TCP

The TCP module implements the layer 4 Transmission Control Protocol of the Internet Protocol. It processes TCP headers on behalf of applications and runs the TCP state machine for conenction setup, data transfer and connection teardown. The module uses the data types and definitions from the *ip.h* header file in the IP module. The module itself is machine independent and does not require any configuration.

Process Information

Prototype Name	tcp
Link Order	does not matter
Process Name	“tcp”

Process Operation

Active and Passive Opens

The TCP layer distinguishes between ‘listener’ dataflows that are open to receive any packets on that port, but are not bound to a remote IP address, and ‘connected’ dataflows which represent a point-to-point connection. The type of the dataflow is determined at *OPEN* time. Listener dataflows are created by sending an open request to the TCP layer with no destination IP address. This is termed a passive open. In order to transmit data packets, the TCP layer must have a dataflow in the ESTABLISHED state. This can result either from a file being opened with a destination IP address set in its open URL (an active open) or an incoming SYN packet converting a passive dataflow into an active one. Unlike the IP layer, packets will be passed up on both active and passive files. Once a dataflow moves out of the passive state, it must be explicitly closed and re-opened to put it back into the LISTEN state inside TCP.

Errors

The TCP code distinguishes between expected ‘errors’ that are anticipated by the protocol (for example packets arriving late) and indications that the remote state machine is not in the same state as the local machine. Anticipated errors are silently ignored. State errors cause the TCP code to send RESET packets to the remote host. As the packet may be being sent to a port that is not otherwise active, the TCP code handles such resets internally, opening a special file to the remote end and transmitting a single IP datagram containing a TCP header with the RESET bit set. To prevent loops, the code never generates a RESET for a TCP packet that had the RESET bit set. It will, though, generate ICMP port unavailable messages for all ports on which there is no listener.

In the case of unexpected messages from the application, for example duplicate *CLOSE* messages, the errored messages are mostly returned silently as though they had been handled correctly.

The TCP State Machine

Almost all of the TCP module operation is concerned with handling messages for application dataflows. The only internal messaging is to generate error or reset requests for incoming packets not associated with known dataflows. Mostly, incoming messages, from applications and from the IP layer, are used to drive the TCP Finite State Machine according to the standard protocol. The protocol machine can be split into three main phases: open; setup/data; and termination. These are presented in three separate diagrams in order to make the state charts readable. In the charts: 'send X' means sending a TCP packet to the remote host with (at least) the 'X' bit set in the flags; 'send data' sends data to the remote host if the window is open; 'reply N' sends reply to an *OPEN* or *CLOSE* message to the application with error code 'N' and 'error E' passes up the received message to the application with error code 'E'.

	NEW
ACTIVEOPEN	save message open file send SYN start timer ->SYN_SENT
PASSIVEOPEN	reply 0 ->LISTEN

No other events are possible in the *NEW* state, which is created only on receipt of an *OPEN* message. In the data phase (state ESTABLISHED) a long-running timer is used to retransmit the last packet (or just an ACK) to keep the window open. The events are determined from the settings of the flag bits in the packets as follows:

ACK	FIN	RST	SYN	
x	x	1	x	RESET
0	0	0	1	SYN
1	0	0	1	SYNACK
0	0	0	0	DATA
1	0	0	0	ACK
0	1	0	0	FIN
1	1	0	0	FINACK

In the ACK event, in state ESTABLISHED, the state machine is re-run to process any incoming data after the ACK has been processed.

	SYN_SENT	LISTEN	SYN_RECEIVED	ESTABLISHED
RESET	(ignore)	(ignore)	cancel timerACTIVE-OPEN?Y: reply ENOLINKstart timer->CLOSEDN: ->LISTEN	cancel timer-error ENOLINK->CLOSE_WAIT
SYN	send SYN+ACK->SYN_RECEIVED	send SYN+ACKstart timer->SYN_RECEIVED	send RESET	cancel timer-error ENO-LINKsend RESET->CLOSE_WAIT
SYNACK	reply 0send ACK->ESTABLISHED	send RESET	send RESET	cancel timer-error ENO-LINKsend RESET->CLOSE_WAIT
SEND	add to output queue	add to output queue	add to ouput queue	send data
TIMEOUT	reply ENOLINKstart timer->CLOSED	(no timer)	ACTIVE-OPEN?Y: reply ENOLINKstart timer->CLOSEDN: ->LISTEN	if no activity?Y: send ACK==========start timer
CLOSE	cancel timerreply ENOLINKstart timer->CLOSED	save messagestart timer->CLOSED	cancel timersave messagesend FINACTIVE-OPEN?Y: reply ENO-LINK==========start timer->FIN_WAIT_ONE	cancel timersave messagesend FINstart timer->FIN_WAIT_ONE
ACK	send RESET	send RESET	correct ack?Y: reply 0send datastart timer->ESTABLISHEDN: (ignore)	remove ACKed datasend queued datarun DATA
DATA	send RESET	send RESET	send RESET	check seq. number-pass up new data
FINACK	send RESET	send RESET	send RESET	cancel timercheck seqsend ACKer-ror ENOLINK->CLOSE_WAIT
FIN	send RESET	send RESET	send RESET	cancel timersend ACKerror ENOLINK->CLOSE_WAIT

	FINWAIT1	FINWAIT2	CLOSING	LASTACK	TIMEWAIT	CLOSEWAIT	CLOSED
RESET	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)
SYN	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)
SYNACK	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)
SEND	reply 0	reply 0	reply 0	reply 0	reply 0	reply 0	cannot occur
TIMEOUT	send ACK-start timer->FINWAIT2	send RESET-start timer->TIMEWAIT	send RESET-start timer->TIMEWAIT	start timer->CLOSED	start timer->CLOSED	(no timer)	return msgsreply 0delete file
CLOSE	reply 0	reply 0	reply 0	reply 0	reply 0	save messagesend FINstart timer->LASTACK	cannot occur
ACK	cancel timer-start timer->FINWAIT2	cancel timersend FIN+ACKstart timer->TIMEWAIT	cancel timer-start timer->TIMEWAIT	cancel timer-start timer->CLOSED	(ignore)	(ignore)	(ignore)
DATA	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)
FINACK	cancel timersend ACKstart timer->TIMEWAIT	cancel timersend FIN+ACKstart timer->TIMEWAIT	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)
FIN	cancel timersend FIN+ACKstart timer->CLOSING	cancel timersend FIN+ACKstart timer->TIMEWAIT	(ignore)	(ignore)	(ignore)	(ignore)	(ignore)

Messages

The module accepts the *Standard* messageset with the following processing.

CLOSE messages inject the *CLOSE* event into the Finite State machine from the main process.

FETMBLK/GETMBLK messages are added to the queue of input requests for the dataflow in the queue handler. Replies to GETMBLK messages from the lower layer inject the appropriate event in to the Finite State Machine according to the table of flags above. Replied to queued requests are generated from the Finite State Machine when the *DATA* event is processed in the *ESTABLISHED* state. The buffer is returned with the read pointer set just after the end of the TCP header.

FLUSH messages set the PUSH bit on the next TCP header that is sent to the remote host.

NEWMBLK messages are processed within the queue handler. The message is passed downstream after increasing the request size to allow for the TCP header. The reply is passed back upwards after the header space is reserved at the start of the buffer.

OPEN messages are handled in the main process. The *ipaddr* field of the URL determines if this is an active or passive open (see above) and the *port* field contains the TCP local port number. By the time a dataflow reaches TCP, the destination machine's IP address should have been resolved (for example using the DNS or NIS services). The TCP process does not look at the *host* field in the URL, only the *ipaddr* field.

OUTMBLK/PUTMBLK messages run the *SEND* event in the Finite State Machine (for non-zero length messages, which are otherwised passed downstream from the queue handler). Messages that are to be sent before the connection is in a suitable state are buffered within the TCP layer. Data are transmitted downwards using *OUTMBLK* messages in case they need to be re-transmitted, and the buffers are freed after the correct *ACK* arrives using zero-length *PUTMBLK* messages.

RETMBLK messages are passed downstream to free the buffer. The TCP process will also generate *RETMBLK* messages internally after processing incoming data, identified by having the address of the *tcp_file* data stream in the message's source context.

TIMEOUT messages inject the *TIMEOUT* event into the Finite State Machine.

Shared Library Macros and Routines

tcp_new_port

int *tcp_new_port*(**void**)

The *tcp_new_port* routine returns a TCP port number outside the range of reserved ports that is not otherwise in use within the machine.