There are two data types that are used for the lwIP API. They are: **netbuf** and **netconn**.

# Netbufs:

**Netbufs** are buffers that are used for sending and receiving data. Internally, a netbuf is associated with a **pbuf.**

A **pbuf** is lwIP's internal representation of a packet. **Pbufs** are of three types, *PBUF\_RAM*, *PBUF\_ROM*, and *PBUF\_POOL*.

**Netbufs** that have been received from the network also contain the IP address and port number of the originator of the packet. Functions for extracting those values exist.

**Network Interface**

Sample code:

struct netif {

struct netif \*next;

char name[2]; // A two-character name for the interface

int num;

struct ip\_addr ip\_addr; //IP address of the host

struct ip\_addr netmask;

struct ip\_addr gw;

void (\* input)(struct pbuf \*p, struct netif \*inp);

int (\* output)(struct netif \*netif, struct pbuf \*p,

struct ip\_addr \*ipaddr);

void \*state;

};

## Command sets for Netbuf:

1. struct netbuf \* netbuf new(void)
2. void netbuf\_delete(struct netbuf \*)
3. int netbuf\_free(struct netbuf \*buf)

Sample code:

struct netbuf \*buf;

buf = netbuf\_new(); /\* create a new netbuf \*/

netbuf\_alloc(buf, 100); /\* allocate 100 bytes of buffer \*/

/\* do something with the netbuf \*/

netbuf\_delete(buf); /\* deallocate netbuf \*/

1. void \* netbuf\_alloc(struct netbuf \*buf, int size)
2. int netbuf\_ref(struct netbuf \*buf, void \*data, int size)  
   Associates the external memory pointer to by the data pointer with the **netbuf** buf.

Sample code:

struct netbuf \*buf;

char string[] = "A string"; /\* create a new netbuf \*/

buf = netbuf\_new();

/\* refernce the string \*/

netbuf\_ref(buf, string, sizeof(string));

/\* do something with the netbuf \*/

netbuf\_delete(buf);

1. int netbuf\_len(struct netbuf \*buf);  
   Returns the total length of the data in the **netbuf** buf.
2. int netbuf\_data(struct netbuf \*buf, void \*\*data, int \*len);  
   This function is used to obtain a pointer to and the length of a block of data in the **netbuf** buf. If the **netbuf** is fragmented, the user can use netbuf\_first() and netbuf\_next() in order to reach all data in the **netbuf**.
3. void netbuf\_first(struct netbuf \*buf);  
   Resets the fragment pointer in the **netbuf** buf so that it points to the first fragment.

Sample code:

char \*data;

int len;

netbuf\_data(buf, &data, &len);

while(netbuf\_next(buf) >= 0)

/\* do something with the data \*/

1. void netbuf\_copy(struct netbuf \*buf, void \*data, int len);  
   Copies all of the data from the **netbuf** buf into the memory pointed to by data even if the **netbuf** buf is fragmented.

Sample code:

char data[200];

netbuf\_copy(buf, data, 200);

1. void netbuf\_chain(struct netbuf \*head, struct netbuf \*tail);  
   Chains the two **netbufs** head and tail together so that the data in tail will become the last fragment(s) in head.
2. struct ip\_addr \* netbuf\_fromaddr(struct netbuf \*buf);  
   Returns the IP address of the host the **netbuf** buf was received from.
3. u16\_t **netbuf\_fromport** (struct netbuf \*buf );  
   Returns the port number of the host the *netbuf* *buf* was received from.

## Network connection commands

1. struct netconn \* netconn\_new(enum netconn type type);  
   Creates a new connection abstraction structure. The argument can be one of NETCONN\_TCP (0x10) or NETCONN\_UDP (0x20). (Still no connection is established).
2. void netconn\_delete(struct netconn \*conn);  
   Deallocates the netconn conn.
3. enum netconn\_type netconn\_type(struct netconn \*conn);  
   Returns the type of the connection *conn*, either NETCONN\_TCP or NETCONN\_UDP.
4. int netconn\_peer(struct netconn \*conn, struct ip addr \*\*addr, unsigned short port);  
   This function is used to obtain the IP address and port of the remote end of a connection.
5. int netconn\_addr(struct netconn \*conn, struct ip addr \*\*addr, unsigned short port);  
   This function is used to obtain the local IP address and port number of the connection *conn*.
6. int netconn\_bind(struct netconn \*conn, struct ip addr \*addr, unsigned short port);  
   Binds the connection *conn* to the local IP address addr and TCP or UDP port port.
7. int netconn\_connect(struct netconn \*conn, struct ip addr \*remote addr, unsigned short remote port);  
   In the case of UDP, sets the remote receiver as given by remote\_addr and remote\_port of UDP messages sent over the connection. For TCP, netconn\_connect() opens a connection with the remote host.
8. int netconn\_listen(struct netconn \*conn);  
   Puts the TCP connection *conn* into the TCP LISTEN state.
9. struct netconn \* netconn\_accept(struct netconn \*conn);  
   Blocks the process until a connection request from a remote host arrives on the TCP connection *conn*. The connection must be in the LISTEN state so netconn\_listen() must be called prior to netconn\_accept(). When a connection is established with the remote host, a new connection structure is returned.

Sample code:

*/\*This example shows how to open a TCP server on port 2000.\*/*

int main()

{

struct netconn \*conn, \*newconn;

/\* create a connection structure \*/

conn = netconn\_new(NETCONN\_TCP);

/\* bind the connection to port 2000 on any local IP address \*/

netconn\_bind(conn, NULL, 2000);

/\* tell the connection to listen for incoming connection requests \*/

netconn\_listen(conn);

/\* block until we get an incoming connection \*/

newconn = netconn\_accept(conn);

/\* do something with the connection \*/

process\_connection(newconn);

/\* deallocate both connections \*/

netconn\_delete(newconn);

netconn\_delete(conn);

}

1. struct netbuf \* netconn\_recv(struct netconn \*conn);  
   Blocks the process while waiting for data to arrive on the connection *conn*. If the connection has been closed by the remote host, NULL is returned, otherwise a **netbuf** containing the received data is returned.

Sample code:

/\*assume a connection has been established before the call to example\_function().\*/

while((buf = netconn\_recv(conn)) != NULL)

{ do\_something(buf); }

/\* the connection has now been closed by the other end, so we close our end \*/

netconn\_close(conn);

1. int netconn\_write(struct netconn \*conn, void \*data, int len, unsigned int flags);  
   This function is only used for TCP connections. It puts the data pointed to by data on the output queue for the TCP connection *conn*.  
   The flag can be:  
   #define NETCONN\_NOCOPY 0x00   
   #define NETCONN\_COPY 0x01  
   When passed the flag NETCONN\_COPY the data is copied into internal buffers which is allocated for the data. It is inefficient both in terms of execution time and memory usage.  
   If the flag NETCONN\_NOCOPY is used, the data is not copied but rather referenced. The data must not be modified after the call, since the data can be put on the retransmission queue for the connection, and stay there for an indeterminate amount of time.

Sample code:

int main()

{

struct netconn \*conn;

char data[10];

char text[] = "Static text";

int i;

/\* set up the connection conn \*/

/\* [...] \*/

/\* create some arbitrary data \*/

for(i = 0; i < 10; i++)

data[i] = i;

netconn\_write(conn, data, 10, NETCONN\_COPY);

netconn\_write(conn, text, sizeof(text), NETCONN\_NOCOPY);

/\* the data can be modified \*/

for(i = 0; i < 10; i++)

data[i] = 10 - i;

/\* take down the connection conn \*/

netconn\_close(conn);

}

1. int netconn send(struct netconn \*conn, struct netbuf \*buf)  
   Send the data in the **netbuf** buf on the UDP connection *conn*. The data in the **netbuf** should not be too large since IP fragmentation is not used (Not larger than 1000 bytes).

Sample code:

/\*This example shows how to send some UDP data to UDP port 7000 on a remote host with IP address 10.0.0.1.\*/

int main()

{

struct netconn \*conn;

struct netbuf \*buf;

struct ip\_addr addr;

char \*data;

char text[] = "A static text";

int i;

/\* create a new connection \*/

conn = netconn\_new(NETCONN\_UDP);

/\* set up the IP address of the remote host \*/

addr.addr = htonl(0x0a000001);

/\* connect the connection to the remote host \*/

netconn\_connect(conn, &addr, 7000);

/\* create a new netbuf \*/

buf = netbuf\_new();

data = netbuf\_alloc(buf, 10);

/\* create some arbitrary data \*/

for(i = 0; i < 10; i++)

data[i] = i;

/\* send the arbitrary data \*/

netconn\_send(conn, buf);

/\* reference the text into the netbuf \*/

netbuf\_ref(buf, text, sizeof(text));

/\* send the text \*/

netconn\_send(conn, buf);

/\* deallocate connection and netbuf \*/

netconn\_delete(conn);

netconn\_delete(buf);

}

1. int netconn close(struct netconn \*conn);  
   Closes the connection *conn*.

**IP processing**

lwIP implements only the most basic functionality of IP. It can send, receive and forward packets, but cannot send or receive fragmented IP packets nor handle packets with IP options.

**Sending and Receiving packets**

For incoming IP packets, processing begins when the **ip\_input()** function is called by a network device driver.

An outgoing packet is handled by the function **ip\_output()**, which uses the function **ip\_route()** to find the appropriate network interface to transmit the packet on.

The transport layer protocols UDP and TCP need to have the destination IP address; This is done by letting the transport layer functions call the ip\_route() function. The ip\_route() function finds the appropriate network interface by linearly searching the list of network interfaces.

When the outgoing network interface is determined, the packet is passed to **ip\_output\_if()** which takes the outgoing network interface as an argument.

Forwarding packets

If a received packet has different ip address than our device, the packet should be forwarded using **ip\_forward()**.

**TCP processing**

TCP is a transport layer protocol that provides a reliable byte stream service to the application layer.

Sending data

When an application wants to send TCP data, tcp\_write() is called.

The function tcp\_write() passes control to tcp\_enqueue(). tcp\_enqueue() breaks the data into appropriate sized TCP segments and put the segments on the transmission queue for the connection.

The function tcp\_output() will then check if it is possible to send the data, i.e., if there is enough space in the receiver's window and if the congestion window is large enough and if so, sends the data using ip\_route() and ip\_output\_if().

Receiving data

Input processing begins when ip\_input() after verifying the IP header hands over a TCP segment to tcp\_input().

In this function the initial sanity checks (i.e., checksumming and TCP options parsing) are done as well as deciding to which TCP connection the segment belongs.

The segment is then processed by tcp\_process().

The function tcp\_receive() will be called if the connection is in a state to accept data from the network. If so, tcp\_receive() will pass the segment up to an application program.

if an ACK for data was received the receiver might be willing to accept more data and therefore tcp\_output() is called.

Sample code:

struct tcp\_pcb {

struct tcp\_pcb \*next;

enum tcp\_state state; /\* TCP state \*/

void (\* accept)(void \*arg, struct tcp\_pcb \*newpcb);

void \*accept\_arg;

struct ip\_addr local\_ip;

u16\_t local\_port;

struct ip\_addr dest\_ip;

u16\_t dest\_port;

u32\_t rcv\_nxt, rcv\_wnd; /\* receiver variables \*/

u16\_t tmr;

u32\_t mss; /\* maximum segment size \*/

u8\_t flags;

/\* rttest, rtseq, sa, and sv are used for the round-trip time estimation. \*/

u16\_t rttest; /\* rtt estimation \*/

u32\_t rtseq; /\* sequence no for rtt estimation \*/

s32\_t sa, sv; /\* rtt average and variance \*/

u32\_t rto; /\* retransmission time-out \*/

u32\_t lastack; /\* last ACK received \*/

u8\_t dupacks; /\* number of duplicate ACKs \*/

u32\_t cwnd, u32\_t ssthresh; /\* congestion control variables \*/

u32\_t snd\_ack, snd\_nxt, /\* sender variables \*/

snd\_wnd, snd\_wl1, snd\_wl2, snd\_lbb;

void (\* recv)(void \*arg, struct tcp\_pcb \*pcb, struct pbuf \*p);

void \*recv\_arg;

struct tcp\_seg \*unsent, \*unacked, /\* queues \*/

\*ooseq;

};

The function pointer recv and recv\_arg are used when passing received data to the application layer. Data that has been received from the application but has not been sent is queued in *unsent* and data that has been sent but not yet acknowledged by the remote host is held in *unacked*. Received data that is out of sequence is buffered in *ooseq*.

The tcp\_seg is the internal representation of a TCP segment.

Sample code:

struct tcp\_seg {

struct tcp\_seg \*next;

u16\_t len;

struct pbuf \*p;

struct tcp\_hdr \*tcphdr;

void \*data;

u16\_t rtime;

};

## BSD SOCKET

1. The socket() call allocates a BSD socket. Only UDP (SOCK DGRAM) or TCP (SOCK STREAM) sockets can be used.

Sample code:

int socket(int domain, int type, int protocol)

{

struct netconn \*conn;

int i;

/\* create a netconn \*/

switch(type)

{

case SOCK\_DGRAM:

conn = netconn\_new(NETCONN\_UDP);

break;

case SOCK\_STREAM:

conn = netconn\_new(NETCONN\_TCP);

break;

}

/\* find an empty place in the sockets[] list \*/

for(i = 0; i < sizeof(sockets); i++) {

if(sockets[i] == NULL) {

sockets[i] = conn;

return i;

}

} return -1;

}

1. The bind() call binds the BSD socket to a local address.

Sample code:

int bind(int s, struct sockaddr \*name, int namelen)

{

struct netconn \*conn;

struct ip\_addr \*remote\_addr;

unsigned short remote\_port;

remote\_addr = (struct ip\_addr \*)name->sin\_addr;

remote\_port = name->sin\_port;

conn = sockets[s];

netconn\_bind(conn, remote\_addr, remote\_port);

return 0;

}

1. connect()

Sample code:

int connect(int s, struct sockaddr \*name, int namelen)

{

struct netconn \*conn;

struct ip\_addr \*remote\_addr;

unsigned short remote\_port;

remote\_addr = (struct ip\_addr \*)name->sin\_addr;

remote\_port = name->sin\_port;

conn = sockets[s];

netconn\_connect(conn, remote\_addr, remote\_port);

return 0;

}

1. Listen()  
   Listen() can only be used for TCP connections.

Sample code:

int listen(int s, int backlog)

{

netconn\_listen(sockets[s]);

return 0;

}

1. accept();  
   The accept() call is used to wait for incoming connections on a TCP socket that previously has been set into LISTEN state by a call to listen(). The call to accept() blocks until a connection has been established with a remote host.

Sample code:

int accept(int s, struct sockaddr \*addr, int \*addrlen)

{

struct netconn \*conn, \*newconn;

struct ip\_addr \*addr;

unsigned short port;

int i;

conn = sockets[s];

newconn = netconn\_accept(conn);

/\* get the IP address and port of the remote host \*/

netconn\_peer(conn, &addr, &port);

addr->sin\_addr = \*addr;

addr->sin\_port = port;

/\* allocate a new socket identifier \*/

for(i = 0; i < sizeof(sockets); i++) {

if(sockets[i] == NULL) {

sockets[i] = newconn;

return i;

}

}

return -1;

}

### Sending and receiving data

1. Send();  
   In the BSD socket API, the send() call is used in both UDP and TCP connection for sending data. Before a call to send() the receiver of the data must have been set up using connect().

Sample code:

int send(int s, void \*data, int size, unsigned int flags)

{

struct netconn \*conn;

struct netbuf \*buf;

conn = sockets[s];

switch(netconn\_type(conn)) {

case NETCONN\_UDP:

/\* create a buffer \*/

buf = netbuf\_new();

/\* make the buffer point to the data that should be sent \*/

netbuf\_ref(buf, data, size);

/\* send the data \*/

netconn\_send(sock->conn.udp, buf);

/\* deallocated the buffer \*/

netbuf\_delete(buf);

break;

case NETCONN\_TCP:

netconn\_write(conn, data, size, NETCONN\_COPY);

break;

}

return size;

}

1. sendto() and sendmsg()  
   They are similar to the send() call, but they allow the application program to specify the receiver of the data in the parameters to the call. sendto() and sendmsg() only can be used for UDP connections.  
   The implementation uses netconn\_connect() to set the receiver of the datagram and must therefore reset the remote IP address and port number if the socket was previously connected.

Sample code:

int sendto(int s, void \*data, int size, unsigned int flags, struct sockaddr \*to, int tolen)

{

struct netconn \*conn;

struct ip\_addr \*remote\_addr, \*addr;

unsigned short remote\_port, port;

int ret;

conn = sockets[s];

/\* get the peer if currently connected \*/

netconn\_peer(conn, &addr, &port);

remote\_addr = (struct ip\_addr \*)to->sin\_addr;

remote\_port = to->sin\_port;

netconn\_connect(conn, remote\_addr, remote\_port);

ret = send(s, data, size, flags);

/\* reset the remote address and port number of the connection \*/

netconn\_connect(conn, addr, port);

}

1. write()  
   The write() call sends data on a connection and can be used for both UDP and TCP connections. For TCP connections, this maps directly to the lwIP API function netconn write(). For UDP, the BSD socket function write() function is equvalent to the send() function.

Sample code:

int write(int s, void \*data, int size)

{

struct netconn \*conn;

conn = sockets[s];

switch(netconn\_type(conn)) {

case NETCONN\_UDP:

send(s, data, size, 0);

break;

case NETCONN\_TCP:

netconn\_write(conn, data, size, NETCONN\_COPY);

break;

}

return size;

}

1. recv() and read() calls  
   The recv() and read() calls are used on a connected socket to receive data. They can be used for both TCP and UDP connections.

Sample code:

int recv(int s, void \*mem, int len, unsigned int flags)

{

struct netconn \*conn;

struct netbuf \*buf;

int buflen;

conn = sockets[s];

buf = netconn\_recv(conn);

buflen = netbuf\_len(buf);

/\* copy the contents of the received buffer into the supplied memory pointer mem \*/

netbuf\_copy(buf, mem, len);

netbuf\_delete(buf);

/\* if the length of the received data is larger than len, this data is discarded and we return len. otherwise we return the actual length of the received data \*/

if(len > buflen)

{ return buflen; }

else

{ return len; }

}

Sample code:

int read(int s, void \*mem, int len)

{

return recv(s, mem, len, 0);

}

1. recvfrom() and recvmsg()  
   The recvfrom() and recvmsg() calls are similar to the recv() call but differ in that the IP address and port number of the sender of the data can be obtained through the call.

Sample code:

int recvfrom(int s, void \*mem, int len, unsigned int flags, struct sockaddr \*from, int \*fromlen)

{

struct netconn \*conn;

struct netbuf \*buf;

struct ip\_addr \*addr;

unsigned short port;

int buflen;

conn = sockets[s];

buf = netconn\_recv(conn);

buflen = netbuf\_len(conn);

/\* copy the contents of the received buffer into the supplied memory pointer \*/

netbuf\_copy(buf, mem, len);

addr = netbuf\_fromaddr(buf);

port = netbuf\_fromport(buf);

from->sin\_addr = \*addr;

from->sin\_port = port;

\*fromlen = sizeof(struct sockaddr);

netbuf\_delete(buf);

/\* if the length of the received data is larger than len, this data is discarded and we return len. otherwise we return the actual length of the received data \*/

if(len > buflen)

{ return buflen; }

else

{ return len; }

}

The lwIP library provides two different APIs: RAW API and Socket API.

## Raw API

The Raw API is callback based. Applications obtain access directly into the TCP stack and vice-versa. As a result, there is no extra socket layer, and using the Raw API provides excellent performance at the price of compatibility with other TCP stacks.

In addition to the lwIP RAW API, the Xilinx adapters provide the xemacif\_input utility function for receiving packets. This function must be called at frequent intervals to move the received packets from the interrupt handlers to the lwIP stack.

Depending on the type of packet received, lwIP then calls registered application callbacks. The <Vitis\_install\_path>/sw/ThirdParty/sw\_services/lwip211/src/lwip-2.1.1/doc/rawapi.txt file describes the lwIP Raw API.

# Functions:

[int xemacif\_input(struct netif \*netif);](https://docs.xilinx.com/r/2021.2-English/oslib_rm/xemacif_input)

The Xilinx lwIP adapters work in interrupt mode. The receive interrupt handlers move the packet data from the EMAC/GigE and store them in a queue. The xemacif\_input() function takes those packets from the queue, and passes them to lwIP; consequently, this function is required for lwIP operation in RAW mode.

[err\_t tcp\_write (struct tcp\_pcb \*pcb , const void \*dataptr , u16\_t len , u8\_t copy );](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html#ga6b2aa0efbf10e254930332b7c89cd8c5)

Write data for sending (but does not send it immediately). This function enqueues the data pointed to by the argument *dataptr*. The *[tcp](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "ga6b2aa0efbf10e254930332b7c89cd8c5)*[\_](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "ga6b2aa0efbf10e254930332b7c89cd8c5)*[write](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "ga6b2aa0efbf10e254930332b7c89cd8c5)*[()](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "ga6b2aa0efbf10e254930332b7c89cd8c5) function will fail and return *ERR\_MEM* if the length of the data exceeds the current send buffer size or if the length of the queue of outgoing segment is larger than the upper limit defined in *lwipopts.h*. The number of bytes available in the output queue can be retrieved with the *[tcp\_sndbuf](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "gad1a95f09deb49cd7341d35527d6d3e2f)*[()](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "gad1a95f09deb49cd7341d35527d6d3e2f) function.

[u16\_t tcp\_sndbuf (struct tcp\_pcb \**pcb* );](https://doc.ecoscentric.com/ref/lwip-api-raw-tcp-write.html)

The number of bytes available in the output queue can be retrieved with the *[tcp\_sndbuf](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "gad1a95f09deb49cd7341d35527d6d3e2f)*[()](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "gad1a95f09deb49cd7341d35527d6d3e2f) function.

err\_t [tcp\_connect](https://www.nongnu.org/lwip/2_0_x/group__tcp__raw.html#ga9a31deea4cadacd39f9485f37cfdd012) (struct [tcp\_pcb](https://www.nongnu.org/lwip/2_0_x/structtcp__pcb.html) \*pcb, const [ip\_addr\_t](https://www.nongnu.org/lwip/2_0_x/group__ipaddr.html" \l "ga16ef96d6cde029029bbf47fee35fd67a) \*ipaddr, u16\_t port, [tcp\_connected\_fn](https://www.nongnu.org/lwip/2_0_x/tcp_8h.html" \l "a939867106bd492caf2d85852fb7f6ae8) connected);

Connects to another host. The function given as the connected argument will be called when the connection has been established. Sets up the *pcb* to connect to the remote host and sends the initial *SYN* segment which opens the connection.

The *[tcp\_connect](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "ga9a31deea4cadacd39f9485f37cfdd012)*[()](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "ga9a31deea4cadacd39f9485f37cfdd012) function returns immediately; it does not wait for the connection to be properly setup. Instead, it will call the function specified as the fourth argument (the "*connected*" argument) when the connection is established. If the connection could not be properly established, either because the other host refused the connection or because the other host didn't answer, the "*err*" callback function of this *pcb* (registered with *tcp\_err* -see bellow) will be called.

void [tcp\_err](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html#gae1346c4e34d3bc7c01e1b47142ab3121) (struct [tcp\_pcb](https://www.nongnu.org/lwip/2_1_x/structtcp__pcb.html) \*pcb, [tcp\_err\_fn](https://www.nongnu.org/lwip/2_1_x/tcp_8h.html" \l "a1b4f9e3551e575c0ef06d6daa7f06e55) err)

Used to specify the function that should be called when a fatal error has occurred on the connection.

If a connection is aborted because of an error, the application is alerted of this event by the err callback. Errors that might abort a connection are when there is a shortage of memory. The callback function to be called is set using the *[tcp\_err](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "gae1346c4e34d3bc7c01e1b47142ab3121)*[()](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html" \l "gae1346c4e34d3bc7c01e1b47142ab3121) function.

The corresponding *pcb* is already freed when this callback is called!

void [tcp\_sent](https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html#ga1596332b93bb6249179f3b89f24bd808) (struct [tcp\_pcb](https://www.nongnu.org/lwip/2_1_x/structtcp__pcb.html) \*pcb, [tcp\_sent\_fn](https://www.nongnu.org/lwip/2_1_x/tcp_8h.html" \l "aa60622ffaa099e97f66fb56e437fca18) sent);

Specifies the callback function that should be called when data has successfully been received (i.e., acknowledged) by the remote host. The *len* argument passed to the callback function gives the amount bytes that was acknowledged by the last acknowledgment.

[err\_t](https://www.nongnu.org/lwip/2_1_x/group__infrastructure__errors.html#gaf02d9da80fd66b4f986d2c53d7231ddb) tcp\_output(struct [tcp\_pcb](https://www.nongnu.org/lwip/2_1_x/structtcp__pcb.html) \* pcb);

Forces all enqueued data to be sent now. Usually used after *tcp\_write* commands.

[void tcp\_recv (struct tcp\_pcb \**pcb* , err\_t *(\*recv)* (void \*arg, struct tcp\_pcb \*tpcb, struct pbuf \*p, err\_t err) );](https://doc.ecoscentric.com/ref/lwip-api-raw-tcp-recv.html)

Sets the callback function that will be called when new data arrives on the connection associated with *pcb*. The callback function will be passed a NULL *pbuf* to indicate that the remote host has closed the connection.

## Socket API

The lwIP socket API provides a BSD socket-style API to programs.

Applications using the Socket API with Xilinx adapters need to spawn a separate thread called xemacif\_input\_thread. This thread takes care of moving received packets from the interrupt handlers to the tcpip\_thread of the lwIP. Application threads that use lwIP must be created using the lwIP sys\_thread\_new API. Internally, this function makes use of the appropriate thread or task creation routines provided by XilKernel or FreeRTOS.

References

* “Design and Implementation of the lwIP TCP/IP Stack”, Adam Dunkels, 2001
* <https://docs.xilinx.com/r/2021.2-English/oslib_rm/LwIP-2.1.1-Library-v1.6>