contains consecutive data for each motor according to the following structure:

```
struct  sav_mot {
      long    sm_pos;         /* Current dial position */
      float   sm_off;         /* Current user/dial offset */
      double  sm_low;         /* Software low limit */
      double  sm_high;        /* Software high limit */
};
```

The *settings* file must have write permission for everybody who runs spec, as it is updated every time someone moves a motor or changes an offset or limit. When spec starts out, it checks to see if there is a *settings* file and creates an empty one if there isn't. spec creates a software lock on the *settings* file using the *lockf()* library call. The lock prevents another instance of spec opening the file for writing.

The Config File

The *config* file is an ASCII file that describes the diffractometer hardware configuration. Although the *config* file can be edited by hand, you will be safer using the *edconf* program to make modifications as *edconf* insures the *config* file obeys the structuring rules required by spec.

Comment lines in the *config* file begin with a #. Other lines contain key words specifying devices, CAMAC slots or motor and counter parameters. Key words are followed by a space-delimited equals sign and one or more parameters.

The *config_adm* help file contains up-to-date information about currently recognized key words and supported hardware devices. The *Hardware Reference* section of the manual describes some of the specific hardware devices recognized by spec and indicates the *config* file syntax required to specify each device.

CAMAC slot assignments in the *config* file consist of a module code on the left and a slot number on the right. For example,

```
CA_KS3610 = 2
```

tells the program a Kinetic Systems 3610 hex scaler is in slot 2.

The following modules names are recognized by spec. More than one of the modules marked with an asterisk are allowed. Append _# to number modules consecutively, where # is 0, 1, 2, etc.

CA_BR5302*	BiRa 5302 ADC as counters
CA_DSP2190	DSP Technology MCS Averager
CA_DXP*	XIA DXP MCA
CA_E250*	DSP E250 12-Bit D/A as Motor Controller
CA_E500*	DSP Technology E500A Stepper Motor Controller
CA_IOM1	BiRa 2601 I/O For E500 Multiplexing
CA_IOM2	F16,A0 I/O For E500 Multiplexing
CA_IOM3	F16,A1 I/O For E500 Multiplexing
CA_IO*	Any module to be accessed with F codes of 0 or 16
CA_KS3112*	Kinetic Systems 3112 D to A (as motor controller)
CA_KS3116*	Kinetic Systems 3116 16-Bit D/A as Motor Controller
CA_KS3195*	Kinetic Systems 3195 16-Bit D/A as Motor Controller
CA_KS3388	Kinetic Systems 3388 GPIB interface
CA_KS3512*	Kinetic Systems 3512/14 ADC as counters
CA_KS3610	Kinetic Systems 3610 6-Channel, 50 MHz Counter
CA_KS3640C*	Kinetic Systems 3640 Up/Down Counter as Counter
CA_KS3640M*	Kinetic Systems 3640 Up/Down Counter (for SMC's)
CA_KS3640T	Kinetic Systems 3640 Up/Down Counter as Timer
CA_KS3655	Kinetic Systems 3655 8-Channel Timing Pulse Generator
CA_KS3929	Kinetic Systems 3929 SCSI Crate Controller
CA_KS3929_HP	Kinetic Systems 3929 SCSI to CAMAC on HP
CA_LC2301	LeCroy 2301 interface for QVT MCA
CA_LC3512	LeCroy 3512 Spectroscopy ADC
CA_LC3521	LeCroy 3521A Multichannel Scaling
CA_LC3588	LeCroy 3588 Multichannel Scaler
CA_LC4434*	LeCroy 4434 32-Channel Scaler
CA_LC8206	LeCroy MM8206A Histogramming Memory
CA_QS450*	DSP Technology QS-450 4-Channel Counter
CA_RTC018	DSP Technology RTC-018 Real Time Clock
CA_RTC018M	DSP RTC-018 2nd Unit For Monitor

CA_SMC*	Joerger Stepper Motor Controller SMC-L or SMC-24
CA_TS201	DSP Technology TS-201 Dual Timer/Scaler

Motor Parameters

Motor parameter assignment consists of key words of the form MOT000, MOT001, ..., followed by 11 values. The MOT key words must be numbered consecutively starting at zero. The values are:

1. Controller type (E500, SMC, OMS, ...)
2. Steps per unit (degrees, mm, ...) (sign changes direction of motion)
3. Sign between user and dial units (+1 or −1)
4. Steady state rate (Hz) (must be positive)
5. Base rate (Hz) (must be positive) (also used as backlash rate)
6. Steps for backlash (sign changes direction of motion)
7. Acceleration time (msec)
8. Not used
9. Motor flags in hexadecimal (protection, units, etc.)
10. Motor mnemonic (th, phi, sl1, ...)
11. Motor name (Theta, Phi, Slit 1, unused, ...)

An example is:

```
# Motor  cntrl steps sign slew base backl accel nada  flags mne  name
MOT000 = E500 −2000    1 2000  200    50   125    0  0x003 tth  Two Theta
```

Valid controller types currently include:

18011	Oriel Encoder Mike Controller 18011
18092	Oriel Encoder Mike 18092
ANORAD	Anorad I-Series Controller
ANORAD_E	As above, but with encoder
CM3000	Compumotor 3000
CM4000	Compumotor 4000
CMSX	Compumoter SX
CMSX_E	As above, but with encoder
DAC_B12	PC DAC 12-Bit D/A (binary output)
DAC_B16	PC DAC 16-Bit D/A (binary output)
DAC_T12	PC DAC 12-Bit D/A (two's complement)
DAC_T16	PC DAC 16-Bit D/A (two's complement)
E250	DSP E250 12-Bit D/A as Motor Controller
E500	DSP Technology E500A

E500_M	As above, but with multiplexor
ECB_M	RISO ECB Motors
EPICS_M1	EPICS using spec's *config* motor parameters
EPICS_M2	EPICS using EPICS' database motor parameters
ES_OMS	ESRF using VME OMS
ES_VPAP	ESRF using VME Vpap
GALIL	Galil DMC-1000 PC Board
HLV544	Highland Technology VME V544
HUB9000	Huber 9000
IP28	Micro-Controle IP28
ITL09	Micro-Controle ITL09
ITL09_E	As above, but with encoder
IXE	Phytron IXE
KS3112	Kinetic Systems 3112 12-Bit D/A
KS3116	Kinetic Systems 3116 16-Bit D/A
KS3195	Kinetic SystemsS 3195 16-Bit D/A
MAXE	ESRF VME MAXE Motor Controller
MAXE_E	As above, but with encoder
MAXE_S	As above, but with servo
MAXE_DC	ESRF VME DC Motor Controller
MC4	Klinger MC4
MCB	Advanced Control Systems MCB
MCU	Advanced Control Systems MCU
MCU_E	As above, but with encoder
MCU_H	As above, with with Heidenhain encoder
MCU_O	As above, but with old PROMs
MM2000	Newport MM2000/3000
MM2000_E	As above, but with encoder
MM2500	Newport MM2500
MM2500_E	As above, but with encoder
MM4000	Newport MM4000/4005
MM4000_E	As above, but with encoder
MMC32	NSLS homemade
MURR	Missouri Research Reactor Motors
MURR_E	As above, but with encoder
NF8732	New Focus Picomotor 8732
NONE	Pseudo controller
NSK	NSK Motor Controller
NT2400	Laboratory Equipment Corporation Model
OMS	Oregon Micro Systems PCX/34/38/39/48
OMS_E	As above, but with encoder

PI	PI DC Motor Controller
PM500	Newport PM500
RIGAKU	Rigaku RINT-2000 Motor Controller
SCIPE_A	SCIPE Actuator Device
SIX19	Micro-Controle SIX19
SMC	Joerger Single Motor Controller
SPI8	Advanced Control System SPI-8
TSUJI	Tsuji PM16C-02N
XIAHSC	XIA HSC-1
XRGCI_M	Inel XRGCI motor controller/timer

Field 2, the steps per unit, may be non-integral, and the units can be in degrees, millimeters or whatever. The rest of the numeric fields must be integral. The motor names should be kept to nine characters or less, as the standard macros truncate them to fit a nine-character field when printing them out.

Field 8 is reserved.

Field 9, the flags field, contains several kinds of information. The lowest order two bits are used to enable particular operations on the selected motor. If bit 0 is set, the user can move the motor. If bit 1 is set, the user can change the software limits of the motor. Bits 2 and 3 are used by the *edconf* program to prevent users from changing certain configuration information. Bits 8 through 12 are used with the shared *config* file feature described below.

Optional motor parameters appear on lines following the MOT keywords and are of the form

```
MOTPAR:dcgain = 1500
```

Each MOTPAR refers to the immediately preceding motor. Possible parameters are:

```
encoder_step_size
step_mode
slop
home_slew_rate
home_base_rate
home_acceleration
dc_dead_band
dc_settle_time
dc_gain
dc_dynamic_gain
dc_damping_constant
dc_integration_constant
dc_integration_limit
dc_following_error
dc_sampling_interval
deceleration
```

```
read_mode
torque
misc_par_1
misc_par_2
misc_par_3
misc_par_4
misc_par_5
misc_par_6
```

Most parameters are not used by most motor controllers.

Linked Configurations

An installation such as a synchrotron beamline uses many motors with most associated with beamline control. Spectrometers used for particular experiments have motors that aren't used in other experiments. To avoid having to merge the motor configurations and settings from one set of files to another when the spectrometer is changed, you can set things up so that a single version of the *config* and *settings* files will describe a number of different spectrometers. Here is how to set up the files:

(1) If you already have several geometry configurations installed, you should make backup copies of the *config* and *settings* files from the current geometries.

(2) If you already have several geometry configurations installed, remove the *config* and *settings* files from all but one of the geometry directories. Save the *config* file that has the most motors, as you will have to add motors from the other geometries to the remaining *config* file.

(3) Set up hard links in all the geometry directories so that the *config* and *settings* in all the geometry directories refer to the same file. For example, if the files already exist in the *fourc* directory, use the commands

```
ln fourc/config surf/config
ln fourc/settings surf/settings
```

to create hard links in the *surf* directory. Don't use symbolic links.

(4) Edit the *config* file by hand to add new control lines that assign numbers to the different geometries. These control lines must be before the lines that assign motor information. The format of the geometry control lines is as follows:

```
GEO0 = common
GEO1 = fourc
GEO2 = surf
GEO3 = fivec
etc.
```

The parameter `GEO0` always refers to the motors that are common to all the geometries. Subsequent lines assign consecutive numbers to the other geometries.

(5) Now run *edconf*. The motor screen will have a new field that lets you assign a spectrometer geometry to each motor or to make the motor in common with all the spectrometers. You can do the same for each scaler on the scaler screen (as of release 4.03.12).

The hard links must be maintained for the shared *config* and *settings* file scheme to work. You can safely use *vi* and *cp* to manipulate the files. However, using *mv* will destroy the links, as will some text editors.

When running *edconf* with a geometry directory as an argument or when invoking the *config* macro from spec, use the `G` command to toggle between displaying all the motors and scalers in the *config* file and just those motors and scalers used by the given geometry.

Security Issues

At some installations, you may wish to prevent ordinary users from accessing selected motors. spec offers several levels of security. The security works with the UNIX file ownership and protection mechanisms, so it is important that user and group ownership of the configuration files and file write permission be properly set.

To restrict configuration modification to a single user or group, you must set the write permission of the diffractometer's associated *config* file accordingly. Type `chmod 644 /usr/lib/spec.d/fourc/config` to allow only the owner of the file to modify it. Setting the mode to 664 allows users in the owner's group to also modify the file.

There are there basic levels of security for each motor. The first level is the most restrictive, as it prevents the motor from being moved and prevents changes to the position being made in the *settings* file. The motor position can still be read from the motor controller, though, and the user angle can still be changed using the `chg_offset()` function (invoked by the `set` macro). If there is ever a conflict between the current position and the position in the *settings* file, such as might happen if the power was turned off to the motor controller, the controller registers are

automatically adjusted to match the position in the *settings* file without moving the motor.

The second level of security allows a motor to be moved, but prevents the software limits from being changed. Not only is the `set_lim()` command restricted, but also the `chg_dial()` command, as a change in the dial position would effectively change the position of the limits.

The third level offers no security and allows any operation on a motor.

When these motor restrictions are set in the *config* file, the restrictions apply to everyone, even the owner of the *config* file. To move a restricted motor, you must first change the *config* file.

Extra Protection

At some spec installations, the administrators need to prevent users from accessing or modifying the configuration of certain motors. The *edconf* program supports a wizard mode that allows such protection. If you type `^W` while running *edconf* you will be prompted for the wizard's password. If you enter it properly, you will be able to select additional levels of configuration protection.

When running spec, a user who knows the password can gain temporary access to protected motors via the `spec_par("specwiz")` function. The standard macros `onwiz` and `offwiz` provide a convenient implementation.

Since modern UNIX systems forbid unprivileged users access to encrypted passwords, spec will first look for a readable `SPECD`/*passwd* file that contains an entry containing an encrypted password for a *specwiz* user. If no such file exists, spec will also look in /*etc*/*passwd* and /*etc*/*shadow*, although the former is unlikely to contain encrypted passwords and the latter is unlikely to be readable. The spec distribution includes a *wiz_passwd* utility, which can be run to create the `SPECD`/*passwd* file.

To prevent users from disabling the wizard protections by editing the *config* file by hand, you can use file protection features built in to UNIX. One possibility is to make the *edconf* program set-user id *specadm*, change the ownership of the *config* files to *specadm*, and change the modes of the *config* files to `rw-r--r--`. Do that using commands (as super user) along the following lines:

```
chown specadm edconf fourc/config surf/config ...
chmod u+s edconf
chmod 644 fourc/config surf/config ...
```

HARDWARE REFERENCE

Introduction

spec includes built-in support for a wide variety of motor controllers, counters, timers and other data acquisition devices, allowing great flexibility in a site's hardware configuration. Information on currently supported devices follow. Support for additional devices is continually being added.

Interface Controllers and General Input/Output

The interface screen of the configuration editor is selected using the `I` command. Before any interfaces have been configured the screen looks like:

```
Interface Configuration

  CAMAC        DEVICE   ADDR  <>MODE                                    <>TYPE
     NO

   GPIB        DEVICE   ADDR  <>MODE                                    <>TYPE
0    NO

    VME        DEVICE   ADDR                                            <>TYPE
     NO

 SERIAL        DEVICE <>TYPE  <>BAUD                                    <>MODE
0    NO
1    NO
2    NO
3    NO

IO PORT                 ADDR  <>MODE  NUM
     NO
     NO
     NO
     NO

                                        Type ? or H for help, ^C to quit
```

The following sections explain the choices for each type of interface. To select a particular interface, use the arrow keys to move to the cell containing the word `NO` and type `y` for yes and then `<return>`. For CAMAC, GPIB and VME devices, move the cursor to the last column and use the < or > keys to select the correct device and then enter `<return>`.

CAMAC Controllers

spec supports only one CAMAC controller at a time. The following CAMAC crate controllers are available:

config file	Description
PC_DSP6001	DSP 6001 with PC004 (no driver)
GP_CC488	DSP CC-488 GPIB Crate Controller
CDEV	DSP-6001/DCC-11/KS-3912 Boards
CA_JOR73A	Jorway 73A SCSI to CAMAC
CA_KS3929_HP	KS 3929 SCSI to CAMAC on HP
CA_KS3929	KS 3929 SCSI to CAMAC on Sun
GP_KS3988	KS 3988 GPIB Crate Controller
PC_KSC2926	KSC 2926 with 3922 (no driver)
CA_KSC	Kinetic Systems CAMAC driver

To select one, move the cursor to the NO box under CAMAC and type y and <return>. Then move the cursor to the rightmost column and type < or > until the appropriate controller appears and then enter <return>. Finally, select appropriate parameters from the other columns.

CAMAC Controllers That Use spec *Drivers*

config file:

```
CDEV = device_name INTR|POLL
```

edconf interfaces screen:

```
  CAMAC        DEVICE   ADDR  <>MODE                                    <>TYPE
    YES     /dev/ca00           INTR            DSP-6001/DCC-11/KS-3912 Boards
```

spec drivers are available for the DSP 6001/6002 with PC004 for PC platforms, the Kinetic Systems 3922 with 2926 for PC platforms, or the Kinetic Systems 3912 CAMAC controller for BSD and Ultrix platforms. The appropriate CSS driver must be installed in each case. The PC platform controllers may be used in a polled or interrupt-driven mode. In interrupt-driven mode, a CAMAC look-at-me (LAM) will generate a call to a spec interrupt service routine.

To use the DSP 6001/6002 controllers in interrupt-driven mode, the boards must be modified to give the module a software programmable interrupt-enable capability. The modifications involve cutting four traces and soldering four jumper wires on one of the boards in the 6001/2 module. Contact CSS to obtain the explicit instructions. No modifications are required to operate the controller in polled mode.

DSP 6001/6002 CAMAC With No Driver

config file:

```
PC_DSP6001 = base_address
```

edconf interfaces screen:

```
  CAMAC        DEVICE   ADDR  <>MODE                                    <>TYPE
    YES                0x240                    DSP 6001 with PC004 (no driver)
```

Use this entry to select the DSP 6001/6002 CAMAC controller on a PC, if you aren't using a driver. No interrupt is used in this configuration.

Kinetic Systems 3988 GPIB To CAMAC

config file:

```
GP_KS3988 = gpib_address INTR|POLL
```

edconf interfaces screen:

```
  CAMAC        DEVICE   ADDR  <>MODE                                    <>TYPE
    YES                   7    POLL                KS 3988 GPIB Crate Controller
```

Use this configuration for the Kinetic Systems 3988 GPIB-to-CAMAC controller. This controller may be used in a polled or interrupt-driven mode. In interrupt-driven mode, a CAMAC look-at-me (LAM) generates a GPIB service request (SRQ), which in turn, will call a spec interrupt service routine. Interrupt-driven mode is currently only available with National Instruments GPIB controllers, and only when not using the *cib.o* GPIB configuration (see below). If multiple versions of spec are sharing the controller on the same computer, the controller must be operated in polled mode.

DSP CC-488 GPIB To CAMAC

config file:

```
GP_CC488 = gpib_address INTR|POLL
```

edconf interfaces screen:

```
  CAMAC        DEVICE   ADDR  <>MODE                                    <>TYPE
    YES                   7    POLL              DSP CC-488 GPIB Crate Controller
```

This configuration is for the DSP CC-488 GPIB-to-CAMAC controller. This controller may be used in a polled or interrupt-driven mode. In interrupt-driven mode, a CAMAC look-at-me (LAM) generates a GPIB service request (SRQ), which in turn, will call a spec interrupt service routine. Interrupt-driven mode is currently only available with National Instruments GPIB controllers. If multiple versions of spec

are sharing the controller on the same computer, the controller must be operated in polled mode.

Jorway 73A SCSI To CAMAC

config file:

```
CA_JOR73A = device_name
```

edconf interfaces screen:

```
  CAMAC        DEVICE   ADDR  <>MODE                                  <>TYPE
    YES      /dev/sga                                Jorway 73A SCSI to CAMAC
    YES   /dev/scsi/1                                Jorway 73A SCSI to CAMAC
```

The Jorway 73A SCSI-to-CAMAC controller is supported on HP 700 Series and *linux* platforms. Note, the four-position "piano" switch should be left in the factory configuration, with positions 1, 2 and 3 off, and position 4 on.

Kinetic Systems 3929 SCSI To CAMAC

config file:

```
CA_KS3929 = device_name
```

edconf interfaces screen:

```
  CAMAC        DEVICE   ADDR  <>MODE                                  <>TYPE
    YES      /dev/sga                             KS-3929 SCSI to CAMAC
    YES     /dev/ksc0                             KS-3929 SCSI to CAMAC
    YES   /dev/scsi/1                             KS-3929 SCSI to CAMAC
```

This Kinetic Systems SCSI-to-CAMAC controller is available on SunOS 4.x SBus platforms, where a CSS provided driver can be installed. On *linux* and HP 700 Series workstations, spec provides direct software support. In all cases, this controller only operates with spec in a polled mode. In addition the software interface available from Kinetic Systems is supported (see below).

config file:

```
CA_KSC = device_name
```

edconf interfaces screen:

```
  CAMAC        DEVICE   ADDR  <>MODE                                    <>TYPE
    YES   /dev/rcamac                              Kinetic Systems CAMAC Driver
```

Kinetic Systems sells software interfaces for some of their CAMAC controllers on some UNIX platforms. Presently, only the package for the 3929 SCSI-to-CAMAC controller on the HP700 platform has been used with spec. To use the Kinetic Systems software, the location of the their object module must be given in response to the "KSC 3929 SCSI-CAMAC file location" query when running the *Install* program. (Note, however, there is no reason to use the rather expensive KSC software for the 3929 on the HP platform as CSS provides bundled support.)

GPIB Controllers

spec works with a variety of GPIB controllers as described below. (In addition, the Kinetic Systems 3388 CAMAC-to-GPIB controller is available.) Up to four GPIB controllers can be configured simultaneously.

On platforms that implement the System V interprocess communications (IPC) semaphore and shared-memory system calls, more than one spec process can share a single GPIB controller. For those systems, each spec must have the *shared* version of the GPIB controller selected. In the *config* file, a _L is appended to the module key word to indicate the shared version. If multiple GPIB controllers are configured, the controller unit number of the shared controller must be the same in each version of spec.

spec uses the GPIB controllers at *board* (as opposed to *device*) level, which makes it unlikely that other programs can use a GPIB controller while spec is using it.

National Instruments GPIB with National Instruments Drivers

The National Instruments GPIB boards and drivers should be installed according to National Instruments instructions. spec communicates with the boards using only the device node */dev/gpib0*.

(On System V PC Platforms, where the GPIB driver is linked into the kernel, and the kernel is patched using the *ibconf* program, you should run *ibconf* directly on the driver file, so that each time you rebuild a kernel, you won't need to rerun *ibconf*. Thus, you might run

```
ibconf /etc/conf/pack.d/ib2/Driver.o
```

after the driver has been installed.)

When you do configure */dev/gpib0* with the *ibconf* program, set the controller primary address to 0, the secondary address to none, board-is-system-controller mode to yes and disable-auto-serial-polling mode to yes.

Other board configuration parameters are programmed by spec each time it is run, thus overriding any values you may set using the *ibconf* program. Those parameters are: timeout setting, EOS byte, terminate-read-on-EOS mode, type of compare on EOS, set-EOI-w/last-byte-of-write mode and UNIX signal. The special nodes that may be created for each individual device by the National Instruments installation program are not used at all.

National Instruments GPIB with cib.o

config file:

```
PC_GPIBPC4 = device_name
PC_GPIBPC4_L = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES    /dev/gpib0                                    Nat Inst with cib.o
    YES    /dev/gpib0                          Nat Inst with cib.o (shared)
```

National Instruments provides a C language interface to its driver on most platforms in a file called *cib.c*. For some older versions of the National Instruments driver, spec has built-in C code that can be used instead of the National Instruments C interface. When using the National Instruments *cib.o* file, it isn't possible for GPIB devices that generate service requests (SRQ) to generate interrupts. Such devices must be used in polled mode.

In addition to specifying this choice in the *config* file, the location of the *cib.o* file must be entered when spec is installed. (See Page ? in the *Administrator's Guide*.

Any of the National Instruments boards (except the DEC MicroVax board) can be used with the *cib.o* file.

National Instruments GPIB on linux

config file:

```
PC_GPIBPC = device_name
PC_GPIBPC_L = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES /dev/gpib0/master                          National Instruments GPIB
    YES /dev/gpib0/master                             Nat Inst GPIB (shared)
```

On *linux* platforms, there is a freely available GPIB driver for National Instruments boards available by anonymous ftp from the site `koala.chemie.fu-berlin.de` in the directory */pub/linux/LINUX-LAB/IEEE488*. That driver should be installed and configured according to the documentation in the driver package. Note, however, that the DMA option in the driver installation should *not* be selected, as the implementation of DMA in the driver is notoriously flakey. The only part of the driver package needed by spec is the *driver/gpib0.o* module. spec does not use any of the application library included with the driver package. The file */etc/gpib.conf* associated with the application library, so also is not used by and does not influence spec. Also, spec communicates with the driver directly through the */dev/gpib0/master* device node. No other device nodes are used.

National Instruments GPIB-ENET

config file:

```
PC_GPIBPC5 = hostname
PC_GPIBPC5_L = hostname
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES         gpib0                                    Nat Inst GPIB-ENET
    YES         gpib0                                Nat Inst GPIB-ENET (shared)
```

spec must be linked with the National Instruments *cib.o* module that comes with the GPIB ethernet device. On the *edconf* interfaces screen, enter the name of the board as indicated by the National Instruments *ibconf* utility.

National Instruments PCII GPIB on PC UNIX System V Platforms

config file:

```
PC_GPIBPC = device_name
PC_GPIBPC_L = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES    /dev/gpib0                                      Nat Inst GPIB PCII
    YES    /dev/gpib0                                Nat Inst GPIB PCII (shared)
```

The PCII board is an old model, but is still supported by spec. This selection uses spec's built-in code. The *cib.o* file configuration described earlier may also be used.

National Instruments AT-GPIB on PC UNIX System V Platforms

config file:

```
PC_GPIBPC2 = device_name
PC_GPIBPC2_L = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES    /dev/gpib0                                        Nat Inst AT-GPIB
    YES    /dev/gpib0                                  Nat Inst AT-GPIB (shared)
```

The AT-GPIB board is a current model. This selection uses spec's built-in code. The *cib.o* file configuration described earlier may also be used.

National Instruments GPIB on SCO UNIX and IBM AIX Platforms

config file:

```
PC_GPIBPC = device_name
PC_GPIBPC_L = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES    /dev/gpib0                                  National Instruments GPIB
    YES    /dev/gpib0                                     Nat Inst GPIB (shared)
```

spec's built-in code should work with SCO UNIX, IBM PS/2 and IBM RS/6000 platforms with this configuration choice. If new versions of the National Instrument drivers don't work, switch to the *cib.o* file configuration, described previously.

National Instruments SB-GPIB Ver 1.3 on SunOS 4.x Platforms

config file:

```
PC_GPIBPC = device_name
PC_GPIBPC_L = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES    /dev/gpib0                                Nat Inst SB-GPIB Ver 1.3
    YES    /dev/gpib0                      Nat Inst SB-GPIB Ver 1.3 (shared)
```

This old version of the driver is supported with built-in code.

National Instruments GPIB 1024-1S on SunOS 4.x Platforms

config file:

```
PC_GPIBPC2 = device_name
PC_GPIBPC2_L = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES    /dev/gpib0                                Nat Inst GPIB 1014-1S
    YES    /dev/gpib0                      Nat Inst GPIB 1024-1S (shared)
```

An old version of the driver for this board is supported with built-in code.

National Instruments SB-GPIB Ver 2.1 on SunOS 4.x Platforms

config file:

```
PC_GPIBPC3 = device_name
PC_GPIBPC3_L = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES    /dev/gpib0                                Nat Inst SB-GPIB Ver 2.1
    YES    /dev/gpib0                      Nat Inst SB-GPIB Ver 2.1 (shared)
```

This old version of the driver for this board is supported with built-in code.

National Instruments GPIB on DEC MicroVax

config file:

```
PC_GPIB11 = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES       /dev/ib                                 DEC GPIB11V-2 For Q-Bus
```

National Instruments distributes the driver source code for this module and platform. The driver is unlikely to updated, so spec should continue to work with this module and platform indefinitely. No *cib.o* file is available for this platform.

HP SICL GPIB On HP Platforms

config file:

```
PC_SICL_H = sicl_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES          hpib                                        HP SICL GPIB
```

This configuration choice supports HP's GPIB using HP's SICL interface library. When the spec *Install* script is run, the question regarding GPIB SICL must be answered with "yes", and a *libsicl.a* or *libsicl.sl* must be available on the system for the SICL GPIB board to be available. If using the HP E2050 LAN/HP-IB Gateway, the device name is of the form *lan*[*hostname*]:*interface* where *lan* is the symbolic name for the device set in the */usr/pil/etc/hwconfig.hw* file, *hostname* is the host name or IP address of the gateway and *interface* is the interface as set for the host-name parameter in the gateway on-board configuration.

IOtech SCSI To GPIB On HP Platforms

config file:

```
PC_SICL_H = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES  /dev/IOtech1                                IOtech SCSI488/H SICL GPIB
```

This configuration uses IOtech's SCSI-to-GPIB interface on an HP 700 Series workstation with the IOtech SICL software. When the spec *Install* script is run, the flags required to load the GPIB SICL library must be entered. Note, CSS has available a

modified version of the standard *libsicl.a* that doesn't use the *ieee488* daemon program, but does allow multiple processes to access the GPIB controller. Contact CSS to obtain the modified version of the library.

IOtech SCSI To GPIB on Sun Platforms

config file:

```
PC_IOTECH = device_name
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES  <eee488/ieee                                  IOtech SCSI488/S GPIB
```

Use this configuration for the IOtech SCSI-to-GPIB on a SunOS 4.x platform.

The full name of the default device is */dev/ieee488/ieee*. (The *edconf* program only displays the last twelve characters of long device names unless the cursor is on the device name cell.)

Scientific Solutions IEEE-488 on PC Platforms

config file:

```
PC_TEC488 = base_address
PC_TEC488_L = base_address
```

edconf interfaces screen:

```
   GPIB        DEVICE   ADDR                                          <>TYPE
    YES                 0x300                    Scientific Solutions IEEE-488
    YES                 0x300                    Scien Solut IEEE-488 (shared)
```

Only the Scientific Solutions (Tecmar) GPIB board (old style) is currently supported by spec (not the GPIB-LM model). Very old models of the board do not work. The card is accessed completely through user-level I/O. No kernel driver is needed.

Kinetic Systems 3388 CAMAC-To-GPIB Module

config file:

```
CA_KS3388 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    KS3388                KS 3388 GPIB Interface
```

When using the Kinetic Systems 3388 GPIB-to-CAMAC module, you must set the talk/listen address switch inside the module to correspond to address zero.

VME Controllers

National Instruments VME with National Instruments Drivers

config file: *edconf*

```
PC_NIVME = /dev/null
```

interfaces screen:

```
    VME        DEVICE   ADDR                                              <>TYPE
    YES                                                 National Instruments VME
```

spec supports the National Instruments VXI-SB2020 on SunOS 4.1 and the VXI-AT2023 on System V PC platforms. The NI drivers must be installed, and the location of the NI *cvxi.o* must be specified when spec is installed.

The National Instruments software needs several patches when used on SVR3 PC platforms. Contact CSS for details.

Bit 3 Model 403 ISA-VME

Bit 3 Model 616/617 PCI-VME

Bit 3 Model 487-1 with Model 933 Driver Software

Bit 3 Model 466-1/467-1 with Model 944 Driver Software

Serial (RS-232C) Ports

config file:

```
SDEV_# = device_name baud_rate tty_modes
```

edconf interfaces screen:

```
 SERIAL        DEVICE <>TYPE  <>BAUD                                    <>MODE
0   YES    /dev/ttya1     <>    9600                              cooked igncr
1   YES    /dev/ttya2     <>    2400                                    raw
2    NO
3    NO
```

Serial ports for use with the user-level `ser_get()` and `ser_put()` built-in functions are also selected on the interfaces screen. The device name, baud rate and serial line modes are selected for up to four serial devices. The number of the device is used as the first argument to the `ser_put()` and `ser_get()` functions. Available tty modes are either *raw* or *cooked*, with *cooked* mode also having *noflow*, *igncr* (a no-op on non-System V systems) and *evenp* or *oddp* options.

The `<>TYPE` field allows serial devices connected through special software servers used at ESRF or with EPICS to be selected. For normal serial devices, the field should contain the characters `<>`. See the *Reference Manual* for a description of the spec functions that access the serial ports.

Generalized CAMAC I/O

config file:

```
CA_IO = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1        IO     0          Generalized CAMAC I/O
```

CAMAC modules configured for Generalized CAMAC I/O can be accessed using spec's `ca_get()` and `ca_put()` functions. (See page 150 in the *Reference Manual*.) The module address for those functions is the unit number on the CAMAC configuration screen. Note that even more generalized access is available using the `ca_fna()` function. (See page 150 in the *Reference Manual*.) Arbitrary commands can be sent to any CAMAC module, whether or not the module is listed in the *config* file.

PC Port Input/Output

config file:

```
PC_PORT_# = base_address number_of_ports read_write_flag
```

edconf interfaces screen:

```
IO PORT                 ADDR  <>MODE  NUM
    YES                0x300    Read    1
    YES                0x310     R/W    4
     NO
     NO
```

On ISA bus systems on 80x86-compatible systems and on HP 700 platforms with E/ISA bus support, the ports available for the built-in `port_get()`, `port_getw()`, `port_put()` and `port_putw()` functions are configured on the interfaces screen. The board's hexadecimal base address is given, along with the number of contiguous 8-bit ports (maximum of 16) that can be accessed. The ports can be configured for read-only access or for read-write access. Be careful not to select port addresses associated with standard PC hardware such as the video board or the hard disk! Also be sure to include enough 8-bit ports to handle 16-bit word access, if that is how you will be using the ports. On the HP platforms, a config file must also be set up in the */etc/eisa* directory using the HP *eisa_config* utility.

Motor Controllers

Motor Controllers

Advanced Control System MCB (GPIB and Serial)

config file:

```
RS_MCB = device_name baud_rate number_of_motors
GP_MCB = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                <>TYPE
    YES                    6            4    Advanced Control System MCB (GPIB)
    YES    /dev/tty00     <>    9600    4  Advanced Control System MCB (Serial)
```

When used on the serial interface, the MCB appears to output characters using even parity in spite of the board's jumpers being set to generate no parity. To accommodate that hardware idiosyncrasy, spec opens the port using even parity. If future versions of the MCB generate parity properly according to the jumper settings, the jumpers should be set for even parity to accommodate spec.

Advanced Control System MCU-2 (Serial)

config file:

```
RS_MCU = device_name baud_rate number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                <>TYPE
    YES    /dev/ttyS1     <>    9600   12           Advanced Control System MCU
```

On the *Motors* screen of the configuration editor, one can select among `MCU`, `MCU_E`, `MCU_H` and `MCU_O` for the controller type.

The `MCU_O` controller type indicates that the controllers have the old-style firmware (which doesn't implement the # start character for message sent to the controllers), so that spec will not test first for the new-style firmware, thus avoiding timeouts.

The `MCU_H` controller type is for a special version of the controller which includes a hardware tie-in for a Heidenhain encoder. For such controllers, the `motor_par("encoder_step_size")` parameter is relevant in order to set the ratio between the Heidenhain encoder readings and the MCU-2 step size. The default value is 131072 /

360.

The MCU_E type indicates an encoder is used. However, currently, there is no difference in the software whether MCU_E or MCU is selected.

Please note, many users have had problems establishing initial communication between the MCU-2 controllers and their computers. The problem is almost always related to the cable. Standard RS-232C cables do not appear to work. A custom cable wired according to the diagram in the MCU-2 manual seems to be needed. Note also, that the connections for pins 2 and 3 may need to be swapped from what is shown in the manual. Be prepared to try the cable both ways before adding the final touches.

Command pass through for the MCU-2 controllers is available using the following:

`motor_par(`*`motor`*`, "send", `*`cmd`*`)` – Sends the string *cmd* to the MCU channel associated with *motor*. For example, set *cmd* to `"J500"` to set the jog rate for *motor* to 500 steps per second.

`motor_par(`*`motor`*`, "read", `*`cmd`*`)` – Sends the string *cmd* to the MCU channel associated with *motor*, as above, and returns a string containing the response.

Compumotor 3000 (GPIB and Serial)

config file:

```
RS_CM3000 = device_name baud_rate number_of_motors
GP_CM3000 = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                              <>TYPE
    YES    /dev/tty00     <>    9600    4              Compumotor 3000 (Serial)
    YES                    6            4                Compumotor 3000 (GPIB)
```

Use of this old motor controller is not recommended!

Compumotor 4000 (GPIB and Serial)

config file:

```
RS1_CM4000 = device_name baud_rate number_of_motors
RS2_CM4000 = device_name baud_rate number_of_motors
GP_CM4000 = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES    /dev/tty00     <>    9600    4       Compumotor 4000 (Serial port 1)
    YES    /dev/tty00     <>    9600    4       Compumotor 4000 (Serial port 2)
    YES                    3            4                Compumotor 4000 (GPIB)
```

The Compumotor 4000 can be used on either an RS-232 or GPIB interface. You must program the RS-232 baud rate or the GPIB address using the Compumotor front panel controls. You should consult the Compumotor manual for details, but in brief, the procedure is as follows. You must first enter the *ACCESS* code (the factory default is *4000*). You then choose the *IMMED* function, and then the *DEFINE GPIB ADDR* statement to select the the GPIB address. Alternatively, choose the *IMMED* function, and the the *RS232 PORT1* or *RS232 PORT2* statement to select the port and configure the baud rate for the RS-232 interface.

There are many other configuration options with this controller. Other than the GPIB address and the baud rate, you should probably not change any of these others. You can reestablish the factory defaults using the *RESET* function from the main menu.

Output pins 46 or 47 on the programmable output connector can be used to gate a counter during powder-mode scans. While the powder-mode motor is moving during these scans, spec sets pin 46 high and pin 47 low. Use the one appropriate for your particular counter. To gate the Ortec 994 counter/timer, for example, pin 46 and an even-numbered pin (all are logic ground) are connected to the front panel *enable* BNC connector on the Ortec module.

Compumotor AX (Serial)

config file:

```
RS_CMAX = device_name baud_rate number_of_motors
```

edconf devices screen:

```
 MOTORS      DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES  /dev/tty00     <>    9600    4 Compumotor AX Motor Controller (Serial)
```

Command Pass Through

Command pass through is available using the following functions. Command pass through should be used with caution to avoid interfering with the built-in programming commands spec sends to the controllers.

`motor_par(`*`motor`*`, "send",` *`cmd`*`)` — Sends the string *`cmd`* to the motor channel associated with *`motor`*.

`motor_par(`*`motor`*`, "read", `*`cmd`*`)` — Sends the string *cmd* to the motor channel associated with *motor*, as above, and returns a string containing the response.

Compumotor SX (Serial)

DSP E250 12-Bit DAC as Motor Controller (CAMAC)

config file:

```
CA_E250 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1      E250     0          DSP E250 12-Bit D/A as Motor Control
```

Some spec users use this DAC to control piezo-electric motion devices. Commanding such a device to move from spec results in an instantaneous change in the output voltage of the DAC.

DSP E500 Stepper Motor Controller (CAMAC)

config file:

```
CA_E500 = slot_number
CA_E500M = slot_number
CA_IOM1 = slot_number
CA_IOM2 = slot_number
CA_IOM3 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1      E500     0          DSP E500 Stepper Motor Controller
   2      IOM1               BiRa 2601 I/O For E500 Multiplexing
   3      IOM2               F16,A0 I/O For E500 Multiplexing
   4      IOM3               F16,A1 I/O For E500 Multiplexing
```

Selecting one of the three I/O module configurations above allows spec to multiplex an E500 motor channel. Currently only one channel of one E500 can be multiplexed. However, up to sixteen motors can be multiplexed on that channel. Contact CSS for additional information on the multiplexing circuitry required.

Huber SMC 9000 (GPIB)

config file:

```
GP_HUB9000 = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES                    6            8    Huber 9000 Motor Controller (GPIB)
```

Note, spec does not support the RS-232C interface for this controller, as CSS was unable to make it work reliably with spec. Note also, a feature of this motor controller is that if a limit switch is hit, communication with the remote computer is shutdown so that one must manually move the motor off the limit and reset the controller in order to reestablish remote communication.

Inel XRGCI as Motor Controller (Serial)

config file:

```
RS_XRGCI_M = device_name baud_rate number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES    /dev/ttyh3     <>    4800    3        Inel XRGCI as Motor Controller
```

Joerger SMC Stepper Motor Controllers (CAMAC)

config file:

```
CA_SMC = slot_number
CA_KS3640M = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1       SMC     0          Joerger Single Motor Controller
   2   KS3640M     0          KS 3640 Counter with Joerger SMC
```

Joerger controller models SMC-L, SMC-24 and SMC-LP can be used with spec. None of these models contain absolute motor position registers, so they are generally supplemented with additional CAMAC counter modules to keep track of motor position. spec supports the Kinetic Systems 3640 Up/Down Counter for this purpose. This module has four 16-bit counters that must be connected in series to form two 32-bit counters. One 3640 module is thus needed for each two SMC modules.

The SMC controllers can be used without a supplemental counter, but spec will not do as well in keeping track of absolute motor positions.

The Joerger controllers have no provision for a soft abort. If you type **^C** to abort moving, the controller will simply stop sending pulses to the motors. Inertia may cause the motors to continue to turn a bit, and absolute positions will be lost. The only work around for this problem is to keep motor velocities low.

The model SMC-LP allows programmable motor speed and acceleration. According to the module's documentation, if the internal frequency adjustments are at the factory settings, motor speed can be programmed between 50 and 2000 steps per second, while acceleration time can be varied from 20 to 2000 msec. For the other models, these parameters are set manually using potentiometers on the module — values entered in the *config* file are ignored.

Klinger MC-4 Stepping Motor Controller

config file:

```
RS_MC4 = device_name baud_rate number_of_motors
GP_MC4 = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                          <>TYPE
    YES    /dev/tty00     <>    4800    4                Klinger MC4 (Serial)
    YES                    3            4                  Klinger MC4 (GPIB)
```

The Klinger MC-4 can be used on either an RS-232 or GPIB interface.

Back panel switches SW, SX, SY and SZ should each have locations 1 and 2 up and locations 3 and 4 down (spec does not currently support an origin switch, top zero or encoders). If you use the MC-4 with a GPIB interface, back panel switch S3 location 4 should be down, indicating a line feed terminator. Use switch S1 locations 1 to 5 select the GPIB address.

If you use the MC-4 with an RS-232 interface, set back panel switch S2 locations 1 to 3 to select the baud rate. Location 4 should be up, indicating software handshake. Switch S3 location 1 should be down and locations 2 and 3 up, to set 8 bit data words with 2 stop bits. Switch S3 location 4 should be down, indicating a line feed terminator.

You must connect certain pins together on the *general purpose I/O* connector on the back panel of the MC-4 to use the MC-4 with spec. Wire one end of a 1K to 10K ohm resistor to pin 35 (+5V). The other end should be connected to both pins 26 (B8) and 29 (E3). Pin 26 is an programmable output pin, and pin 29 is an external input. spec sets the state of pin 26 at the beginning of each move (which may involve all the

motors on the board) and clears the state of the pin at the end of the move. The status of pin 29 is read to determine when all activity, including backlash, is completed. The connection to pin 35 (+5V) is necessary to pull up the output.

Output pins 24 or 25 on the *general purpose I/O* connector can be used to gate a counter during powder-mode scans. While the powder-mode motor is moving during these scans, spec sets pin 24 high and pin 25 low. Use the one appropriate for your particular counter. To gate the Ortec 994 counter/timer, for example, pin 24 and pin 36 (logic ground) are connected to the front panel *enable* BNC connector on the Ortec module.

Kinetic Systems 3112 12-Bit DAC as Motor Controller (CAMAC)

config file:

```
CA_KS3112 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    KS3112     0          KS 3112 12-Bit D/A as Motor Control
```

Some spec users use this DAC to control piezo-electric motion devices. Commanding such a device to move from spec results in an instantaneous change in the output voltage of the DAC.

Micro-Controle IP28 (GPIB and Serial)

config file:

```
RS_IP28 = device_name baud_rate number_of_motors
GP_IP28 = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                <>TYPE
    YES    /dev/tty00     <>    9600    4          Micro-Controle IP28 (Serial)
    YES                    6            4            Micro-Controle IP28 (GPIB)
```

Code for this motor controller was developed by a spec user. If new users plan on using this controller, please contact CSS first.

MicroControle SIX19 (Serial)

config file:

```
RS_SIX19 = device_name baud_rate number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                              <>TYPE
    YES                                13                   MicroControle SIX19
```

Missouri University Research Reactor Motor Controller (GPIB)

config file:

```
HW_MURR = number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                <>TYPE
    YES                                13      Missouri Research Reactor Motors
```

edconf motor screen:

```
Number: <>Controller        0:  MURR_E   1:  MURR_E   2:    MURR   3:    MURR
Unit/Channel                       0/1          0/2          0/3          0/4
Name                         Two Theta        Theta          Chi          Phi
Mnemonic                           tth           th          chi          phi
```

The University of Missouri Research Reactor uses custom motor controllers. Each channel of the motor controller requires one GPIB address. The GPIB address is set using the channel number in the required unit/channel configuration on the motor screen of the configuration editor. The unit number is not relevant for these motors.

The controller type `MURR_E` or `MURR` is selected depending on whether or not the controller channel uses an encoder.

There are several unique parameters associated with each motor channel. The parameters assume default values when the controllers are powered up. Alternate values can be set in the *config* file that will be programmed by spec. The values are

Name	Parameter Name	Power-On Value
Modulo	Generic Parameter 1	360000
Grain	Generic Parameter 2	5
Direction	Generic Parameter 3	1
Drive Mode	Generic Parameter 4	1
Cut Point	Generic Parameter 5	0

For the *modulo* parameter, the value 360000 is appropriate for rotation stages. For

translation stages, the maximum value (999999?) would be appropriate. The *grain* parameter is a multiplier for the steps sent to the motor by spec and should be simply related to a gear reducer value. The *direction* parameter should be set to zero to reverse the direction of the motor so that its position agrees with the controller display. The *drive-mode* parameter changes the meaning of the output signals. For a value of one, the output signals are count-up/count-down. For a value of zero, the output signals are step/direction.

The *cut-point* parameter allows negative positions to be reported by spec, even though the controller only reports positive positions. Positions reported by spec will be between the cut point and the cut point plus the modulo parameter multiplied by the step size parameter. For a module of 360000 and a cut point of −180, positions will be between −180 and +180, for example.

New Focus Model 8732 Picomotor Controller (GPIB and Serial)

config file:

```
RS_NF8732 = device_name baud_rate number_of_motors
GP_NF8732 = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES    /dev/tty00           9600    4     New Focus Picomotor 8732 (Serial)
    YES                    6            4       New Focus Picomotor 8732 (GPIB)
```

Each New Focus Picomotor controller unit has five card slots available. Each card has four connectors, and each connector can control up to three channels. The slot, connector and channel numbers need to be encoded in the configuration file by entering unit/channel information with the configuration editor. The unit number selects which Picomotor controller. The channel number selects the slot/connector/channel number of the controller encoded as XYZ, where the slot number is 1 <= X <= 5, the connector is 1 <= Y <= 4 and the channel is 1 <= Z <= 3.

The only parameter from the *config* file used to program these controllers is the steady-state rate, which is sent to the controller as the pulse frequency.

The New Focus Picomotors are unlike most motor controllers in that there is no way to read the motor positions, and no way to know how far the motors move when commanded. spec attempts to guess how far the motor has moved if the positions are read while the motor is active, or if the move was aborted or stopped, based on the elapsed time of the move and the pulse frequency programmed into the controller. However, the positions reported by spec for the New Focus controllers should not be taken too seriously.

The motor controller can only move 65,535 steps at a time. For larger moves, spec will automatically send 65,535 steps at a time until the complete move is performed.

Only the up-to-three motors on the same connector can be moved simultaneously. spec prints an error message if you to try to move a motor on a one connector while motors on another connector are active. (The moves could be automatically queued in software if users think such additional code development is warranted.)

Note also, the commands `chg_dial(`*`mne`*`, "lim+")` and `chg_dial(`*`mne`*`, "lim-")` can be used to start continuous moves, which can be stopped either with `^C` or the `stop()` function. Please beware also, there is no limit switch capability with this controller, so moves must be stopped by user intervention.

Command pass through is available using the following functions.

`motor_par(`*`motor`*`, "send", `*`cmd`*`)` — Sends the string *cmd* to the New Focus channel associated with *motor*.

`motor_par(`*`motor`*`, "read", `*`cmd`*`)` — Sends the string *cmd* to the New Focus channel associated with *motor*, as above, and returns a string containing the response.

The following special commands are also available:

`motor_par(`*`motor`*`, "always_address", `*`mode`*`)` — By default, spec currently prefixes each command sent to the controller with an eleven-character channel-select instruction. Communication can be made more efficient by turning off the `"always_address"` feature by calling this function with *mode* set to 0. With the mode off, the channel-select instruction is only sent when needed. The reason for always sending the channel-select information is to avoid spec losing track of the current channel in the event a user manually switches the controller to local mode.

`motor_par(`*`motor`*`, "was_local")` — This command will force spec to issue a channel-select instruction for the next command, allowing recovery after manually switching the controller to local mode when the `"always_address"` mode is off.

Newport (Klinger) Motion Master 2000/3000 (GPIB, Serial and PC Board)

config file:

```
RS_MM2000 = device_name baud_rate number_of_motors
RS2_MM2000 = device_name baud_rate number_of_motors
GP_MM2000 = gpib_address number_of_motors
PC_MM2000 = base_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                <>TYPE
    YES    /dev/tty00           9600    4           Newport MM2000/3000 (Serial)
    YES    /dev/tty00           9600    4      Newport MM2000/3000 (Daisy Chain)
    YES                  6            4             Newport MM2000/3000 (GPIB)
    YES               0x280           4                Newport MM2000 (AT bus)
```

Optional parameters:

```
MOTPAR:dc_proportional_gain
MOTPAR:dc_derivative_gain
MOTPAR:dc_integral_gain
MOTPAR:dc_integration_limit
MOTPAR:dc_sampling_interval
MOTPAR:dc_following_error
MOTPAR:home_base_rate
MOTPAR:home_slew_rate
MOTPAR:home_acceleration
MOTPAR:slop
```

The Newport (formerly Klinger) MM2000 and MM3000 motor controllers are supported by spec on both RS-232C and GPIB interfaces. The MM2000 is also supported on the ISA bus interface. On the serial interface, spec supports the daisy chaining available on the MM2000 and MM3000 controllers. All these controllers can be used both with DC motors (with encoders) and with the 1.5M-type stepper motors.

Newport Motion Master 4000/4005 (GPIB and Serial)

config file:

```
RS_MM4000 = device_name baud_rate number_of_motors
GP_MM4000 = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                <>TYPE
    YES    /dev/tty00           9600    4          Newport MM4000/4005 (Serial)
    YES                  6            4            Newport MM4000/4005 (GPIB)
```

Optional parameters:

```
MOTPAR:dc_gain
MOTPAR:dc_damping_constant
MOTPAR:dc_integration_constant
MOTPAR:dc_following_error
MOTPAR:home_slew_rate
MOTPAR:home_acceleration
MOTPAR:slop
```

Before using the MM4000/4005 with spec, you need to set the communication parameters using the front panel buttons and display. The default communication

timeout of 0.5 seconds should be fine. Choose `CR` as the communication terminator for both GPIB and RS-232C interfaces. The SRQ feature of the GPIB interface is not used by spec, so the `IEEE SRQ` setting must be set to `NO`. For the RS-232C interface, the factory defaults for a parity setting of none, a word length of 8 bits and a stop-bits setting of 1 bit should be appropriate.

NSLS Brand MMC32 Controller (GPIB)

config file:

```
GP_MMC32 = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES                    5           32    NSLS Brand MMC32 Controller (GPIB)
```

Oregon Micro Systems (PC Board and VME)

config file:

```
PC_OMS = device_name number_of_motors INTR|POLL
PC_OMSP = base_address number_of_motors POLL
PC_OMSP58 = base_address memory_address number_of_motors POLL
PC_OMSV = VME_address number_of_motors IRQ_number|POLL
PC_OMSV58 = VME_address number_of_motors IRQ_number|POLL
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES    /dev/oms00           INTR    4        Oregon Micro Systems PCX/38/39
    YES                0x330    POLL    4 Oregon Micro Systems PCX/38/39 polled
    YES        0xe000  0x300    POLL    4      Oregon Micro Systems PC58 polled
    YES               0xfc00    IRQ5    8             Oregon Micro Systems VME8
    YES               0xf000    POLL    8            Oregon Micro Systems VME58
```

spec currently supports PC board and VME module Oregon Micro Systems motor controllers.

For the PC versions of spec, the PCX, PC38 or PC39 models may be used in two, four, six or eight motor configurations. (Note, the newer PC34 and PC48 models should be used with the PCX/38/39 configuration.) spec can operate with a CSS-supplied driver (on certain platforms) or completely from user level using I/O port polling. The driver does require a dedicated PC interrupt, and at present, is limited to support of only one board. If I/O port polling is used, spec allows use of multiple boards.

The driver is contained in the file *oms.c* in the *drivers* subdirectory of the spec distribution. See the *README* file in the that directory for instructions on installing the driver into the UNIX kernel.

There is no driver for the PC58 board. It is only supported in polled mode.

For VME, the VME8, VME44 and VME58 models are supported. The VME8 operates eight motors, while the VME44 operates four motors with encoders, although software options for encoders are not currently implemented in spec. Multiple OMS VME motor controllers can be used simultaneously and are generally operated in polled mode. Interrupt-driven mode is currently only supported with the National Instruments MXI-VXI controllers.

OMS motor controllers can have from two to eight motors. spec numbers the motors the OMS manuals designate `X`, `Y`, `Z`, `T`, `U`, `V`, `R` and `S` as 0 through 7, in that order.

The first example above selects the PC board with the driver node */dev/oms00*. The driver may be used in either interrupt or polled mode. Interrupt mode means the spec program will be interrupted when motors complete their motions or hit a limit. In polled mode, the `wait()` function must be called repeatedly to check the status of the motor. Interrupt mode generally gives better performance, although in earlier versions of spec, software problems could be overcome by using polled mode. A PC interrupt is always required when the driver is used, even when polled mode is selected.

The second example selects the PC board with I/O port polling, with the board's base address at `0x330`, and with four motors on the board.

The third example selects the PC58 board at I/O port `0x300`. The PC58 also require 4,096 bytes of low memory. The example configuration with the address entered as `0xE000` in the DEVICE column selects a real memory address of `0xE0000`, as the value in the configuration is multiplied by 16.

The fourth example selects the VME8 and VME44 modules, with the board's A16 base address jumpered at `0xFC00` and with the VME interrupt request jumpered for `IRQ5`. Any of the VME IRQ vectors may be selected as can be polled mode. If more than one VME OMS controller is being used, all must be in polled mode or all must use interrupts. Different boards may use the same interrupt, though.

The last example selects the VME58 model. Note, this model requires 4096 bytes of A16 address space, so valid addresses have one hexadecimal digit followed by three zeroes.

On the *motor* screen (M) of the configuration editor, all of the OMS controllers use the symbol `OMS` or `OMS_E` in the controller field of the screen. The latter indicates the motor is being used with an encoder.

Special Commands

The following special commands are available through the `motor_par()` function. The two letter commands are direct implementations of commands described in the OMS manual. Refer to that manual for more information. Not all commands are available on all versions of the OMS controllers or on all firmware versions for a particular controller.

`motor_par(`*`motor`*`, "PA", `*`mode`*`)` – If *mode* is 1, the controller turns motor power on before each move and off after the move (assuming motor power is controlled by the auxiliary output pins). If *mode* is 0, motor power stays on.

`motor_par(`*`motor`*`, "SE", `*`msec`*`)` – Sets the settling time in milliseconds to be used before the power is reduced in PA mode.

`motor_par(`*`motor`*`, "AF")` – Turns auxiliary power off.

`motor_par(`*`motor`*`, "AN")` – Turns auxiliary power on.

`motor_par(`*`motor`*`, "BH", `*`mask`*`)` – Sets general purpose output pins high, according to which of bits 0-13 in *mask* are set.

`motor_par(`*`motor`*`, "BL", `*`mask`*`)` – Sets general purpose output pins low, according to which of bits 0-13 in *mask* are set.

`motor_par(`*`motor`*`, "BX")` – Returns the state of the general purpose input pins. A one in any binary position in the value returned indicates that the corresponding pin is low.

`motor_par(`*`motor`*`, "RB")` – Returns the direction of the general purpose I/O lines. Output bits return a one, while input bits return a zero.

Command Pass Through

Command pass through is available using the following functions. Command pass through should be used with caution to avoid interfering with the built-in programming commands spec sends to the OMS controllers.

`motor_par(`*`motor`*`, "send", `*`cmd`*`)` – Sends the string *cmd* to the OMS channel associated with *motor*. For example, set *cmd* to `"LF"` to disable hardware limits on the associated motor.

`motor_par(`*`motor`*`, "read", `*`cmd`*`)` – Sends the string *cmd* to the OMS channel associated with *motor*, as above, and returns a string containing the response. For example,

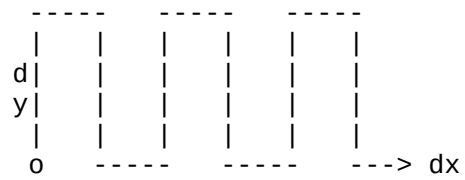
```
30.FOURC> print motor_par(tth, "read", "RP")
240000

31.FOURC>
```

results in the string `"AX RP\n"` being sent to the controller.

Asynchronous Surface Scanning

The following commands implement a special asynchronous, two-dimensional scanning mode available with newer versions of the OMS firmware. The scan is in the form of a repeating square wave, as illustrated below.



The scan starts at the point `o`, as specified with the commands below, and continues in the *x* and *y* directions in the range as specified with the commands below. At the end of the range, the motors are returned to the starting position and the scan is repeated.

Two motors must be configured with the mnemonics `dx` and `dy` in order for the asynchronous scanning mode to be available. When not in scanning mode, these motors may be moved normally.

Once started, the scanning will continue until explicitly stopped either with the `stop_scan` command (shown below), with a `^C` typed at the keyboard or with a `sync` command (which aborts the motors, but doesn't update spec's positions). While scanning, the `wait()` function will not indicate these motors are moving. The `getangles` command will, however, return the current positions of these motors.

When the `dx` and `dy` motors are scanning, the remaining motors may be moved independently.

`motor_par(`*`motor`*`, "x_start", `*`value`*`)` — Sets the starting position for the `dx` motor.

`motor_par(`*`motor`*`, "x_range", `*`value`*`)` — Sets the extent of the motion in the *x* direction.

`motor_par(`*`motor`*`, "x_stepsize", `*`value`*`)` — Sets the size of each step in *x*. The number of steps is determined by dividing this number into the range for *x*.

`motor_par(`*`motor`*`, "y_start", `*`value`*`)` — Sets the starting position for the `dy` motor.

`motor_par(`*`motor`*`, "y_range", `*`value`*`)` — Sets the extent of the motion in the *y* direction.

`motor_par(`*`motor`*`, "start_scan")` — Starts the asynchronous scan.

`motor_par(`*`motor`*`, "stop_scan")` — Stops the asynchronous scan.

Oriel Encoder Mike Controller 18011 (Serial)

config file:

```
RS_18011 = device_name baud_rate number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES    /dev/ttyh3     <>    4800    3   Oriel Encoder Mike Controller 18011
```

Oriel Encoder Mike Controller 18092 (Serial)

config file:

```
RS_18092 = device_name baud_rate number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES    /dev/ttyh3     <>    4800    3   Oriel Encoder Mike Controller 18092
```

Phytron IXEα-C (GPIB and Serial)

config file:

```
RS_IXE = device_name baud_rate number_of_motors
GP_IXE = gpib_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES    /dev/tty00           9600    2                  Phytron IXE (Serial)
    YES                  6            4                    Phytron IXE (GPIB)
```

Finding Reference Positions

The Phytron's axis initialization command `"0"`, which searches for the *negative* limit switch as described in the Phytron manual, is sent to a specific motor when the `chg_dial()` function is invoked with either the `"home+"` or `"home-"` arguments. The

free-axis-displacement commands "L+" and "L-" are sent with the "lim+" and "lim-" arguments, respectively.

There is no fixed hardware signal for a home switch on the Phytron controller, but there are twelve digital inputs available on the input connector. There is also a command which will perform a relative move at the base rate until one of the inputs goes high or low or the magnitude of the move is reached. There are four parameters in this command: the direction of the move, the magnitude of the move, the binary input number and the sense of the input switch. All four parameters are set by entering a string as *generic parameter 1* on the optional motor parameter screen of the configuration editor. (Get there by typing m twice from the standard motor parameter screen.) The string you enter will be sent to the motor when the "home" argument is used with the "chg_dial()" function. The string is the actual command sent to the Phytron, and is of the form

S mag vE *nn D*

where *S* is a + or a − for the sign of the move, *mag* is the magnitude of the move (maximum of 65535 steps), *nn* is the input number (01 through 12) and *D* is the sense of the input where 0 means the motor stops if the input goes off and 1 means the motor stops if the input goes on. For example,

```
+200vE071
```

would command the motor to move no more than 200 steps in the plus direction, or until input 7 goes ON.

Use the m command twice from the motor screen of the configuration editor to reach the screen where you can enter *generic parameter 1*. Type an initial single quote to enter a string.

Special Commands

On faster computers, the Phytron apparently cannot keep up with commands sent by the computer at full speed. You can slow down the communication between spec and the Phytron controllers with the following commands:

motor_par(*motor*, "rdelay" [, *value*]) — If *value* is given, sets the delay before reading a response from the Phytron to *value* seconds, otherwise returns the current value. The default value is 0.015.

motor_par(*motor*, "wdelay" [, *value*]) — If *value* is given, sets the delay before sending a command to the Phytron to *value* seconds, otherwise returns the current value. The default value is 0.015.

Only one copy of the *rdelay* and *wdelay* parameters is kept for all the Phytron controllers. The motor mnemonic *motor* can be associated with any of the Phytron controllers. The values for the parameters are saved in the state file, so should only need to be reset after starting fresh. (See page 57 in the *Reference Manual*.)

Command pass through is available using the following functions.

`motor_par(`*motor*`, "send", `*cmd*`)` — Sends the string *cmd* to the Phytron channel associated with *motor*.

`motor_par(`*motor*`, "read", `*cmd*`)` — Sends the string *cmd* to the Phytron channel associated with *motor*, as above, and returns a string containing the response.

`motor_par(`*motor*`, "usend", `*cmd*`)` — Sends the string *cmd* to the Phytron controller associated with *motor*.

`motor_par(`*motor*`, "uread", `*cmd*`)` — Sends the string *cmd* to the Phytron controller associated with *motor*, as above, and returns a string containing the response.

For example,

```
31.FOURC> print motor_par(tth, "read", "P20R")
240000

32.FOURC>
```

results in the string `"\002XP20R\003\r\n"` being sent to the controller. Command pass through should be used with caution to avoid interfering with the built-in programming commands spec sends to the Phytron controllers.

The following command is also available to help with debugging:

`motor_par(`*motor*`, "dump")` — Displays the values of Phytron parameters P01 through P10 and P12 through P17 for the channel associated with *motor*.

PC DAC as Motor Controller

config file:

```
PC_DAC_B12 = base_address number_of_motors
PC_DAC_T12 = base_address number_of_motors
PC_DAC_B16 = base_address number_of_motors
PC_DAC_T16 = base_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                                <>TYPE
    YES                 0x300            1     PC DAC 12-Bit D/A (binary output)
    YES                 0x310            1  PC DAC 12-Bit D/A (two's complement)
    YES                 0x320            1     PC DAC 16-Bit D/A (binary output)
    YES                 0x330            1  PC DAC 16-Bit D/A (two's complement)
```

PMC Corporation DCX-100 (Serial and PC Board)

config file:

```
RS_DCX = device_name baud_rate number_of_motors
PC_DCX = base_address number_of_motors
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                              <>TYPE
    YES    /dev/tty00           9600    4             PMC Corp DCX-100 (Serial)
    YES               0xD000            4           PMC Corp DCX-100 (PC Board)
```

Optional parameters:

```
MOTPAR:dc_proportional_gain
MOTPAR:dc_derivative_gain
MOTPAR:dc_integral_gain
MOTPAR:dc_integration_limit
MOTPAR:dc_sampling_interval
MOTPAR:dc_following_error
MOTPAR:slop
```

XIA HSC (Huber Slit Controller)

config file:

```
RS_XIAHSC = device_name baud_rate number_of_slits
```

edconf devices screen:

```
 MOTORS        DEVICE   ADDR  <>MODE  NUM                              <>TYPE
    YES    /dev/ttyS0     <>    9600    2                    XIA HSC-1 (Serial)
```

sample *edconf* motor screen:

```
Number: <>Controller        0:  XIAHSC   1:  XIAHSC   2:  XIAHSC   3:  XIAHSC
Unit/Channel                       0/0          0/1          0/2          0/3
Name                           Slit1 A      Slit1 B    Slit1 Gap    Slit1 Cen
Mnemonic                           s1a          s1b          s1g          s1c
Steps per degree/mm                400          400          400          400
```

The X-Ray Instrumentation Associates Huber Slit Controller Model HSC-1 is a specialized device only used to control Huber slits. The HSC-1 communicates through a serial port, and several HSC-1 modules can be daisy-chained together and run off a single serial port. On the device screen of the configuration editor, the `NUM` field is set to the number of HSC-1 modules on the serial port.

The HSC-1 module needs to be sent a calibration command before the HSC-1 motors can be moved with the normal move commands. The HSC-1 manual describes a manual calibration procedure. It is also possible to set the controller to the calibrated state by sending the `"calibrate"` command with `motor_par()` as described

below.

Each HSC-1 contains two motors that control the slit blades. Each blade can be moved independently. The HSC-1 also implements commands to move both blades simultaneously to change either the gap or the center-of-the-gap position. spec can be configured to control just the two blades, just the gap and the center position, or all four motions. When all four motions are configured, moving either blade changes the positions reported for gap and center, and moving either the gap or the center causes the positions reported for each blade to change.

Configuration for the HSC-1 requires the unit/channel field on the second line of the motor screen of the configuration editor to be filled in according to the following special format. The unit number corresponds to successive entries on the devices screen – each unit is associated with a different serial port. The channel number combines two values. Each HSC-1 module requires an arbitrary module number *N* (see below). This number is multiplied by 10 and added to the channel number that identifies the motion, as follows: For motor controller *N*, channel *N*×10 + 0 corresponds to blade A, channel *N*×10 + 1 corresponds to blade B, channel *N*×10 + 2 corresponds to the gap and channel *N*×10 + 3 corresponds to the center of the gap. Channel numbers ending in 4 through 9 are invalid.

Entering the Serial Number

The module number *N* (see above) is used only for internal bookkeeping and does not designate a particular HSC-1 module. Each module is identified by a thirteen-character unique serial number of the form `XIAHSC-B-0014`. The serial number needs to be entered as a string in the field *generic parameter 1* on the optional motor parameter screen of the configuration editor. Use the `m` command twice from the motor screen of the configuration editor to reach the screen where you can enter *generic parameter 1*. Type an initial single quote to enter a string. Note, the serial number can also be entered as `B-0014`, `0014` or `14` if such a string is sufficient to distinguish among modules. Also note, the serial number should be entered for just one of the motors associated with module *N*.

If the alias feature of the HSC-1 is used, and bit 6 of the *control word* (see below) is set for “use alias as ID”, spec requires there be no space characters in the alias. Of course, the alias, rather than the serial numbers would need to be entered as *generic parameter 1*. Setting and changing aliases requires establishing serial communication with the modules, which may be difficult for novice spec administrators, so CSS recommends simply using the serial numbers as the modules come from the factory.

Motor Parameters

The *steps per deg/mm* parameter should be set to 400 for the HSC-1 modules.

Normally, the positions for each blade become more positive as the blade is opened. However, if the *sign of user * dial* parameter is negative for either blade (or both), the motor position will become more negative as the blade opens. The sense of the center-of-the-gap motion can also be changed by changing the sign of the *sign of user * dial* parameter. The gap motion is always positive as the gap increases, though.

The backlash, speed and acceleration parameters in the *config* file are ignored.

CSS recommends using the calibration feature of the HSC modules to set the zero positions, rather than using the `set` macro to set the user *offset* parameter. That is, it is best to keep the user and dial positions the same. The `chg_dial()` function will, in fact, send the "immediate calibration" command to the controller, but only when setting the position to zero. Note, the gap should be physically at zero before using `set_dial`. Use of the standard `set_dial` macro should be followed by the `set` macro to set the user-dial offset back to zero.

The HSC-1 controller stores a number of parameters in nonvolatile memory. spec will read and display them with the command `motor_par(`*`motor`*`, "dump")`, where *motor* is the mnemonic for any of the motions on the particular HSC-1 module. The display format is as follows:

```
 1    Outer motion limit (rw) = 4400 (11 mm)
 2       Origin position (rw) = 400 (1 mm)
 3      Motor A position (ro) = 900 (2.25 mm)
 4      Motor B position (ro) = 900 (2.25 mm)
 5      Motor step delay (rw) = 200 (roughly 0.272 mm/sec)
 6         Gear backlash (rw) = 10 (0.25 mm)
 7          Control word (rw) = 142 (0x8e)
 8      Escape character (rw) = 33
 9  Arbitration priority (rw) = 8
10         Motor A phase (ro) = 0
11         Motor B phase (ro) = 0
12  Calibration complete (ro) = 150
13      EEPROM signature (ro) = 23205
14        EEPROM version (ro) = 4
```

These parameters can be modified using a command such as

```
motor_par(motor, "send", "W 6 20")
```

which changes memory location 6 (gear backlash) to 20.

Special Commands

The `motor_par()` options implemented for the HSC-1 are as follows:

`motor_par(`*`motor`*`, "calibrate")` — Sends the "immediate calibration" command to the unit. The effect is to set the current position of each blade in the controller to the *origin* parameter. It also sets spec's positions for the gap, center and blades to zero. Thus the gap should physically be at zero before sending this command.

`motor_par(`*`motor`*`, "origin", `*`value`*`)` — Sets the controller's *origin* parameter (parameter 2 of the controller's memory map) to *`value`*. The units of *`value`* are steps, where 400 steps corresponds to 1 mm. The origin parameter determines how far beyond the zero position each slit blade can be moved. Note, changing the origin parameter will change the setting of the current position of the blades. The blades should thus be both at zero before sending the `"origin"` command, and a `"calibrate"` command should be sent immediately afterwards. The factory default value for *origin* is 400.

`motor_par(`*`motor`*`, "range" [, `*`value`*`])` — Sets the controller's "outer limit" parameter (parameter 1 of the controller's memory map) to *`value`*, where the units of *`value`* are steps. This parameter controls how far each blade can be moved. The factory default value for this parameter is 4400.

`motor_par(`*`motor`*`, "step+")` or `motor_par(`*`motor`*`, "step-")` — Moves blade A or blade B one step in the specified direction. This command can be used to position the slits whether or not they have been calibrated.

`motor_par(`*`motor`*`, "send", `*`cmd`*`)` — Sends the string *`cmd`* to the HSC-1 unit associated with *`motor`*. The module serial number will be included automatically.

`motor_par(`*`motor`*`, "read", `*`cmd`*`)` — Sends the string *`cmd`* to the HSC-1 unit associated with *`motor`*, as above, and returns a string containing the response.

`motor_par(`*`motor`*`, "usend", `*`cmd`*`)` — Sends the string *`cmd`* to the serial port connected to the HSC-1 unit associated with *`motor`*. The *`cmd`* must include the full HSC-1 command syntax.

`motor_par(`*`motor`*`, "uread", `*`cmd`*`)` — Sends the string *`cmd`* to the serial port connected to the HSC-1 unit associated with *`motor`*, as above, and returns a string containing the response.

Timers and Counters

Timers and Counters

Am9513-based Counter/Timer PC Boards

config file:

```
PC_AM9513 = base_address number_of_counters
```

edconf devices screen:

```
SCALERS        DEVICE   ADDR  <>MODE  NUM                                <>TYPE
    YES                0x340            3        Am9513 Counter/Timer PC Boards
```

edconf scalers screen:

```
NUMBER        NAME  MNEMONIC  <>DEVICE  UNIT  CHAN   <>USE AS  SCALE FACTOR
     0     Seconds       sec    AM9513     0     0   timebase          1000
     1     Monitor       mon    AM9513     0     1    monitor             1
     2    Detector       det    AM9513     0     2    counter             1
```

The ComputerBoards CIO-CTR05/10/20 cards, the Keithley-Metrabyte Model CTM-05/10 cards, the Scientific Solutions Labmaster series cards, and similar models from other manufactures all use the Advanced Micro Devices Am9513 System Timing Controller chip. The chip contains five 16-bit counters that can be programmed in a wide range of configurations. spec's programming uses two of the counters for a 32-bit detector counter, two for a 32-bit monitor counter and one for a 16-bit elapsed time counter. On boards with two or four chips, the additional chips are each programmed for two more 32-bit detector counters. You can program the chip through spec to count to either a time preset using the `tcount()` function or a monitor-count preset using `mcount()`.

You must connect the detector to the input connector pin labeled *source 3*. Counts received from the monitor go to the pin labeled *source 5*. In addition, you must wire the connector pin labeled *output 1* to the pins *gate 2*, *gate 4* and *gate 5*. (In the new Keithley-Metrabyte CTM-05A manual, the *source* pins are now labeled *ACLKIN*, the *output* pins are now labeled *ATIMEROUT*, and the *gate* pins are now labeled *AGATE*.)

If it is a two- or four-chip board, the additional detectors are connected to the *source 3* and *source 5* pins of the chips. In addition, the *output 1* from the first chip must be also connected to *gate 2* and *gate 4* of the additional chips.

The counter boards are accessed from user level and are polled to determine when count intervals have elapsed. Thus, interrupts should be disabled on the boards.

You will need to enter the base address of the counter chip in the *config* file. Note that for the Labmaster board, the base address of the counter chip is eight plus the base address of the board itself.

When counting to time, the resolution of the clock depends on the length of the count interval. The maximum count time is 71.5 minutes. The time base resolution (in seconds) is set according to the following table:

0.00001	for $t < 0.6$ sec
0.0001	for $t < 6$ sec
0.001	for $t < 60$ sec
0.01	for $t < 655.35$ sec (10.9 min)
0.0655	for $t < 71.5$ min

When counting to monitor counts, the 0.01 second time base is used, and the value returned for the time channel will be corrected to account for the rollovers that occur every 655.36 seconds.

Bi Ra 5302 64-Channel ADC (CAMAC)

config file:

```
CA_BR5302 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    BR5302     0         BiRa 5302 ADC as counters
```

scalers screen:

```
NUMBER        NAME  MNEMONIC  <>DEVICE  UNIT  CHAN   <>USE AS  SCALE FACTOR
     0    Sensor 1      sen1    BR5302     0     0    counter             2
```

Up to 64 counters may be configured per ADC module. Each channel is 12-bits. Currently spec assumes a ±10 Volt range on each channel and scales the readings to that range. The scale factor from the *config* file is used to program the gain on the corresponding channel. The values returned by `getcounts` in the `S[]` array are scaled by the gain value. Allowed values for the gain are from 1 to 1024 in powers of two. If an illegal value is entered, spec uses the next lower legal value.

DSP RTC-018 Real Time Clock (CAMAC)

config file:

```
CA_RTC018 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    RTC018                DSP RTC-018 Real Time Clock
```

The Standard Engineering DSP RTC018 Real-Time Clock is wired as follows:

(1) If counting to time, the crystal oscillator output (2^{18}Hz) is connected to input A. If counting to the monitor, the monitor cable from one of the scaler inputs (usually channel 1) is connected to input A.

(2) The *preset out* output is connected to *start*.

(3) If using the DSP QS-450 scaler, connect the *busy* output to the *gate* input on the scaler module. If using the Kinetic Systems 3610 hex scaler, connect the *end* output to the *inhibit* input of the scaler module.

(4) A 1 KHz signal should be fed into a scaler input (normally channel 0).

(5) The detector signal should be fed into a third scaler input (normally channel 2).

DSP QS-450 4-Channel Counter (CAMAC)

config file:

```
CA_QS450 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1     QS450     0          DSP QS-450 4-Channel Counter
```

DSP TS-201 Dual Timer/Scaler (CAMAC)

config file:

```
CA_TS201 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1     TS201                DSP TS-201 Dual Timer/Scaler
```

Inel 715 Dual Scaler

config file:

```
RS_INEL = device_name baud_rate number_of_counters
```

Inel XRGCI as Timer/Counter

config file:

```
RS_XRGCI_T = device_name baud_rate number_of_counters
```

Joerger VSC16/8 Timer/Counter (VME)

config file:

```
PC_VSC16T = base_address number_of_counters INTR|POLL
```

edconf devices screen:

```
SCALERS       DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES              0x1000    POLL    8       Joerger VSC16/8 as Timer/Counter
    YES              0x1100    POLL    8            Joerger VSC16/8 as Counters
```

Note, you need to add two zeros to the value of the settings of the six hexadecimal digits on the module's address switches to form the A32 address entered in spec's configuration editor.

Note also, the *ARM IN* connector needs to be jumpered to the *ARM OUT* on the Joerger front panel. If more than one Joerger module is used, the one module designated as Timer/Counter is the master, and the *ARM OUT* from that module needs to be connected to the *ARM IN* of all the modules.

CSS recommends users order the Joerger scaler with a 1 MHz crystal oscillator rather than the 10 MHz oscillator normally provided. The 10 MHz oscillator only allows preset counting times of a bit more than seven minutes before the counter overflows. The oscillator can also easily be changed in the field. Its only purpose is to provide the front panel time-base output. In either case, a value corresponding to the oscillator rate must be entered into the configuration editor on the scalers screen for the scale factor for the channel corresponding to seconds.

Kinetic Systems 3610 6-Channel 50 MHz Counter (CAMAC)

config file:

```
CA_KS3610 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    KS3610     0          KS 3610 6-Channel 50 MHz Counter
```

Kinetic Systems 3640 Used as Counter or Timer (CAMAC)

config file:

```
CA_KS3640T = slot_number
CA_KS3640C = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1   KS3640T                KS 3640 Counter used as Timer
   2   KS3640C     0          KS 3640 Counter used as Counter
```

In order to use a 3640 as a timer to gate other 3640 modules, you need to make modifications. The modifications will leave one of the front panel inhibit inputs alone and convert the other to an inhibit output. One way to do this is to add an LM311 comparator IC to the circuit. The negative input (pin 2) of the 311 is connected to the LAM signal from pin 9 of IC 30 of the 3640 module. The positive input (pin 3) is held high at about 3.2V through a 1200 ohm over 2200 ohm voltage divider between +5V and ground. Pins 1 and 4 of the 311 are connected to ground. Pin 8 is connected to +5V. The output of the 311 (pin ?) is connected to the front panel LIMO connector, which must have the factory connection cut. In addition, a 2K Ohm resistor is connected between the output and +5V to pull up the output.

In operation, the inhibit inputs of all the 3640s are connected to the inhibit outputs of all the 3640s. Thus any module can be used as the gate. One module should be fed a fixed time base, say 1KHz or 10KHz which must come from some external source.

config file:

```
CA_KS3655 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    KS3655                KS 3655 8-Channel Timing Generator
```

Current users of spec have made the following modifications to the Kinetic Systems Model 3655 Timing Generators. This timing generator is customarily used with the Kinetic Systems Model 3610 Hex Scaler. All modifications are made on the component side of the board.

(1) Bring the internal 1 KHz timing signal out through the channel 7 front-panel lemo connector. Do this by first unsoldering the end of the wire that connects the center pin of the channel 7 lemo connector from the feed through on the circuit board. Do not unsolder the wire from the connector, as it will be difficult to solder on a new wire. Instead splice a longer wire to the one already attached to the connector and solder the other end of that wire to pin 11 of chip BJ. (Pin 11 is the center pin on the front-panel side of the chip.)

(2) Disconnect the internal inhibit signal from the CAMAC dataway, and bring it out through the channel 8 lemo connector. Do this by folding up and/or snipping pin 6 of the socketed 7407 chip in position BX, near the front panel. Next either fold up pin 3 of chip AU or cut the long trace that leads from that pin to the dataway Inhibit connector. Then solder a wire from pin 3 of chip AU to the bottom lead of resistor R45. The resistor is located near the top-right corner of the circuit board, and the bottom lead is the one nearest the letters R45.

Channel 7 is then connected to channel 0 of the scaler module, while Channel 8 is connected to the *inhibit* input. The signal from the source of monitor counts is connected, using a tee, to the *clock* input of the 3655 and the monitor scaler channel (usually channel 1; the detector is usually channel 2).

config file:

```
RS_OR9XT = device_name baud_rate number_of_counters
GP_OR9XT = gpib_address number_of_counters
RS_OR9XC = device_name baud_rate number_of_counters
GP_OR9XC = gpib_address number_of_counters
RS_OR9XB = device_name baud_rate number_of_counters
GP_OR9XB = gpib_address number_of_counters
```

edconf devices screen:

```
SCALERS       DEVICE   ADDR  <>MODE  NUM                                  <>TYPE
    YES    /dev/tty1           9600    4   Ortec 974/994 Counter/Timer (Serial)
    YES                    3           4     Ortec 974/994 Counter/Timer (GPIB)
    YES    /dev/tty2           9600    2 Ortec 974/994/995/997 Counter (Serial)
    YES                    3           2   Ortec 974/994/995/997 Counter (GPIB)
    YES    /dev/tty3           9600    3 Ortec 994 Blind Timer/Counter (Serial)
    YES                    3           3   Ortec 994 Blind Timer/Counter (GPIB)
```

edconf scalers screen:

```
NUMBER        NAME  MNEMONIC  <>DEVICE  UNIT  CHAN   <>USE AS  SCALE FACTOR
     0     Seconds       sec     OR9XX     0     0   timebase          1000
     1     Monitor       mon     OR9XX     0     1    monitor             1
     2    Detector       det     OR9XX     0     2    counter             1
```

spec supports the Ortec 974, 994, 995 and 997 counter and counter-timer NIM modules over both GPIB and RS-232 interfaces. When running the configuration editor, select from the above descriptions on the device configuration screen to specify which Ortec modules you are using and how you are using them.

Only one module can be selected as a counter/timer. The 974 module can be assigned a maximum of four channels. The 994 should be assigned two channels normally and three channels when used as a blind timer. The 995 has two channels and the 997 has one.

On the scaler configuration screen, choose `OR9XX` as the controller for all channels associated with an Ortec module. The unit numbers selected for each channel correspond to the order the Ortec modules appear on the device configuration screen. When using the 994 as a blind timer, you must select channel number 2 for the timebase.

Using the 974

The 974 is a four-channel counter/timer having a minimum 0.1 second time base. You should connect the monitor counts through a tee to the EXT IN connector on the back of module and to the COUNTER INPUT 2 connector on the front of the module. Use the COUNTER INPUT 3 and 4 connectors for one or two detector input

channels. Also, make sure that the internal dip switch S-1 has position 6 set to one-cycle.

Using the 994 as a Normal Timer

The 994 is a two-channel counter/timer with a minimum 0.01 second time base. In order to obtain accurate elapsed time readings, one counter channel is used to count time and the other is used to count monitor counts. An additional counter, such as the 995 or 997 is normally used to accumulate detector counts and is gated by the 994. The monitor count source should be connected to both the IN A and IN B front panel connectors of the 994 using a tee. The internal jumpers W3 and W4 must both be set to the TIME position. Jumper W1 must be set to the NORMAL position. Also, make sure the internal dip switch S-1 has position 6 set to one-cycle and position 7 set to COUNTER/TIMER. Finally, make sure the front panel DWELL switch is turned all the way off.

Using the 994 as a Blind Timer

In the blind timer mode, the 994 has the monitor counts connected to IN A and detector counts connected to IN B. The internal jumpers W3 and W4 must both be set to the COUNTS position. Jumper W1 must be set to the NORMAL position. Also, make sure the internal dip switch S-1 has position 6 set to one-cycle and position 7 set to COUNTER/TIMER. Finally, make sure the front panel DWELL switch is turned all the way off.

When operated as a blind timer, spec cannot read back the elapsed time from the module. Instead, when counting to monitor counts, when counting in powder mode, when reading the counters during updated counting and when counting is aborted with a `^C`, the elapsed count time is estimated from the software clock.

Gating

An external enable signal from certain motor controllers may be be fed into the rear-panel gate BNC input on the 974 or the front panel enable BNC input on the 994 for precise counter gating in powder-mode scans.

If using a second Ortec module as a counter, you must connect the INTERVAL BNC connector (rear panel on 974, front panel on 994) to the master GATE on the 974 rear panel or to the ENABLE or individual GATE inputs on the 994, 995 or 997 modules.

Setting Operational Parameters

The `counter_par()` function can be used to set various parameters associated with the Ortec module code in spec. The first argument to `counter_par()` is a channel number, although all the commands affect all channels of the associated module, or all of the Ortec modules, if appropriate.

`counter_par(`*`counter`*`, "alarm", `*`mode`*`)` – If *mode* is zero, turns off the more efficient *ALARM* mode of operation of the timer, and turns on a slower polled mode. If *mode* is one, *ALARM* mode is turned on. The default operation is for *ALARM* mode to be turned on, and there is generally no reason to turn it off.

`counter_par(`*`counter`*`, "alarm")` – Returns one if *ALARM* mode is on. Otherwise returns zero.

`counter_par(`*`counter`*`, "display", `*`channel`*`)` – Sets the counter channel that will be displayed on the associated module. For the 974 modules, valid values for *channel* are 1 to 4. For the the 994 and 995 modules, valid values for *channel* are 0 and 1.

`counter_par(`*`counter`*`, "display")` – Returns the channel number currently being displayed.

`counter_par(`*`counter`*`, "local", `*`mode`*`)` – If *mode* is nonzero, will force the associated module to go into local mode to allow front panel operation. In addition, the module will be placed in local mode after each count interval. If *mode* is zero, the module will be set to remote mode at the start of the next count interval, and will not be set back to local mode after counting. When spec starts up, not switching to local mode is the default behavior to minimize overhead.

`counter_par(`*`counter`*`, "local")` – Returns zero if the associated module is to be kept in remote mode. Otherwise, returns one.

Software Timer

config file:

```
SW_SFTWARE = 1
```

If no hardware timer is available, the system clock can be used as a timer. Only counting to time is allowed, as counting to monitor makes no sense. The nominal resolution depends on the underlying operating system, although 10 msec is typical. The accuracy, though, is certainly less than that.

Multichannel Data Acquisition Devices

MCA Devices

DSP 2190 MCS Averager

config file:

```
CA_DSP2190 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1   DSP2190               DSP 2190 MCS Averager
```

The DSP Technology 2190 Multichannel Scaling Averager consists of a pair of CAMAC modules: the 2090S Multichannel Scaling module and the 4101 Averaging Memory module. These two modules must occupy consecutive slots in the CAMAC crate, with the 2090S in the lower-numbered slot. There is no entry for the 4101 module in the *config* file.

Functions

The `mca_par()` function controls the module's behavior as follows:

`mca_par("run")` — programs the MCS for the number of bins and sweeps set with the functions described below, then enables any other counters and starts the averaging process. When the programmed number of sweeps is completed, the MCS will generate a CAMAC LAM, which will cause the other counters to be disabled. Use the `wait()` function to determine when the programmed number of sweeps are complete.

`mca_par("halt")` — halts the MCS and disables the other counters.

`mca_par("bins")` — returns the number of bins in each sweep. (Referred to in the module documentation as "record length").

`mca_par("bins", value)` — sets the number of bins in each sweep to *`value`*. The number of bins can range from 8 to 32,767.

`mca_par("sweeps")` — returns the number of sweeps to be summed.

`mca_par("sweeps", value)` — sets the number of sweeps to be summed in the next scan to *`value`*. The number of sweeps can range from 1 to 65,536.

`mca_par("sweeps_comp")` – returns the number of sweeps completed in the previous scan. An error message is printed if this function is called while a scan is in progress.

`mca_par("first_ch")` – returns the first channel to be read out using `mca_get()`.

`mca_par("first_ch", value)` – sets the first channel to be read out using `mca_get()` to *value*.

`mca_par("npts")` – returns the number of channels to be read out using `mca_get()`.

`mca_par("npts", value)` – sets the number of channels to be read out using `mca_get()` to *value*.

Note that the `mca_get()` function cannot be used while the MCS module is taking data.

Note that the 4101 doesn't actually average the sweeps, but only accumulates sums in each channel. To obtain an average, you must divide the data in each channel by the number of sweeps. The averaging scan will halt before the programmed number of sweeps is completed if any of the channels overflow.

The module expects an external trigger and the external trigger is required to begin each sweep.

LeCroy 2301 interface for qVT MCA

config file:

```
CA_LC2301 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    LC2301                LeCroy 2301 interface for qVT MCA
```

Functions

The `mca_par()` function controls the MCA module's behavior as follows:

`mca_par("clear")` – clears the MCA. spec inserts a 1.5 second delay to give the device time to clear.

`mca_par("run")` – starts the MCA.

`mca_par("halt")` – stops the MCA.

`mca_par("first_ch")` – returns the first channel to be read out.

`mca_par("first_ch", value)` — sets the first channel to be read out to *value*.

`mca_par("npts")` — returns the number of channels to be read out.

`mca_par("npts", value)` — sets the number of channels to be read out to *value*. The maximum number of channels is 1,024.

`mca_par("delay")` — returns the delay time in seconds that spec sleeps after the MCA is cleared.

`mca_par("delay", value)` — sets the time for spec to delay after sending the clear command. The hardware does require some delay. Some users have reported 1.5 seconds are needed, others report 0.1 seconds is adequate. The default value is 0.1 seconds.

LeCroy 3512 Spectroscopy ADC

config file:

```
CA_LC3512 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    LC3512                LeCroy 2301 interface for qVT MCA
```

LeCroy 3588 Fast Histogram Memory

config file:

```
CA_LC3588 = slot_number
```

edconf CAMAC screen:

```
Slot    Module  Unit          Description
   1    LC3588                LeCroy 3588 Fast Histogram Memory
```

Keithley 2001 Multimeter (GPIB)

config file:

```
 GP_K2001
```

config file:

```
GP_PCA_M = gpib_address
PC_PCA3  = base_address
PC_PCAII = device_name base_address INTR|POLL
```

edconf devices screen:

```
MCAs           DEVICE   ADDR  <>MODE                                  <>TYPE
    YES                    7                    The Nucleus PCA Multport (GPIB)
    YES                0x210                     The Nucleus PCA-3 MCA Board
    YES                0x1e0    POLL                The Nucleus PCA II MCA Board
    YES      /dev/pca  0x1e0    INTR                The Nucleus PCA II MCA Board
```

The PCA II MCA can be used in either a user-level I/O mode or in an interrupt-driven mode with the CSS provided driver. The interrupt-driven mode allows dead-time corrections and more accurate counting times. See the *drivers/README* file in the spec distribution for information on installing the driver.

If using the interrupt-driven mode, note the following: Apparently, the PCA II doesn't trigger an interrupt on some PC mother boards. This problem can be fixed by changing the value of the resistor labeled R12 on the "PCA2 Memory Card" circuit diagram. This resistor is located near the lower left corner of the main board when viewed from the component side with the connector fingers pointing down and the input BNC to the right. R12 is about a centimeter down and to the left of the U26 IC. The circuit diagram indicates the resistor's value is 2K, however the boards seem to be shipped with a 1K resistor (brown-black-red stripes). Soldering a second 1K resistor alongside R12 and in parallel electrically will lower the resistance to 0.5K, which seems to work. (This modification was suggested by the manufacturer.)

Functions

The `mca_par()` function controls the board's behavior as follows:

`mca_par("clear")` — clears the channels of the current group.

`mca_par("run")` — programs the board with the current parameters and starts acquisition. Note that the `tcount()` and `mcount()` functions, as used in the various counting macros will also start PCA II acquisition.

`mca_par("halt")` — stops acquisition. Note that the PCA II will also be halted when the `tcount()` and `mcount()` functions, as used in the various counting macros, complete their count intervals or are aborted.

`mca_par("group_size")` — returns the current group size.

`mca_par("group_size", size)` — sets the group size to *size*. Legal values are 256, 512, 1024, 2048, 4096 and 8192. Values above 1024 may not be legal if insufficient memory is installed on the board.

`mca_par("select_group")` — returns the currently active group. Groups are numbered starting at zero.

`mca_par("select_group", group)` — set the active group to *group*. The number of possible groups is given by the total number of channels on the board divided by the group size. If the *group* passed to the function is greater than the maximum number of groups (based on the current group size and total number of channels), the current group selected is *group* modulus the maximum number of groups.

`mca_par("pha")` — selects pulse-height analysis mode on the board.

`mca_par("gain")` — returns the current gain value used in pulse-height analysis mode.

`mca_par("gain", value)` — sets the pulse-height analysis gain to *value*. Legal values are 256, 512, 1024, 2048, 4096 and 8192.

`mca_par("offset")` — returns the current channel offset used in pulse-height analysis mode.

`mca_par("offset", value)` — sets the pulse-height analysis offset to *value*. Legal values are multiples of 256 from 0 to 7936.

`mca_par("mcs")` — selects multichannel scaling mode on the board.

`mca_par("dwell")` — returns the current multichannel scaling dwell time.

`mca_par("dwell", value)` — set the multichannel scaling dwell time. Allowed values are numbers between 1e-6 and 60 seconds with mantissa of 1, 2, 4 or 8. A value of -1 selects external dwell. If *value* isn't an allowed value, it is rounded to the nearest allowed value.

`mca_par("mode")` — returns two if the board is in PHA live-time mode, one if the board is in PHA real-time mode and zero if the board is in MCS mode.

`mca_par("readone", channel)` — returns the contents of channel number *channel*.

`mca_par("chan#")` — returns the contents of channel number *#*. The channel number is with respect to the current group.

`mca_par("chan#", value)` — sets channel *#* to *value*. The channel number is with respect to the current group.

The following `mca_par()` functions are only valid when the board is used with the interrupt-driven driver.

`mca_par("preset")` – in PHA mode, returns the current live-time or real-time preset value in seconds.

`mca_par("preset", value)` – in PHA mode, sets the current live-time or real-time preset value to *value* seconds.

`mca_par("passes")` – in MCS mode, returns the number of preset passes.

`mca_par("passes", value)` – in MCS mode, sets the number of passes to *value*.

`mca_par("live")` – in PHA mode, selects live-time counting.

`mca_par("real")` – in PHA mode, selects real-time counting.

`mca_par("dead")` – in PHA mode, returns the percent dead time, if accumulating in live-time mode.

`mca_par("elapsed_live")` – in PHA live-time mode, returns the elapsed live time in seconds.

`mca_par("elapsed_real")` – in PHA mode, returns the elapsed real time in seconds.

`mca_par("elapsed_passes")` – in MCS mode, returns the elapsed number of passes.

Silena CATO MCA (Serial)

config file:

```
RS_CATO
```

Nicomp TC-100 Autocorrelator (Serial)

config file:

```
RS_TC100 = device_name baud_rate
```

The Nicomp TC-100 Autocorrelator is selected in the *config* file in the MCA section of the devices screen.

Functions

The `mca_par()` function controls the correlator behavior as follows:

`mca_par("clock")` – returns the value of the current clock time parameter in microseconds.

`mca_par("clock", value)` – sets the clock time parameter. The units for *value* are microseconds. Valid clock times are of the form *X.XeY* where *X.X* ranges from

0.1 to 1.6 and *Y* ranges from 0 to 5. Values outside these bounds will be rounded to the closest allowed value. The new value takes effect on the next `run` command.

`mca_par("prescale")` — returns the value of the prescale factor.

`mca_par("prescale", value)` — sets the value of the prescale factor. Valid prescale values are from 1 to 99. The new value takes effect on the next `run` command.

`mca_par("dbase_mode")` — returns the state of the baseline mode. A return value of 1 means delayed baseline mode is in effect. A return value of 0 means delayed baseline mode is off.

`mca_par("dbase_mode", 1|0)` — sets the state of the baseline mode. A value of 1 turns on delayed-baseline mode. A value of 0 turns it off. The new mode takes effect on the next `run` command.

`mca_par("dbase")` — returns the value of the delayed baseline from the last data obtained using `mca_get()`.

`mca_par("cbase")` — returns the value of the calculated baseline from the last data obtained using `mca_get()`.

`mca_par("tcnts")` — returns the value of the total-counts monitor channel from the last data obtained using `mca_get()`.

`mca_par("pcnts")` — returns the value of the total-prescaled-counts monitor channel from the last data obtained using `mca_get()`.

`mca_par("rtime")` — returns the value of the run-time monitor channel from the last data obtained using `mca_get()` in seconds.

`mca_par("clear")` — clears the correlator.

`mca_par("run")` — sends the current clock-time, prescale and delayed-baseline parameters to the correlator and starts the correlator. The `tcount()` and `mcount()` functions also start the correlator.

`mca_par("halt")` — stops the correlator. The correlator is also halted when count intervals specified by `tcount()` or `mcount()` have elapsed, or when counting is aborted using a `^C`.

`mca_par("plot")` — reads off the real-time data plot from the running correlator. The data obtained is a very low resolution version of the correlation function.

`mca_get(grp, el)` — reads the current data from the correlator, and stuffs the data into the data group *grp* element *el*.

REFERENCES

Journal articles describing the various supported X-ray diffractometers are available in the following references.

The four-circle diffractometer is discussed in

W. R. Busing and H .A. Levy, *Acta Cryst.* **22**, 457 (1967).

There is an error in Equation (48) of the above paper. The last line of the equation should be $\omega = \text{atan}(-R_{23}, R_{13})$.

Surface diffraction using a four-circle diffractometer is discussed in

S. G. J. Mochrie, *J. Appl. Cryst.* **21**, 1-4 (1988).

The z-axis diffractometer is described in

J. M. Bloch, *J. Appl. Cryst* **18**, 33-36 (1985).

The liquid-surface diffractometer supported by spec is described in

J. Als-Nielsen and P. S. Pershan, *Nucl. Instrum. Methods* **208**, 545 (1983).

A. H. Weiss, M. Deutsch, A. Braslau, B. M. Ocko, and P. S. Pershan, *Rev. Sci. Instrum.* **57** (10), 2554 (1986).

Angle calculations and operating modes for a six-circle diffractometer are presented in

M. Lohmeier and E. Vlieg, *J. Appl. Cryst.* **26**, 706 (1993).

A description of the CAMAC driver can be found in

G. Swislow, A. Braslau and S. G. J. Mochrie, *Interrupt-Driven CAMAC Software for* UNIX*-Based Computers*, AT&T Bell TM# 11115-870817-39.

INDEX

special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

X

Y

Z

NAME

spec – X-ray diffractometer control and general data acquisition package

SYNOPSIS

spec [**–fhsvyFLS**] [**–d** *debug*] [**–g** *geometry*] [**–l** *outputfile* [...]] [**–o** *option=value* [...]] [**–p** *fd pid*] [**–t** *tty*] [**–u** *user*] [**–C** *file* [[...] [**–D** *directory*] [**–N** *my_name*] [**–S** [*p1*] | **–S** *p1-p2*] [**–T** *fake_tty*]

DESCRIPTION

spec provides a software environment for the operation of an X-ray diffractometer and other data-acquisition instruments. spec contains a sophisticated command interpreter that uses a C-like grammar and powerful macro language. spec supports a variety of X-ray diffractometer configurations. The diffractometer geometry is chosen by the program name. Those currently supported include:

spec – Generic instrument control
fourc – Standard four-circle diffractometer
twoc – Standard two-circle diffractometer
sixc – Six-circle diffractometer (δ, θ, χ, ϕ, μ, γ)
psic – An S4-D2 six-circle diffractometer
kappa – Kappa diffractometer
surf – Various liquid surface diffractometers
zaxis – Standard z-axis diffractometer

The following options are recognized:

–C *file* Open the command file *file* as a start-up command file to be read after the standard start-up command files, but before the optional file *spec.mac* in the current directory, which will always be read last. If there is an error in reading or executing the commands in these files, spec will jump to the main prompt and not read any remaining queued command files. Up to 32 files may be specified with multiple **–C** options.

–d *debug* Set the initial value of the debugging variable DEBUG to *debug*. The available debugging categories are described on page 63 in the *Reference Manual*. A value of 192 is useful for debugging hardware problems.

–D *direc* Use *direc* instead of the compiled-in name (usually */usr/local/lib/spec.d*) or the SPECD environment name as the auxiliary file directory.

–f Fresh start. All symbols are set to their default values and the standard macros are read to establish the default state. Command-line history is reset unless the **–h** flag is also present.

–F Clean and fresh start. All symbols are set to their default values but no command files are read and no macros are defined. Only the built-in commands are available.

–g *geometry*
Load macro files and activate geometry calculations for the specified geometry, while using the configuration files taken from the name by which spec is invoked.

–h Retain history. When starting fresh, reset symbols and macros but keep command-line history. (Added in spec release 6.05.01.)

–l *logfile* Specify an output file. Output to the file will begin immediately, so will include the initial hardware configuration messages. Files will be opened even when starting fresh. Files opened this way will not be saved as output files in the state file, so will not be automatically reopened the next time spec starts. (As of spec release 6.04.05.)

–L Do not check or create the state-file lock. Normally, spec prevents more than one instance of itself from running with the same state file (derived from the user name plus tty name). With some system configurations, if the state file resides on an NFS-mounted disk, the file locking doesn't work well and spec will not start. This flag overrides the lock test.

–N *my_name*
Use *my_name* for setting the interactive prompt and the name of the directory containing the *config*, *settings* and state files. Normally the name by which spec is invoked is used.

-o *option=value*
Initialize the `spec_par()` *option* to *value*. The available `spec_par()` options are described on page 83 in the *Reference Manual*.

–p *fd pid* Indicates that spec input is coming from a pipe from another program. The argument *fd* is the file descriptor that spec should use for standard input. The argument *pid* is the process ID of the spawning process. If *fd* is zero, spec will not re-echo input from the file descriptor to spec's standard output.

–s Simulation mode. No hardware commands are issued. If started in simulation mode, simulation mode cannot be turned off without restarting the program.

–S Start spec in server mode listening at the first available port in the default range of 6510 to 6530.

–S *p1* Start spec in server mode listening at the specified port number *p1*.

–S *p1-p2* Start spec in server mode listening on the first available port in the given range.

–t *tty* Use the current user (or *user*'s) last saved state from the terminal specified by *tty*. The terminal can be specified as **–t** */dev/tty01* or **–t** *tty01*.

–q Indicates that spec should operate in quiet mode and allow output to all devices to be turned off. This option is only valid when used with the **–p** option.

–T *fake_tty*
This option creates a new user state with a name derived from *fake_tty*, which may be any name. This option allows you to bypass the state-file lock that prevents multiple instances of spec to be started by the same user from the same terminal.

–u *user* Use *user*'s last saved state as the current user's initial state. The **–t** flag will also be needed if the user was at a different terminal.

–v Print version information and exit.

–y "Yes", change motor controller registers initially if they disagree with the *settings* file. Normally, spec requires you to confirm such a change. This flag would be useful if you know controller power had been turned off, clearing the hardware memory.

ENVIRONMENT

spec uses the following environment variables:

SPECD An auxiliary file directory to use instead of the compiled in name.

TERM or **term**
The text terminal type.

GTERM The graphics terminal type for high-resolution graphics.

HOME The user's home directory.

SHELL or **shell**
The shell program to be used for interactive subshells.

DISPLAY
The display name and screen number on which to display the X-window plots.

FILES

(***SPECD*** is the auxiliary file directory, normally */usr/local/lib/spec.d*.)
(***geom*** is the first four letters of the name by which spec was invoked.)
(***spec*** is the complete name by which spec was invoked, as in *fourc*, *twoc*, etc.)
(If using a virtual terminal, ***tty*** is always **ttyp#** on *Linux* and **ttys00** on *macOS*.)

./spec.mac	Optional private command file always read at start up.
SPECD*/site.mac*	Optional site command file always read at start up.
SPECD*/site_f.mac*	Optional site command file only read when starting fresh.
SPECD*/standard.mac*	Standard macro definitions.
SPECD*/**geom**.mac*	Geometry macros.
SPECD*/**spec**/**geom**.mac*	Possibly more geometry macros.
SPECD*/**spec**/config*	Hardware configuration file.
SPECD*/**spec**/settings*	Motor settings file.
SPECD*/**spec**/conf.mac*	Optional configuration command file always read at start up.
SPECD*/**spec**/userfiles/hdw_lock*	Spectrometer lock file.
SPECD*/**spec**/userfiles/**user_tty**S*	User's state file. Uses only first six letters of ***user*** and of ***tty***.
SPECD*/**spec**/userfiles/**user_tty**H*	User's history file.
SPECD*/**spec**/userfiles/**user_tty**L*	User's lock file.
SPECD*/**spec**/userfiles/**user_tty**P*	User's data points file.
SPECD*/spec_help/**	Help files.