

**NAME**

serial – RS-232C interface functions

**DESCRIPTION**

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, as in the example below.

SERIAL	DEVICE	<>TYPE	<>BAUD	<>MODE
0 YES	/dev/ttyS0	<>	19200	raw
1 YES	/dev/ttyS1	<>	9600	cooked igncr

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. Only three devices are displayed at a time. Use the `^F` and `^B` commands to display additional rows.

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 on either the **Devices** screen or the **Acquisition** screen of the configuration editor.

The default serial interface is through the built-in standard UNIX serial driver. The **TYPE** menu allows optional selection of additional serial interfaces, namely **EPICS**, **TACO** or **SOCKET**. (TACO was previously labeled **ESRF**.) For **EPICS**, the **DEVICE** field should contain the base name of the serial record process variables. For **TACO**, the **DEVICE** field should contain the name of the associated device server. The **SOCKET** serial interface connects to Ethernet-to-serial devices using the IP address (or resolvable host name) and port number entered in the **DEVICE** field, such as `192.168.1.100:7890`.

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.

The behavior of the serial interface depends on a number of configurable parameters, set by the **MODE** menu, described next. The seldom-used data-bits and stop-bits parameters can only be adjusted at run time using the `ser_par()` function, described further below.

**TTY MODES**

The serial device **tty** modes refer to the kind of character processing the UNIX kernel inserts between the serial device and the `spec` program. (Depending on the platform, **tty** mode descriptions might be found in the *termios(3)*, *termio(7)* or *tty(4)* sections of the UNIX man pages.) `spec` sets the **tty** modes for a particular serial device according to the selection in `spec`'s hardware *config* file.

There are many more **tty**-mode parameters recognized by the serial device drivers than `spec` offers as configuration options. 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

eightth 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.

## BUILT-IN FUNCTIONS

`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 from the serial device with address `addr` 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". (For `spec` versions prior to release 5.01.01, use `int2` for short and `int4` for long.)

`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, "device_id")` – Returns the name of the associated serial device or `-1` if there is no serial device configured as `addr` (as of `spec` release 5.05.05-7).

`ser_par(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` (as of `spec` release 5.05.05-7).

`ser_par(addr, "drain")` – Waits for pending output on the associated serial device to be transmitted (as of `spec` release 5.09.01-1), but can be interrupted with `^C`. Use the "flush" option, described next, to empty the output queue. Does nothing and returns immediately for `SOCKET`, `EPICS` and `TACO` serial interfaces.

`ser_par(addr, "flush" [, how])` – Flushes the input and/or output queues for the serial device with address `addr`. If `how` is zero or absent, the input queue is flushed. If `how` is one, the output queue is flushed. 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.

`ser_par(addr, "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.

`ser_par(addr, "timeout" [, t])` – Returns or sets the read timeout for the serial device with address `addr`. The units 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.

`ser_par(addr, "baud" [, value])` – Returns or sets the baud rate for the serial device with address `addr` (as of `spec` release 5.08.6-4). Valid baud rates are from 300 to 4000000. The function returns the device's baud rate. If `value` 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.

`ser_par(addr, "stop_bits" [, bits])` – Returns or sets the stop-bits value for the serial device with address `addr` (as of `spec` release 5.06.05-7). 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.

`ser_par(addr, "data_bits" [, bits])` – Returns or sets the data-bits value for the serial device with address `addr` (as of `spec` release 5.07.02-2). 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 (on start up and on `reconfig`). This mode is not supported on `SOCKET` interfaces.

`ser_par(addr, "dtr" [, 1|0])` Returns the current setting or sets or clears the Data Terminal Ready (DTR) control line (as of `spec` release 5.08.03-4). Only available on standard serial interfaces. Reset on hardware reconfiguration.

`ser_par(addr, "rts" [, 1|0])` Returns the current setting or sets or clears the Request To Send (RTS) control line (as of `spec` release 5.08.03-4). Only available on standard serial interfaces. Reset on hardware reconfiguration.

`ser_par(addr, "dsr")` – Returns the current setting of the Data Set Ready (DSR) control line (as of spec release 5.08.03-4). 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 of spec release 5.08.03-4), 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.

**SEE ALSO**

gpib sockets config\_adm