

NAME

sockets – Connect to sockets

DESCRIPTION

spec user-level socket functions allow you to connect and communicate with sockets created by other processes on the local platform or on a remote host. The user-level access is through the functions `sock_get()`, `sock_put()` and `sock_par()`. (These functions were introduced in spec release 5.01.01, replacing the now deprecated `sock_io()` function.)

BUILT-IN FUNCTIONS

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 creating 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 timeout expires. The maximum number of characters that can be read at a time in 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". (For spec versions prior to release 5.01.01, use `int2` for short and `int4` for long.)

`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("?")` – Lists the available commands.

`sock_par("show")` – Lists the current open sockets and their status.

`sock_par("info")` – Returns a string that lists the current open sockets and their status.

`sock_par("host:port", "connect")` – Opens a socket to the specified host and port. Returns **true** for success and **false** for failure.

`sock_par("host:port", "connect_udp")` – Opens a socket to the specified host and port using the UDP protocol. Returns **true** for success and **false** for failure.

`sock_par("host:port", "listen")` – Sets up a socket for listening, allowing another instance of `spec` or some other program to make a connection.

`sock_par("host:port", "close")` – Closes the socket associated with the specified host and port.

`sock_par("host: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.

`sock_par("host:port", "ignore_sim" [, 1|0])` – Turns *ignore-simulate* mode on or off, or 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.

`sock_par("host:port", "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.

`sock_par("host:port", "timeout" [, t])` – Returns or sets the read timeout for the socket described by `"host:port"`. The units 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.

`sock_par("host:port", "nodelay" [, 1|0])` – Returns or sets the state of the `TCP_NODELAY` socket option. 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.

```
SPEC.1> sock_put("localhost:7", "This is a test.0)
```

```
SPEC.2> print sock_get("localhost:7")  
This is a test.
```

SEE ALSO

gpib serial