

NAME

spec server/client – issue commands and control hardware remotely

INTRODUCTION

Server mode `spec` allows communication between instances of `spec` on different platforms or between `spec` and other client software. The server protocol includes special commands for motor and counter control, and general commands for transfer of information. When acting as a client, `spec` can connect to a `spec` server and be configured for motors and counters that are on the server, but that behave as if they were local.

The `spec` server can send asynchronous messages to the client when certain events occur. Clients can arrange to receive messages when hardware-related events occur (such as a motor position change), when a variable is assigned a new value or when certain server state changes occur. Clients can have commands executed on the server and retrieve the result. The `spec` client implements both synchronous and asynchronous retrieval of results.

The new `spec` user-level commands for server/client communications are described below, as is the protocol used by the `spec` server. Although, `spec` will very often be the client, the communication protocol is open so arbitrary clients can be created.

See the *Updates* section at the end of this file for a summary of changes to the protocol associated with particular `spec` releases.

Starting the Server

The `spec` server mode is started by invoking `spec` with the `-S` flag (or `-S1` or `-S2`, see below) one of three ways. In the first instance, a specific TCP/IP port number can be specified, as in:

```
shell> fourc -S 6789
```

The server will attempt to listen for incoming connections at the specified port number. Only one server can listen at a particular port number at a time.

The second invocation specifies a range of port numbers, as in:

```
shell> fourc -S 5320-5330
```

The server will use the first port number in the range that is available.

Finally, server mode can be invoked with no port number arguments, as in:

```
shell> fourc -S
```

In this case, `spec` will use its default port number range, which is 6510 to 6530.

Port numbers are arbitrary, but must be known by the client and must not conflict with any other services currently configured. (See www.iana.org for an official list of port number assignments.) Port numbers below 1024 should be avoided, as they require root privilege for binding.

There are currently two server modes possible (as of `spec` release 5.07.04-4). The difference in the two modes relates to the handling of motor-limit events. The first mode, activated with `-S` or `-S1`, is the original mode. In this mode, unlike a standard non-server mode `spec` session, if a motor hits a limit, although that motor is stopped, `spec` does not stop other motors that may be moving nor does `spec` reset to command level. With a mode-2 server, activated with `-S2`, behavior on hitting a limit is the same as when operating in non-server mode. Note, in both modes events are sent to registered clients when a limit is hit. Also, a `spec` client will send a command to a `spec` server to abort all motors and reset to command level in any case. Note also, some motor controllers implement an emergency stop and/or a motor fault condition, which results in nearly the same actions as hitting a limit. (The only differences are in the text of the messages displayed and the name of the event sent to clients.)

When started in server mode, `spec` creates three threads: one to listen for client connections and read command requests from clients, one to write replies to clients and one to execute commands received from the client or typed in to the interactive `spec` prompt.

Although the server provides an interactive prompt (at least in the current implementation), the server has only one command thread. Thus, if the command thread is busy running a command entered at the server prompt, commands sent by remote clients will remain in the command queue until the interactive command has finished executing. Certain client requests, such as requests to get current motor positions don't put commands on the queue, as the requests are instead satisfied using existing data.

CSS recommends that interactive use of the server be limited. If an interactive `spec` session is required on the same host on which the server runs, it may be better to configure a second `spec` instance on the host that will be a client to the server `spec` session.

Most motor controller types and all counter/timers associated with a `spec` server can be controlled by a client. Certain motor controllers (namely the 18011, CM3000, CM4000, E500, ES_OMS, ES_VPAP, IP28, MC4, MCB, SIX19, SMC and XRGCI_M) are not currently supported by the `spec` server for remote control. Contact CSS to discuss the feasibility of adding server support for any of the above controllers.

Configuring the spec Client

`spec` server motor and timer/counter controllers are selected on **Devices** screen of the `edconf` configuration editor on the client along the following lines:

Motor and Counter Device Configuration (Not CAMAC)

MOTORS	DEVICE	ADDR	<>MODE	NUM	<>TYPE
YES	host_A:fourc			16	SPEC Motor (TCP/IP)
YES	host_A:specX			16	SPEC Motor (TCP/IP)
YES	host_B:6789			16	SPEC Motor (TCP/IP)

SCALERS	DEVICE	ADDR	<>MODE	NUM	<>TYPE
YES	host_A:specX			4	SPEC Timer/Counter (TCP/IP)
YES	host_B:6789			4	SPEC Counters Only (TCP/IP)

The **DEVICE** entry contains the host name or IP address of the server followed by a colon and either the name of a `spec` process that is being run in server mode or a port number on which the server is listening.

If a `spec` process name is given, the `spec` client will try each of the port numbers in the default range (6510 to 6530) and look for a matching server.

(Note, in the current implementation, a client must reference a particular server consistently by port number or by `spec` process name. The two methods cannot be mixed. The same method must be used both in the `config` file and with the user-level commands described below.)

The scaler device can be configured as either a timer/counter, making that device the master timer, or as counters only. When configured as counters only, the associated `spec` client scaler channels passively mirror the values on the server.

On the **Motor** and **Scaler** screens, the controller types should be selected as `SPEC_M2` and `SPEC_SC` respectively. The specific motor or counter channel on the server is selected by the mnemonic. That is, the client and server have to use the same mnemonic for the same motor or counter channel.

Server/Client spec Built-in Functions

In the following functions, *host* is a string of the form "remotehost:6789" or "remote-host:spec" where remotehost is the host name or IP address of the server, 6789 is a specific port number on which the server is listening (given by the `-S port` start-up option on the server) or *spec* is the name of the SPEC process using a port number in the default range (6510 to 6530). (As mentioned above, the current implementation requires a client to reference a particular server consistently by port number or by SPEC process name.)

The argument *property* is a string of the form "motor/tth/base_rate" or "var/DEGC", for example. The built-in properties are documented below.

Server Functions

There is only one built-in function available to the server to communicate with its clients.

`prop_send(property, value)` – Sends an event to all clients registered for *property*.

There is nothing to prevent a user-level call of `prop_send()` from generating events for built-in properties, although that may lead to an unexpected client response.

Client Functions

`prop_get(host, property)` – Reads and returns the current value of *property* from the remote *host*. Single-valued, associative-array and data-array types can be returned.

`prop_put(host, property, value)` – Sets *property* to *value* on the remote *host*. Single-valued, associative-array and data-array types can be sent.

`prop_watch(host, property)` – Registers *property* on the remote *host*. When the *property* value changes, the remote *host* will send an event to the client. Consider:

```
prop_watch("remotehost:6789", "var/TEMP")
```

If a variable named TEMP exists on the local client, then the value of the local client's instance will track changes to the value of the same variable on the remote host.

The variable must exist on the server before the client requests it be watched. If the variable goes out of existence on the server, but is subsequently recreated as the same type of global variable, the watched status will be reinstated (as of SPEC release 5.05.05-1). If the variable doesn't exist on the client or goes out of existence, the client will continue to receive events, and if the variable is recreated on the client, its value will track the values sent with the events (as of SPEC release 5.05.05-1).

Regular global variables, associative arrays and associative array elements can be watched. Data arrays cannot be watched. The built-in motor and scaler arrays `A[]` and `S[]` can be watched, but events will only be generated when the elements are explicitly assigned values on the server, not when the values change by way of built-in code, such as from `calcA`, `getangles` or `getcounts`.

`remote_stat(host)` – Returns nonzero if the connection to *host* is up, otherwise returns zero.

`remote_stat(host, "?")` – Returns the string "up" if the connection to *host* is up.

Returns the string "lost" if the connection had been up, but has been lost. Returns the string "trying" if in the process of creating a connection. Returns the string "no connection" if unable to contact the host. Returns the string "not connected" if the client has not attempted to create a connection to the host. (These options available with SPEC release 5.06.03-7.)

`remote_par(host, "connect")` – Initiates a connection to *host*. Connections are initiated automatically when SPEC server hardware is configured or when a server is accessed with the commands below. However, in order to set a timeout for `remote_eval()` prior to accessing the host, this "connect" option is available.

- `remote_par(host, "close")` – Closes the connection to the `spec` server `host`, but only if no server hardware is configured on the client.
- `remote_par(host, "abort")` – Sends an `SV_ABORT` message to the server `host`, which has the effect on the server as if a `^C` had been typed at its keyboard.
- `remote_par(host, "timeout" [, value])` – Returns or sets the timeout interval for a `remote_eval()` call to complete. The units are seconds, and the default value is four seconds.
- `remote_cmd(host, cmd)` – Puts the `spec` command `cmd` on the execution queue of the remote `host`. This function does not wait for the command to execute on the server. The return is immediate. Use `remote_eval()` or `remote_async()` with `remote_poll()` for synchronous execution of commands on the remote server.
- `remote_eval(host, cmd)` – Puts the `spec` command `cmd` on the execution queue of the remote `host` and returns the result. The function will not return until the command has been executed on the server. Single-valued (number or string), associative-array and data-array return values are allowed. If a `^C` keyboard interrupt is received before the host returns a result and before the connection timeout interval, an `SV_ABORT` message will be sent to the server, which will have the same effect on the server as a `^C` from the server's keyboard (as of `spec` release 5.08.04-3). In addition, any pending commands in the server queue from the client will be removed.
- `id = remote_async(host, cmd)` – Puts the `spec` command `cmd` on the execution queue of the remote `host` and returns a unique `id`. The return value of the command can be retrieved using the `remote_poll()` function, below. A `^C` keyboard interrupt will clear the local queue of all pending events, but will not stop the commands from being executed on the remote host. The `wait()` command can be used to wait for all asynchronous remote events. Bit 0x8 in the (optional) argument to `wait()` corresponds to remote events.
- `remote_poll(id, "status")` – Returns 1 (or 0) to indicate whether result number `id` from `remote_async()` is available (or not). Returns `-1` if `id` is no longer in queue or if the connection to the server has disappeared.
- `remote_poll(id)` – Returns the result of the command associated with the `id` value of a particular call to `remote_async()`. Single-valued result types (number or string), along with associative-array and data-array results are allowed. Note, if called before the result is available, the `id` will be removed from the queue, and the return value will be zero. Always check first if the result is available using either the "status" option above or the `wait()` function for remote events before fetching the result. Returns `-1` if `id` is not in the queue or if the connection to the server has disappeared.

Protocol

The protocol used by the `spec` server is described below. Clients other than a `spec` client that follow the protocol can also communicate with `spec` servers.

Messages are sent between the server and the clients via a TCP/IP socket using a structured packet with a fixed-size header and an optional variable-sized data section.

All definitions described below will be in the header file `SPECD/include/spec_server.h` where `SPECD` is the `spec` auxiliary file directory, usually `/usr/local/lib/spec.d`.

The default port range definitions appear in the header file as follows:

```
#define SV_PORT_FIRST 6510
#define SV_PORT_LAST 6530
```

Clients that wish to connect using `spec` process names rather than explicit port numbers

should scan through the ports in the above (inclusive) range to look for the correct server, using the “hello” protocol described below.

The packet header structure is currently 132 bytes long and defined as follows:

```
#define SV_NAME_LEN      80

struct svr_head {
    int     magic; /* SV_SPEC_MAGIC */
    int     vers; /* Protocol version number */
    unsigned size; /* Size of this structure */
    unsigned sn; /* Serial number (client's choice) */
    unsigned sec; /* Time when sent (seconds) */
    unsigned usec; /* Time when sent (microseconds) */
    int     cmd; /* Command code */
    int     type; /* Type of data */
    unsigned rows; /* Number of rows if array data */
    unsigned cols; /* Number of cols if array data */
    unsigned len; /* Bytes of data that follow */
#ifdef SV_VERSION >= 3
    int     err; /* Error code */
#endif
#ifdef SV_VERSION >= 4
    int     flags; /* Flags */
#endif
    char    name[SV_NAME_LEN]; /* name of property */
};
```

The element `magic` should be set to the following:

```
#define SV_SPEC_MAGIC    4277009102
```

The server will reject any packets lacking the magic number.

The `spec` server will check the endianness of the `magic` element of the first packet sent by the client and swap header and data bytes in that packet and subsequent incoming and outgoing data, if necessary, to accommodate the client. The client can send and read packet headers (and binary array data) using the native endian format of the client's platform.

The `vers` element should be set to the current version number:

```
#define SV_VERSION      4
```

The `size` element should be set to the size of the header structure (currently 132 bytes). Clients should check both the version number and size of the structure and be prepared to accommodate future `spec` releases where the version number and structure size may change.

(Note, in `spec` release 5.05.04, the header version number changed from 2 to 3 due to the addition of the `err` error-code element to the header structure. In `spec` release 5.05.05, the header version number changed from 3 to 4 due to the addition of the `flags` element to the header structure. However, `spec` releases from 5.05.04-5 on should work with both clients and servers using earlier or later header versions.)

The serial-number element `sn` is under the control of the client and can be used to keep track of server replies. For `SV_REPLY` packets sent in response to client commands, the server will set the serial number to that of the request. The serial number for all `SV_EVENT` packets sent by the server is zero.

The `sec` and `usec` packet elements may be useful for debugging. The server sets them based on the host's real-time clock. The client can do the same. When the `spec` debug level is set to 4096 (0x1000), the elapsed time in milliseconds between packets will be displayed. When the

elapsed time between a packet and the previous becomes greater than one second, the time difference is shown as zero.

The `cmd` element contains one of the following command codes:

```
#define SV_CLOSE          1      /* From Client */
#define SV_ABORT          2      /* From Client */
#define SV_CMD            3      /* From Client */
#define SV_CMD_WITH_RETURN 4     /* From Client */
#define SV_RETURN        5      /* Not yet used */
#define SV_REGISTER      6      /* From Client */
#define SV_UNREGISTER    7      /* From Client */
#define SV_EVENT         8      /* From Server */
#define SV_FUNC          9      /* From Client */
#define SV_FUNC_WITH_RETURN 10   /* From Client */
#define SV_CHAN_READ     11     /* From Client */
#define SV_CHAN_SEND     12     /* From Client */
#define SV_REPLY         13     /* From Server */
#define SV_HELLO        14     /* From Client */
#define SV_HELLO_REPLY  15     /* From Server */
```

`SV_CLOSE` – Can be sent by the client to terminate a connection, allowing the server to release resources. Resources will be released in any case, if the server loses the connection to the client.

`SV_ABORT` – Sent by the client to abort commands the server is currently running. Receiving an `SV_ABORT` packet is equivalent to typing `^C` at the server keyboard.

`SV_CMD` – Sent by the client to place the commands from the data string following the header onto the server command queue.

`SV_CMD_WITH_RETURN` – As above, but the output generated by the command will be returned to the client in an `SV_REPLY` packet.

`SV_RETURN` – Not currently used.

`SV_REGISTER` – Sent by the client to register a property on which to receive events.

`SV_UNREGISTER` – Sent by the client to unregister a property, so that events will no longer be sent.

`SV_EVENT` – Asynchronous packet sent by the server to clients registered for a property when the property value changes. The packet is also sent when the property is registered with `SV_REGISTER`.

`SV_FUNC` – Sent by the client to put a single function (or command) on the server command queue. The data following the header contains the function (or command) name and any arguments, separated by null bytes. The server will add parentheses, commas or space characters, as needed, depending on whether the first item is a function or a command.

`SV_FUNC_WITH_RETURN` – As above, but the result of the function will be returned to the client in an `SV_REPLY` packet.

`SV_CHAN_READ` – Sent by the client to get the value of a property.

`SV_CHAN_SEND` – Sent by the client to set the value of a property.

`SV_REPLY` – Sent by the server with the result of a command.

SV_HELLO – Sent by the client to check if the desired server is listening on a particular port. The client should then look for the expected **SV_HELLO_REPLY** response. The **spec** client puts a short message in the property name field for debugging purposes, which the server otherwise ignores.

SV_HELLO_REPLY – Sent by the server in response to an **SV_HELLO** packet. The data section of the reply contains the name of the **spec** process by which the server was invoked. That name is used by clients to find servers specified by **spec** process name.

The `type` element of the header structure describes the type of data (if any) that follow. The recognized types are:

```
#define SV_DOUBLE      1
#define SV_STRING     2
#define SV_ERROR      3
#define SV_ASSOC      4
#define SV_ARR_DOUBLE  5
#define SV_ARR_FLOAT   6
#define SV_ARR_LONG    7
#define SV_ARR_ULONG   8
#define SV_ARR_SHORT   9
#define SV_ARR_USHORT 10
#define SV_ARR_CHAR    11
#define SV_ARR_UCHAR   12
#define SV_ARR_STRING  13
```

The **spec** client does not currently use the **SV_DOUBLE** data type, and the **spec** server does not currently send any data using that type. However, the **spec** server will recognize incoming packets used to set a `var/property` with **SV_DOUBLE** data.

The **spec** server sends both string-valued and number-valued items as strings. Numbers are converted to strings using a `printf("%.15g")` format.

For the **SV_ERROR** type, which is only sent by the server, the data following the header is a string containing an error message.

The **SV_ASSOC** type is used for sending **spec**'s associative arrays. (An associative array element has an arbitrary string or number index and a string or number value.) Number-valued indices and values are converted to strings. The associative array data is sent as a series of null-terminated strings in the order index, value, index, value, etc. There is an additional null byte appended to the sequence.

When sending data to the server, any number of elements of the associative array may be included. Note, for the built-in associative arrays (`A[]`, `S[]` and possibly `G[]`, `Q[]`, `Z[]`, `U[]` and `UB[]`, depending on geometry), only existing elements may be sent to the server. For non-built-in associative arrays, the array must already exist, but new elements will be automatically created if included in the data.

The **SV_ARR_*** data types are for transferring **spec** data arrays. The data is transferred in binary format row by row using the client's native byte order.

For **spec** data arrays, the `rows` and `cols` header elements are set to the array dimensions. **spec** supports only one- and two-dimensional arrays. For a one-dimensional array, the value of one of `rows` or `cols` will be one.

For all packets containing data, the `len` header element is the number of additional bytes transmitted. If there is no data, `len` must be set to zero.

Starting with version 3 headers, a new `err` element is included in the header structure. This element is set to a nonzero value if the `SV_CMD_WITH_RETURN` or `SV_FUNC_WITH_RETURN` commands fail with an unrecoverable error (the type of error that causes an interactive `spec` session to reset to the main command prompt, such as a syntax error or a divide-by-zero error). This element is needed to distinguish between an unrecoverable error and a command or function that returns zero.

The `flags` structure element was introduced with version 4 headers. Currently, it is used only by the server to transmit the flag

```
#define SV_DELETED      0x1000
```

in packets sent to clients when watched variables or associative array elements are deleted. The `spec` client currently does not take any action on receipt of such events.

The `name` element of the header contains the null-terminated property name in ASCII, when applicable. Properties contain one, two or three parts, separated by slashes. Currently recognized property names begin with one of the following:

```
error
status/
var/
output/
scaler/
motor/
```

The `error` Property

The `error` property is used by the server to inform a client when the client tries to register an unavailable property.

```
error
```

`SV_EVENT` – Sent when an `SV_REGISTER` packet contains an unrecognizable or unacceptable property string. The data will contain an error message.

Note, a client must register for the `error` property in order to receive these events.

The `status` Properties

The `status` properties reflect changes in the server state that may affect the server's ability to execute client commands or control hardware.

```
status/ready
```

`SV_EVENT` – Sent when the server is waiting for input at the interactive prompt (data is 1) and after a return has been typed at the interactive prompt (data is 0).

The server is available to execute commands from clients when it is ready.

`SV_CHAN_READ` – Data is 1 if the server command thread is busy and unable to immediately process a new command, otherwise data is 0.

```
status/shell
```

`SV_EVENT` – Sent when the server enters a subshell (data is 1) or returns from a subshell (data is 0).

`SV_CHAN_READ` – Data is 1 if server command thread is in a subshell, otherwise data is 0.

Note, when in a subshell, the server will not process commands on the input queue.

```
status/simulate
```

`SV_EVENT` – Sent when the server enters (data is 1) or leaves (data is 0) simulate mode.

`SV_CHAN_READ` – Data is 1 if server is in simulate mode, otherwise data is 0.

Note, when in simulate mode, the server will not send commands to hardware devices.

status/quit

SV_EVENT – Sent when the server exits.

SV_CHAN_READ – Always read back as zero.

The var Properties

The var properties allow values of any variables to be transferred between the server and the client.

var/var_name

SV_EVENT – Sent to clients who have registered when the variable *var_name* changes value.

SV_CHAN_READ – Returns the value of the *var_name* in the data, if *var_name* is an existing variable on the server.

SV_CHAN_SEND – Sets the value of *var_name* on the server to the contents of data.

All data types (numbers, strings, associative arrays and data arrays) are supported. Numbers and strings are always sent by the server as null-terminated strings, although the server will accept SV_DOUBLE-type data for number-valued single variables.

For built-in associative arrays (A[], S[] and possibly G[], Q[], Z[], U[] and UB[], depending on geometry), only existing elements can be set. For associative arrays created at user-level, any number of elements can be sent and created in one call.

An associative array element can be sent to the server in two ways, either with the array name and element specified in the property name (as in "var/arr[13]") and the value contained in the data, or with the array name only specified in the property name (as in "var/arr") and the index and value specified in the data. In the first case, the associative array element must already exist on the server. In the second case, any number of elements can be sent as a series of null-terminated strings in the order index, value, index, value, etc.

Data arrays are transferred in binary format row by row using the client's native byte order. The packet header contains the number of rows and columns and the data type.

The output Property

The output property puts copies of the strings written to files or to the screen in events sent to clients.

output/filename

SV_EVENT – Sent when the server sends output to the file or device given by *filename*, where *filename* can be the built-in name "tty" or a file or device name. The data will be a string representing the output.

Once a client has registered for output events from a particular file, the server will keep track of the client's request as the file is opened and closed. File names are given relative to the server's current directory and can be relative or absolute path names, just as with the built-in commands that refer to files.

(The output property was introduced in spec release 5.07.04-1.)

The scaler Properties

The scaler properties are used to control the hardware timer and monitor counters on the server.

scaler/.all./count

SV_EVENT – Sent when counting starts (data is 1) and when counting stops (data is 0).

SV_CHAN_READ – Data indicates counting (1) or not counting (0).

SV_CHAN_SEND – If data is nonzero, the server pushes a

count_em data\n

onto the command queue. If data is 0, counting is aborted as if a ^C had been typed at the server.

scaler/*mne*/value

SV_EVENT – Sent periodically while counting on the server with current contents of scaler channel *mne*. Also sent after counting has finished.

SV_CHAN_READ – Returns the current scaler value for channel *mne*.

When configured as a timer/counter on a `spec` client, both `count` and `value` events will be received from the server. When configured as “counters only” on a `spec` client, only `value` events will be received. In fact, for “counters only” counters on a `spec` client, the client count functions do not trigger any activity on the server.

The motor Properties

The `motor` properties are used to control the motors. The parameters for the commands that are sent from the client and the values in the replies and events that sent from the server are always transmitted as ASCII strings in the data that follows the packet header.

motor/*mne*/position

SV_EVENT – Sent when the dial position or user offset changes. The data contains the motor position in user units.

SV_CHAN_READ – Returns the current motor position in user units.

SV_CHAN_SEND – Sets the user offset on the server by pushing a

```
set mne data\n
```

onto the command queue.

motor/*mne*/dial_position

SV_EVENT – Sent when the dial position changes. The data contains the motor position in dial units.

SV_CHAN_READ – Returns the current motor position in dial units.

SV_CHAN_SEND – Sets the dial position on the server by pushing a

```
set_dial mne data\n
```

onto the command queue, unless the dial position is already set to that value.

motor/*mne*/offset

SV_EVENT – Sent when the offset changes. The data contains the user offset in motor units (degrees, mm, etc.)

SV_CHAN_READ – Returns the current user offset in dial units.

SV_CHAN_SEND – Sets the user offset by pushing the

```
set mne value\n
```

command onto the command queue, unless the offset is already at the value.

The data should contain the offset value in motor units (degrees, mm, etc.).

The server will calculate *value* for the argument in `set` appropriately.

motor/*mne*/step_size

SV_EVENT – Sent when the steps-per-unit parameter changes.

SV_CHAN_READ – Returns the current steps-per-unit parameter.

The server doesn't allow clients to change this parameter.

motor/*mne*/sign

SV_EVENT – Sent when the sign-of-user×dial parameter changes.

SV_CHAN_READ – Returns the current sign-of-user×dial parameter.

The server doesn't allow clients to change this parameter.

motor/./prestart_all

SV_CHAN_SEND – Should be sent in preparation of a move of more than one motor.

Puts the string

```
{getangles;
```

into a buffer that will be pushed onto the server command queue when the move is started. Must be followed by a start_all packet, with possible intervening start_one packets.

motor/mne/start_one

SV_CHAN_SEND – If preceded by a prestart_all, adds a

```
A[mne]=data;
```

to the buffer that will be pushed onto the server command queue. Otherwise, pushes

```
{get_angles;A[mne]=data;move_em;}\n
```

onto the command queue in order to start the single motor moving.

motor/./start_all

SV_CHAN_SEND – When preceded by prestart_all and one or more start_one packets, adds a

```
move_em;}\n
```

to the buffer created by those commands and pushes the entire buffer onto the command queue. Otherwise, does nothing.

motor/./abort_all

SV_CHAN_SEND – Causes a ^C-type interrupt in the server command thread.

motor/mne/move_done

SV_EVENT – Sent when moving starts (data is 1) and when moving stops (data is 0).

SV_CHAN_READ – Returns one if the motor is busy, otherwise zero.

motor/mne/high_lim_hit

SV_EVENT – Sent when the high-limit switch has been hit.

SV_CHAN_READ – Returns nonzero if the high-limit switch has been hit.

motor/mne/low_lim_hit

SV_EVENT – Sent when the low-limit switch has been hit.

SV_CHAN_READ – Returns nonzero if the low-limit switch has been hit.

motor/mne/emergency_stop

SV_EVENT – Sent when a motor controller indicates a hardware emergency stop.

SV_CHAN_READ – Returns nonzero if an emergency-stop switch or condition has been activated.

motor/mne/motor_fault

SV_EVENT – Sent when a motor controller indicates a hardware motor fault.

SV_CHAN_READ – Returns nonzero if a motor-fault condition has been activated.

motor/mne/high_limit

SV_EVENT – Sent when the value of the high limit position changes. The data contains the high limit in dial units.

SV_CHAN_READ – Returns the high limit in dial units.

SV_CHAN_SEND – Sets the high limit by pushing

```
set_lm mne data user(mne,get_lim(mne,-1))\n
```

onto the server command queue.

motor/mne/low_limit

SV_EVENT – Sent when the value of the low limit position changes. The data contains the low limit in dial units.

SV_CHAN_READ – Returns the low limit in dial units.

SV_CHAN_SEND – Sets the low limit by pushing

```
set_lm mne data user(mne,get_lim(mne,+1))\n
```

onto the server command queue.

motor/mne/limits

SV_CHAN_SEND – Sets both motor limits by pushing

```
set_lm mne data\n
```

onto the server command queue, where *data* should contain the low and high motor limit values in a string.

motor/mne/search

SV_CHAN_SEND – The server starts a home or limit search by pushing a

```
chg_dial(mne, how)\n
```

or a

```
chg_dial(mne, how, home_pos)\n
```

onto the command queue, depending on whether the data contains one or two arguments. The *how* argument is one of the strings recognized by `chg_dial()`, namely "home", "home+", "home-", "lim+" or "lim-". The optional *home_pos* is the home position in dial units.

motor/mne/sync_check

SV_EVENT – Sent when a position discrepancy occurs between `spec` and the motor controller that requires human intervention to resolve. The data contains two values: the position that `spec` has in memory and the position read from the controller, both in dial units. A response is required, either from the server keyboard or by a client sending the property described below. Note, a motor's `read_mode` can be configured to always accept the controller position so that discrepancy events will never be generated.

SV_CHAN_SEND – A client should send a yes or no (1 or 0) response when the server needs to resolve a motor discrepancy. The server requires this response or a keyboard response on the server's controlling terminal.

Note, until a response is received by the server, either from a client or the server's interactive prompt, the server will not be able to process commands in the command queue.

motor/mne/unusable

SV_EVENT – Sent when a "disable" option to `motor_par()` has changed the enabled/disabled state of a motor on the server. (This event implemented in `spec` release 5.07.01-5.)

SV_CHAN_READ – Returns nonzero if the motor is unusable. The motor may be unusable because it didn't respond to the presence test, it has been explicitly disabled with the "disable" option to `motor_par()`, or it has received an unusable event from another server.

The following properties correspond to standard and optional motor parameters. All behave the same with respect to server/client communication.

Standard Parameters:

```
motor/mne/base_rate
motor/mne/slew_rate
motor/mne/acceleration
motor/mne/backlash
```

Optional Parameters:

```
motor/mne/home_base_rate
motor/mne/home_slew_rate
motor/mne/home_acceleration
motor/mne/encoder_step_size
motor/mne/dc_dead_band
motor/mne/dc_settle_time
motor/mne/dc_proportional_gain
motor/mne/dc_derivative_gain
motor/mne/dc_integral_gain
motor/mne/dc_integration_limit
motor/mne/dc_following_error
motor/mne/dc_sampling_interval
motor/mne/dc_veloc_feedforward
motor/mne/dc_accel_feedforward
motor/mne/step_mode
motor/mne/disable_limit_checks
motor/mne/slop
motor/mne/read_mode
motor/mne/deceleration
motor/mne/torque
motor/mne/misc_par_1
motor/mne/misc_par_2
motor/mne/misc_par_3
motor/mne/misc_par_4
motor/mne/misc_par_5
motor/mne/misc_par_6
motor/mne/powder_base
motor/mne/powder_slew
motor/mne/powder_acceleration
```

SV_EVENT – Sent when the parameter changes. The data contains the value of the parameter.

SV_CHAN_READ – Returns the value of the parameter.

SV_CHAN_SEND – Sets the parameter by pushing

```
motor_par(mne, cmd, data)\n
```

onto the server command queue with the appropriate arguments, using the value passed in the data portion of the packet.

The following property names correspond to features in SPEC's internal motor control code, but aren't currently used with the server.

```
motor/mne/magnitude
motor/mne/backlash_rate
motor/./preread_all
motor/mne/preread_one
motor/mne/abort_one
motor/./flush_all
motor/mne/flush_one
motor/mne/set_position
motor/mne/diff_position
motor/mne/prestart_one
```

Updates

The following summarizes updates to the spec server/client protocol that mainly affect compatibility with older or newer servers or clients. Search for “server” in the *changes* help file for a complete list of updates.

Release 5.04.03 – July 20, 2003 – Initial server/client support released with SV_VERSION at 2.

Release 5.05.04 – July 18, 2004 – Added `err` element to `svr_head` structure. SV_VERSION at 3, but maintained server compatibility with version 2 clients.

Release 5.05.04-4 – August 17, 2004 – Fixed server so that it can work with future clients with higher SV_VERSION numbers.

Release 5.05.05 – Sept 30, 2004 – Added `flags` element to `svr_head` structure. SV_VERSION at 4. spec clients and servers should be both forward and backward compatible with respect to protocol version.