NAME

zshtcpsys − zsh tcp system

DESCRIPTION

A module **zsh/net/tcp** is provided to provide network I/O over TCP/IP from within the shell; see its description in *zshmodules*(1). This manual page describes a function suite based on the module. If the module is installed, the functions are usually installed at the same time, in which case they will be available for autoloading in the default function search path. In addition to the **zsh/net/tcp** module, the **zsh/zselect** module is used to implement timeouts on read operations. For troubleshooting tips, consult the corresponding advice for the **zftp** functions described in *zshzftpsys*(1).

There are functions corresponding to the basic I/O operations open, close, read and send, named **tcp_open** etc., as well as a function **tcp_expect** for pattern match analysis of data read as input. The system makes it easy to receive data from and send data to multiple named sessions at once. In addition, it can be linked with the shell's line editor in such a way that input data is automatically shown at the terminal. Other facilities available including logging, filtering and configurable output prompts.

To use the system where it is available, it should be enough to '**autoload −U tcp_open**' and run **tcp_open** as documented below to start a session. The **tcp_open** function will autoload the remaining functions.

TCP USER FUNCTIONS

Basic I/O

tcp_open [**−qz**] *host port* [*sess*]
tcp_open [**−qz**] [**−s** *sess* | **−l** *sess*[**,**...]] ...
tcp_open [**−qz**] [**−a** *fd* | **−f** *fd*] [*sess*]

Open a new session. In the first and simplest form, open a TCP connection to host *host* at port *port*; numeric and symbolic forms are understood for both.

If *sess* is given, this becomes the name of the session which can be used to refer to multiple different TCP connections. If *sess* is not given, the function will invent a numeric name value (note this is *not* the same as the file descriptor to which the session is attached). It is recommended that session names not include 'funny' characters, where funny characters are not well−defined but certainly do not include alphanumerics or underscores, and certainly do include whitespace.

In the second case, one or more sessions to be opened are given by name. A single session name is given after **−s** and a comma−separated list after **−l**; both options may be repeated as many times as necessary. A failure to open any session causes **tcp_open** to abort. The host and port are read from the file **.ztcp_sessions** in the same directory as the user's zsh initialisation files, i.e. usually the home directory, but **$ZDOTDIR** if that is set. The file consists of lines each giving a session name and the corresponding host and port, in that order (note the session name comes first, not last), separated by whitespace.

The third form allows passive and fake TCP connections. If the option **−a** is used, its argument is a file descriptor open for listening for connections. No function front−end is provided to open such a file descriptor, but a call to '**ztcp −l** *port*' will create one with the file descriptor stored in the parameter **$REPLY**. The listening port can be closed with '**ztcp −c** *fd*'. A call to '**tcp_open −a** *fd*' will block until a remote TCP connection is made to *port* on the local machine. At this point, a session is created in the usual way and is largely indistinguishable from an active connection created with one of the first two forms.

If the option **−f** is used, its argument is a file descriptor which is used directly as if it were a TCP session. How well the remainder of the TCP function system copes with this depends on what actually underlies this file descriptor. A regular file is likely to be unusable; a FIFO (pipe) of some sort will work better, but note that it is not a good idea for two different sessions to attempt to read from the same FIFO at once.

If the option **−q** is given with any of the three forms, **tcp_open** will not print informational messages, although it will in any case exit with an appropriate status.

If the line editor (zle) is in use, which is typically the case if the shell is interactive, **tcp_open**

installs a handler inside zle which will check for new data at the same time as it checks for keyboard input. This is convenient as the shell consumes no CPU time while waiting; the test is performed by the operating system. Giving the option **−z** to any of the forms of **tcp_open** prevents the handler from being installed, so data must be read explicitly. Note, however, this is not necessary for executing complete sets of send and read commands from a function, as zle is not active at this point. Generally speaking, the handler is only active when the shell is waiting for input at a command prompt or in the **vared** builtin. The option has no effect if zle is not active; '**[[−o zle]]**' will test for this.

The first session to be opened becomes the current session and subsequent calls to **tcp_open** do not change it. The current session is stored in the parameter **$TCP_SESS**; see below for more detail about the parameters used by the system.

The function **tcp_on_open**, if defined, is called when a session is opened. See the description below.

tcp_close [**−qn**] [**−a** | **−l** *sess*[**,**...] | *sess* ...]

Close the named sessions, or the current session if none is given, or all open sessions if **−a** is given. The options **−l** and **−s** are both handled for consistency with **tcp_open**, although the latter is redundant.

If the session being closed is the current one, **$TCP_SESS** is unset, leaving no current session, even if there are other sessions still open.

If the session was opened with **tcp_open −f**, the file descriptor is closed so long as it is in the range 0 to 9 accessible directly from the command line. If the option **−n** is given, no attempt will be made to close file descriptors in this case. The **−n** option is not used for genuine **ztcp** session; the file descriptors are always closed with the session.

If the option **−q** is given, no informational messages will be printed.

tcp_read [**−bdq**] [**−t** *TO*] [**−T** *TO*]
[**−a** | **−u** *fd*[**,**...] | **−l** *sess*[**,**...] | **−s** *sess* ...]

Perform a read operation on the current session, or on a list of sessions if any are given with **−u**, **−l** or **−s**, or all open sessions if the option **−a** is given. Any of the **−u**, **−l** or **−s** options may be repeated or mixed together. The **−u** option specifies a file descriptor directly (only those managed by this system are useful), the other two specify sessions as described for **tcp_open** above.

The function checks for new data available on all the sessions listed. Unless the **−b** option is given, it will not block waiting for new data. Any one line of data from any of the available sessions will be read, stored in the parameter **$TCP_LINE**, and displayed to standard output unless **$TCP_SILENT** contains a non−empty string. When printed to standard output the string **$TCP_PROMPT** will be shown at the start of the line; the default form for this includes the name of the session being read. See below for more information on these parameters. In this mode, **tcp_read** can be called repeatedly until it returns status 2 which indicates all pending input from all specified sessions has been handled.

With the option **−b**, equivalent to an infinite timeout, the function will block until a line is available to read from one of the specified sessions. However, only a single line is returned.

The option **−d** indicates that all pending input should be drained. In this case **tcp_read** may process multiple lines in the manner given above; only the last is stored in **$TCP_LINE**, but the complete set is stored in the array **$tcp_lines**. This is cleared at the start of each call to **tcp_read**.

The options **−t** and **−T** specify a timeout in seconds, which may be a floating point number for increased accuracy. With **−t** the timeout is applied before each line read. With **−T**, the timeout applies to the overall operation, possibly including multiple read operations if the option **−d** is present; without this option, there is no distinction between **−t** and **−T**.

The function does not print informational messages, but if the option **−q** is given, no error message is printed for a non−existent session.

A return status of 2 indicates a timeout or no data to read. Any other non−zero return status indicates some error condition.

See **tcp_log** for how to control where data is sent by **tcp_read**.

tcp_send [**−cnq**] [**−s** *sess* | **−l** *sess*[**,**...]] *data ...*
tcp_send [**−cnq**] **−a** *data ...*

Send the supplied data strings to all the specified sessions in turn. The underlying operation differs little from a '**print −r**' to the session's file descriptor, although it attempts to prevent the shell from dying owing to a **SIGPIPE** caused by an attempt to write to a defunct session.

The option **−c** causes **tcp_send** to behave like **cat**. It reads lines from standard input until end of input and sends them in turn to the specified session(s) exactly as if they were given as *data* arguments to individual **tcp_send** commands.

The option **−n** prevents **tcp_send** from putting a newline at the end of the data strings.

The remaining options all behave as for **tcp_read**.

The data arguments are not further processed once they have been passed to **tcp_send**; they are simply passed down to **print −r**.

If the parameter **$TCP_OUTPUT** is a non−empty string and logging is enabled then the data sent to each session will be echoed to the log file(s) with **$TCP_OUTPUT** in front where appropriate, much in the manner of **$TCP_PROMPT**.

Session Management

tcp_alias [**−q**] *alias***=***sess ...*
tcp_alias [**−q**] [*alias ...*]
tcp_alias −d [**−q**] *alias ...*

This function is not particularly well tested.

The first form creates an alias for a session name; *alias* can then be used to refer to the existing session *sess*. As many aliases may be listed as required.

The second form lists any aliases specified, or all aliases if none.

The third form deletes all the aliases listed. The underlying sessions are not affected.

The option **−q** suppresses an inconsistently chosen subset of error messages.

tcp_log [**−asc**] [**−n** | **−N**] [*logfile*]

With an argument *logfile*, all future input from **tcp_read** will be logged to the named file. Unless **−a** (append) is given, this file will first be truncated or created empty. With no arguments, show the current status of logging.

With the option **−s**, per−session logging is enabled. Input from **tcp_read** is output to the file *logfile***.***sess*. As the session is automatically discriminated by the filename, the contents are raw (no **$TCP_PROMPT**). The option **−a** applies as above. Per−session logging and logging of all data in one file are not mutually exclusive.

The option **−c** closes all logging, both complete and per−session logs.

The options **−n** and **−N** respectively turn off or restore output of data read by **tcp_read** to standard output; hence '**tcp_log −cn**' turns off all output by **tcp_read**.

The function is purely a convenient front end to setting the parameters **$TCP_LOG**, **$TCP_LOG_SESS**, **$TCP_SILENT**, which are described below.

tcp_rename *old new*

Rename session *old* to session *new*. The old name becomes invalid.

tcp_sess [*sess* [*command* [*arg ...*]]]

With no arguments, list all the open sessions and associated file descriptors. The current session is marked with a star. For use in functions, direct access to the parameters **$tcp_by_name**, **$tcp_by_fd** and **$TCP_SESS** is probably more convenient; see below.

With a *sess* argument, set the current session to *sess*. This is equivalent to changing **$TCP_SESS** directly.

With additional arguments, temporarily set the current session while executing '*command arg ...*'. *command* is re−evaluated so as to expand aliases etc., but the remaining *arg*s are passed through as that appear to **tcp_sess**. The original session is restored when **tcp_sess** exits.

Advanced I/O

tcp_command *send−option ... send−argument ...*

This is a convenient front−end to **tcp_send**. All arguments are passed to **tcp_send**, then the function pauses waiting for data. While data is arriving at least every **$TCP_TIMEOUT** (default 0.3) seconds, data is handled and printed out according to the current settings. Status 0 is always returned.

This is generally only useful for interactive use, to prevent the display becoming fragmented by output returned from the connection. Within a programme or function it is generally better to handle reading data by a more explicit method.

tcp_expect [**−q**] [**−p** *var* | **−P** *var*] [**−t** *TO* | **−T** *TO*]
[**−a** | **−s** *sess* | **−l** *sess*[**,**...]] *pattern ...*

Wait for input matching any of the given *pattern*s from any of the specified sessions. Input is ignored until an input line matches one of the given patterns; at this point status zero is returned, the matching line is stored in **$TCP_LINE**, and the full set of lines read during the call to **tcp_expect** is stored in the array **$tcp_expect_lines**.

Sessions are specified in the same way as **tcp_read**: the default is to use the current session, otherwise the sessions specified by **−a**, **−s**, or **−l** are used.

Each *pattern* is a standard zsh extended−globbing pattern; note that it needs to be quoted to avoid it being expanded immediately by filename generation. It must match the full line, so to match a substring there must be a '*****' at the start and end. The line matched against includes the **$TCP_PROMPT** added by **tcp_read**. It is possible to include the globbing flags '**#b**' or '**#m**' in the patterns to make backreferences available in the parameters **$MATCH**, **$match**, etc., as described in the base zsh documentation on pattern matching.

Unlike **tcp_read**, the default behaviour of **tcp_expect** is to block indefinitely until the required input is found. This can be modified by specifying a timeout with **−t** or **−T**; these function as in **tcp_read**, specifying a per−read or overall timeout, respectively, in seconds, as an integer or floating−point number. As **tcp_read**, the function returns status 2 if a timeout occurs.

The function returns as soon as any one of the patterns given match. If the caller needs to know which of the patterns matched, the option **−p** *var* can be used; on return, **$var** is set to the number of the pattern using ordinary zsh indexing, i.e. the first is 1, and so on. Note the absence of a '**$**' in front of *var*. To avoid clashes, the parameter cannot begin with '**_expect**'. The index −1 is used if there is a timeout and 0 if there is no match.

The option **−P** *var* works similarly to **−p**, but instead of numerical indexes the regular arguments must begin with a prefix followed by a colon: that prefix is then used as a tag to which *var* is set when the argument matches. The tag **timeout** is used if there is a timeout and the empty string if there is no match. Note it is acceptable for different arguments to start with the same prefix if the matches do not need to be distinguished.

The option **−q** is passed directly down to **tcp_read**.

As all input is done via **tcp_read**, all the usual rules about output of lines read apply. One exception is that the parameter **$tcp_lines** will only reflect the line actually matched by **tcp_expect**; use **$tcp_expect_lines** for the full set of lines read during the function call.

tcp_proxy

This is a simple−minded function to accept a TCP connection and execute a command with I/O redirected to the connection. Extreme caution should be taken as there is no security whatsoever

and this can leave your computer open to the world. Ideally, it should only be used behind a firewall.

The first argument is a TCP port on which the function will listen.

The remaining arguments give a command and its arguments to execute with standard input, standard output and standard error redirected to the file descriptor on which the TCP session has been accepted. If no command is given, a new zsh is started. This gives everyone on your network direct access to your account, which in many cases will be a bad thing.

The command is run in the background, so **tcp_proxy** can then accept new connections. It continues to accept new connections until interrupted.

tcp_spam [**−ertv**] [**−a** | **−s** *sess* | **−l** *sess*[**,**...]] *cmd* [*arg ...*]
Execute '*cmd* [*arg ...*]' for each session in turn. Note this executes the command and arguments; it does not send the command line as data unless the **−t** (transmit) option is given.

The sessions may be selected explicitly with the standard **−a**, **−s** or **−l** options, or may be chosen implicitly. If none of the three options is given the rules are: first, if the array **$tcp_spam_list** is set, this is taken as the list of sessions, otherwise all sessions are taken. Second, any sessions given in the array **$tcp_no_spam_list** are removed from the list of sessions.

Normally, any sessions added by the '**−a**' flag or when all sessions are chosen implicitly are spammed in alphabetic order; sessions given by the **$tcp_spam_list** array or on the command line are spammed in the order given. The **−r** flag reverses the order however it was arrived it.

The **−v** flag specifies that a **$TCP_PROMPT** will be output before each session. This is output after any modification to **TCP_SESS** by the user−defined **tcp_on_spam** function described below. (Obviously that function is able to generate its own output.)

If the option **−e** is present, the line given as '*cmd* [*arg ...*]' is executed using **eval**, otherwise it is executed without any further processing.

tcp_talk
This is a fairly simple−minded attempt to force input to the line editor to go straight to the default **TCP_SESS**.

An escape string, **$TCP_TALK_ESCAPE**, default '**:**', is used to allow access to normal shell operation. If it is on its own at the start of the line, or followed only by whitespace, the line editor returns to normal operation. Otherwise, the string and any following whitespace are skipped and the remainder of the line executed as shell input without any change of the line editor's operating mode.

The current implementation is somewhat deficient in terms of use of the command history. For this reason, many users will prefer to use some form of alternative approach for sending data easily to the current session. One simple approach is to alias some special character (such as '**%**') to '**tcp_command −−**'.

tcp_wait
The sole argument is an integer or floating point number which gives the seconds to delay. The shell will do nothing for that period except wait for input on all TCP sessions by calling **tcp_read −a**. This is similar to the interactive behaviour at the command prompt when zle handlers are installed.

'One−shot' file transfer

tcp_point *port*

tcp_shoot *host port*
This pair of functions provide a simple way to transfer a file between two hosts within the shell. Note, however, that bulk data transfer is currently done using **cat**. **tcp_point** reads any data arriving at *port* and sends it to standard output; **tcp_shoot** connects to *port* on *host* and sends its standard input. Any unused *port* may be used; the standard mechanism for picking a port is to think of a random four−digit number above 1024 until one works.

To transfer a file from host **woodcock** to host **springes**, on **springes**:

tcp_point 8091 >output_file

and on **woodcock**:

tcp_shoot springes 8091 <input_file

As these two functions do not require **tcp_open** to set up a TCP connection first, they may need to be autoloaded separately.

TCP USER−DEFINED FUNCTIONS

Certain functions, if defined by the user, will be called by the function system in certain contexts. This facility depends on the module **zsh/parameter**, which is usually available in interactive shells as the completion system depends on it. None of the functions need be defined; they simply provide convenient hooks when necessary.

Typically, these are called after the requested action has been taken, so that the various parameters will reflect the new state.

tcp_on_alias *alias fd*
: When an alias is defined, this function will be called with two arguments: the name of the alias, and the file descriptor of the corresponding session.

tcp_on_awol *sess fd*
: If the function **tcp_fd_handler** is handling input from the line editor and detects that the file descriptor is no longer reusable, by default it removes it from the list of file descriptors handled by this method and prints a message. If the function **tcp_on_awol** is defined it is called immediately before this point. It may return status 100, which indicates that the normal handling should still be performed; any other return status indicates that no further action should be taken and the **tcp_fd_handler** should return immediately with the given status. Typically the action of **tcp_on_awol** will be to close the session.

  The variable **TCP_INVALIDATE_ZLE** will be a non−empty string if it is necessary to invalidate the line editor display using '**zle −I**' before printing output from the function.

  ('AWOL' is military jargon for 'absent without leave' or some variation. It has no pre−existing technical meaning known to the author.)

tcp_on_close *sess fd*
: This is called with the name of a session being closed and the file descriptor which corresponded to that session. Both will be invalid by the time the function is called.

tcp_on_open *sess fd*
: This is called after a new session has been defined with the session name and file descriptor as arguments. If it returns a non−zero status, opening the session is assumed to fail and the session is closed again; however, **tcp_open** will continue to attempt to open any remaining sessions given on the command line.

tcp_on_rename *oldsess fd newsess*
: This is called after a session has been renamed with the three arguments old session name, file descriptor, new session name.

tcp_on_spam *sess command ...*
: This is called once for each session spammed, just *before* a command is executed for a session by **tcp_spam**. The arguments are the session name followed by the command list to be executed. If **tcp_spam** was called with the option **−t**, the first command will be **tcp_send**.

  This function is called after **$TCP_SESS** is set to reflect the session to be spammed, but before any use of it is made. Hence it is possible to alter the value of **$TCP_SESS** within this function. For example, the session arguments to **tcp_spam** could include extra information to be stripped off and processed in **tcp_on_spam**.

  If the function sets the parameter **$REPLY** to '**done**', the command line is not executed; in

addition, no prompt is printed for the **−v** option to **tcp_spam**.

tcp_on_unalias *alias fd*
: This is called with the name of an alias and the corresponding session's file descriptor after an alias has been deleted.

TCP UTILITY FUNCTIONS

The following functions are used by the TCP function system but will rarely if ever need to be called directly.

tcp_fd_handler
: This is the function installed by **tcp_open** for handling input from within the line editor, if that is required. It is in the format documented for the builtin '**zle −F**' in *zshzle*(1) .

  While active, the function sets the parameter **TCP_HANDLER_ACTIVE** to 1. This allows shell code called internally (for example, by setting **tcp_on_read**) to tell if is being called when the shell is otherwise idle at the editor prompt.

tcp_output [**−q**] **−P** *prompt* **−F** *fd* **−S** *sess*
: This function is used for both logging and handling output to standard output, from within **tcp_read** and (if **$TCP_OUTPUT** is set) **tcp_send**.

  The *prompt* to use is specified by **−P**; the default is the empty string. It can contain:

  %c Expands to 1 if the session is the current session, otherwise 0. Used with ternary expressions such as '**%(c.−.+)**' to output '**+**' for the current session and '**−**' otherwise.

  %f Replaced by the session's file descriptor.

  %s Replaced by the session name.

  %% Replaced by a single '**%**'.

  The option **−q** suppresses output to standard output, but not to any log files which are configured.

  The **−S** and **−F** options are used to pass in the session name and file descriptor for possible replacement in the prompt.

TCP USER PARAMETERS

Parameters follow the usual convention that uppercase is used for scalars and integers, while lowercase is used for normal and associative array. It is always safe for user code to read these parameters. Some parameters may also be set; these are noted explicitly. Others are included in this group as they are set by the function system for the user's benefit, i.e. setting them is typically not useful but is benign.

It is often also useful to make settable parameters local to a function. For example, '**local TCP_SILENT=1**' specifies that data read during the function call will not be printed to standard output, regardless of the setting outside the function. Likewise, '**local TCP_SESS=***sess*' sets a session for the duration of a function, and '**local TCP_PROMPT=**' specifies that no prompt is used for input during the function.

tcp_expect_lines
: Array. The set of lines read during the last call to **tcp_expect**, including the last (**$TCP_LINE**).

tcp_filter
: Array. May be set directly. A set of extended globbing patterns which, if matched in **tcp_output**, will cause the line not to be printed to standard output. The patterns should be defined as described for the arguments to **tcp_expect**. Output of line to log files is not affected.

TCP_HANDLER_ACTIVE
: Scalar. Set to 1 within **tcp_fd_handler** to indicate to functions called recursively that they have been called during an editor session. Otherwise unset.

TCP_LINE
: The last line read by **tcp_read**, and hence also **tcp_expect**.

TCP_LINE_FD
: The file descriptor from which **$TCP_LINE** was read. **${tcp_by_fd[$TCP_LINE_FD]}** will give the corresponding session name.

tcp_lines
: Array. The set of lines read during the last call to **tcp_read**, including the last (**$TCP_LINE**).

TCP_LOG
: May be set directly, although it is also controlled by **tcp_log**. The name of a file to which output from all sessions will be sent. The output is proceeded by the usual **$TCP_PROMPT**. If it is not an absolute path name, it will follow the user's current directory.

TCP_LOG_SESS
: May be set directly, although it is also controlled by **tcp_log**. The prefix for a set of files to which output from each session separately will be sent; the full filename is **${TCP_LOG_SESS}.***sess*. Output to each file is raw; no prompt is added. If it is not an absolute path name, it will follow the user's current directory.

tcp_no_spam_list
: Array. May be set directly. See **tcp_spam** for how this is used.

TCP_OUTPUT
: May be set directly. If a non−empty string, any data sent to a session by **tcp_send** will be logged. This parameter gives the prompt to be used in a file specified by **$TCP_LOG** but not in a file generated from **$TCP_LOG_SESS**. The prompt string has the same format as **TCP_PROMPT** and the same rules for its use apply.

TCP_PROMPT
: May be set directly. Used as the prefix for data read by **tcp_read** which is printed to standard output or to the log file given by **$TCP_LOG**, if any. Any '**%s**', '**%f**' or '**%%**' occurring in the string will be replaced by the name of the session, the session's underlying file descriptor, or a single '**%**', respectively. The expression '**%c**' expands to 1 if the session being read is the current session, else 0; this is most useful in ternary expressions such as '**%(c.−.+)**' which outputs '**+**' if the session is the current one, else '**−**'.

  If the prompt starts with **%P**, this is stripped and the complete result of the previous stage is passed through standard prompt **%**−style formatting before being output.

TCP_READ_DEBUG
: May be set directly. If this has non−zero length, **tcp_read** will give some limited diagnostics about data being read.

TCP_SECONDS_START
: This value is created and initialised to zero by tcp_open.

  The functions **tcp_read** and **tcp_expect** use the shell's **SECONDS** parameter for their own timing purposes. If that parameter is not of floating point type on entry to one of the functions, it will create a local parameter **SECONDS** which is floating point and set the parameter **TCP_SECONDS_START** to the previous value of **$SECONDS**. If the parameter is already floating point, it is used without a local copy being created and **TCP_SECONDS_START** is not set. As the global value is zero, the shell elapsed time is guaranteed to be the sum of **$SECONDS** and **$TCP_SECONDS_START**.

  This can be avoided by setting **SECONDS** globally to a floating point value using '**typeset −F SECONDS**'; then the TCP functions will never make a local copy and never set **TCP_SECONDS_START** to a non−zero value.

TCP_SESS
: May be set directly. The current session; must refer to one of the sessions established by **tcp_open**.

TCP_SILENT
May be set directly, although it is also controlled by **tcp_log**. If of non−zero length, data read by **tcp_read** will not be written to standard output, though may still be written to a log file.

tcp_spam_list
Array. May be set directly. See the description of the function **tcp_spam** for how this is used.

TCP_TALK_ESCAPE
May be set directly. See the description of the function **tcp_talk** for how this is used.

TCP_TIMEOUT
May be set directly. Currently this is only used by the function **tcp_command**, see above.

TCP USER−DEFINED PARAMETERS

The following parameters are not set by the function system, but have a special effect if set by the user.

tcp_on_read
This should be an associative array; if it is not, the behaviour is undefined. Each key is the name of a shell function or other command, and the corresponding value is a shell pattern (using **EXTENDED_GLOB**). Every line read from a TCP session directly or indirectly using **tcp_read** (which includes lines read by **tcp_expect**) is compared against the pattern. If the line matches, the command given in the key is called with two arguments: the name of the session from which the line was read, and the line itself.

If any function called to handle a line returns a non−zero status, the line is not output. Thus a **tcp_on_read** handler containing only the instruction '**return 1**' can be used to suppress output of particular lines (see, however, **tcp_filter** above). However, the line is still stored in **TCP_LINE** and **tcp_lines**; this occurs after all **tcp_on_read** processing.

TCP UTILITY PARAMETERS

These parameters are controlled by the function system; they may be read directly, but should not usually be set by user code.

tcp_aliases
Associative array. The keys are the names of sessions established with **tcp_open**; each value is a space−separated list of aliases which refer to that session.

tcp_by_fd
Associative array. The keys are session file descriptors; each value is the name of that session.

tcp_by_name
Associative array. The keys are the names of sessions; each value is the file descriptor associated with that session.

TCP EXAMPLES

Here is a trivial example using a remote calculator.

To create a calculator server on port 7337 (see the **dc** manual page for quite how infuriating the underlying command is):

tcp_proxy 7337 dc

To connect to this from the same host with a session also named '**dc**':

tcp_open localhost 7337 dc

To send a command to the remote session and wait a short while for output (assuming **dc** is the current session):

tcp_command 2 4 + p

To close the session:

tcp_close

The **tcp_proxy** needs to be killed to be stopped. Note this will not usually kill any connections which have

already been accepted, and also that the port is not immediately available for reuse.

The following chunk of code puts a list of sessions into an xterm header, with the current session followed by a star.

```
print −n "\033]2;TCP:" ${(k)tcp_by_name:/$TCP_SESS/$TCP_SESS\*} "\a"
```

TCP BUGS

The function **tcp_read** uses the shell's normal **read** builtin. As this reads a complete line at once, data arriving without a terminating newline can cause the function to block indefinitely.

Though the function suite works well for interactive use and for data arriving in small amounts, the performance when large amounts of data are being exchanged is likely to be extremely poor.