There are two data types that are used for the lwIP API. These are:

1. netbuf,
2. 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.

Command set:

Set (1)

struct netbuf \* netbuf new(void)

void netbuf delete(struct netbuf \*)

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 \*/

Set (2)

void \* netbuf\_alloc(struct netbuf \*buf, int size)

int netbuf\_ref(struct netbuf \*buf, void \*data, int size)

Associates the external memory pointer to by the data pointer with the netbuf buf.

example:

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);

SET (3)

int netbuf\_len(struct netbuf \*buf)

Returns the total length of the data in the netbuf buf.

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.

void netbuf first(struct netbuf \*buf)

Resets the fragment pointer in the netbuf buf so that it points to the first fragment.

Example:

char \*data;

int len;

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

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

/\* do something with the data \*/

SET (4)

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.

Example:

char data[200];

netbuf\_copy(buf, data, 200);

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.

struct ip\_addr \* netbuf\_fromaddr(struct netbuf \*buf);

Returns the IP address of the host the netbuf buf was received from.

u16\_t **netbuf\_fromport** (struct netbuf \*buf );

Returns the port number of the host the netbuf *buf* was received from.

**Network connection functions**

SET (5)

struct netconn \* netconn\_new(enum netconn type type)

Creates a new connection abstraction structure. The argument can be one of NETCONN\_TCP or NETCONN\_UDP. (Still no connection is established).

void netconn\_delete(struct netconn \*conn);

Deallocates the netconn conn.

enum netconn\_type netconn\_type(struct netconn \*conn)

Returns the type of the connection conn, either NETCONN\_TCP or NETCONN\_UDP.

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.

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.

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.

int netconn\_connect(struct netconn \*conn, struct ip addr \*remote addr, unsigned short remote port)

In 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.

int netconn\_listen(struct netconn \*conn)

Puts the TCP connection conn into the TCP LISTEN state.

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.

EXAMPLE:

Example 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);

}

SET(6)

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.

EXAMPLE:  
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);

SET(7)

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.

EXAMPLE:

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);

}

SET (8)

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).

Example:

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);

}

SET(9)

int netconn close(struct netconn \*conn);

Closes the connection conn.

**BSD SOCKET**

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

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;

}

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

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;

}

connect()

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;

}

Listen()

Listen() can only be used for TCP connections.

int listen(int s, int backlog)

{

netconn\_listen(sockets[s]);

return 0;

}

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.

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**

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().

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;

}

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.

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);

}

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.

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;

}

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.

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; } }

//////////////////

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

{

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

}

recvfrom() and recvmsg() calls

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.

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; }

}