Raw TCP/IP interface for lwIP

Authors: Adam Dunkels, Leon Woestenberg, Christiaan Simons

lwIP provides three Application Program's Interfaces (APIs) for programs

to use for communication with the TCP/IP code:

\* low-level "core" / "callback" or "raw" API.

\* higher-level "sequential" API.

\* BSD-style socket API.

The sequential API provides a way for ordinary, sequential, programs

to use the lwIP stack. It is quite similar to the BSD socket API. The

model of execution is based on the blocking open-read-write-close

paradigm. Since the TCP/IP stack is event based by nature, the TCP/IP

code and the application program must reside in different execution

contexts (threads).

The socket API is a compatibility API for existing applications,

currently it is built on top of the sequential API. It is meant to

provide all functions needed to run socket API applications running

on other platforms (e.g. unix / windows etc.). However, due to limitations

in the specification of this API, there might be incompatibilities

that require small modifications of existing programs.

\*\* Threading

lwIP started targeting single-threaded environments. When adding multi-

threading support, instead of making the core thread-safe, another

approach was chosen: there is one main thread running the lwIP core

(also known as the "tcpip\_thread"). The raw API may only be used from

this thread! Application threads using the sequential- or socket API

communicate with this main thread through message passing.

As such, the list of functions that may be called from

other threads or an ISR is very limited! Only functions

from these API header files are thread-safe:

- api.h

- netbuf.h

- netdb.h

- netifapi.h

- sockets.h

- sys.h

Additionaly, memory (de-)allocation functions may be

called from multiple threads (not ISR!) with NO\_SYS=0

since they are protected by SYS\_LIGHTWEIGHT\_PROT and/or

semaphores.

Only since 1.3.0, if SYS\_LIGHTWEIGHT\_PROT is set to 1

and LWIP\_ALLOW\_MEM\_FREE\_FROM\_OTHER\_CONTEXT is set to 1,

pbuf\_free() may also be called from another thread or

an ISR (since only then, mem\_free - for PBUF\_RAM - may

be called from an ISR: otherwise, the HEAP is only

protected by semaphores).

\*\* The remainder of this document discusses the "raw" API. \*\*

The raw TCP/IP interface allows the application program to integrate

better with the TCP/IP code. Program execution is event based by

having callback functions being called from within the TCP/IP

code. The TCP/IP code and the application program both run in the same

thread. The sequential API has a much higher overhead and is not very

well suited for small systems since it forces a multithreaded paradigm

on the application.

The raw TCP/IP interface is not only faster in terms of code execution

time but is also less memory intensive. The drawback is that program

development is somewhat harder and application programs written for

the raw TCP/IP interface are more difficult to understand. Still, this

is the preferred way of writing applications that should be small in

code size and memory usage.

Both APIs can be used simultaneously by different application

programs. In fact, the sequential API is implemented as an application

program using the raw TCP/IP interface.

--- Callbacks

Program execution is driven by callbacks. Each callback is an ordinary

C function that is called from within the TCP/IP code. Every callback

function is passed the current TCP or UDP connection state as an

argument. Also, in order to be able to keep program specific state,

the callback functions are called with a program specified argument

that is independent of the TCP/IP state.

The function for setting the application connection state is:

- void tcp\_arg(struct tcp\_pcb \*pcb, void \*arg)

Specifies the program specific state that should be passed to all

other callback functions. The "pcb" argument is the current TCP

connection control block, and the "arg" argument is the argument

that will be passed to the callbacks.

--- TCP connection setup

The functions used for setting up connections is similar to that of

the sequential API and of the BSD socket API. A new TCP connection

identifier (i.e., a protocol control block - PCB) is created with the

tcp\_new() function. This PCB can then be either set to listen for new

incoming connections or be explicitly connected to another host.

- struct tcp\_pcb \*tcp\_new(void)

Creates a new connection identifier (PCB). If memory is not

available for creating the new pcb, NULL is returned.

- err\_t tcp\_bind(struct tcp\_pcb \*pcb, ip\_addr\_t \*ipaddr,

u16\_t port)

Binds the pcb to a local IP address and port number. The IP address

can be specified as IP\_ADDR\_ANY in order to bind the connection to

all local IP addresses.

If another connection is bound to the same port, the function will

return ERR\_USE, otherwise ERR\_OK is returned.

- struct tcp\_pcb \*tcp\_listen(struct tcp\_pcb \*pcb)

Commands a pcb to start listening for incoming connections. When an

incoming connection is accepted, the function specified with the

tcp\_accept() function will be called. The pcb will have to be bound

to a local port with the tcp\_bind() function.

The tcp\_listen() function returns a new connection identifier, and

the one passed as an argument to the function will be

deallocated. The reason for this behavior is that less memory is

needed for a connection that is listening, so tcp\_listen() will

reclaim the memory needed for the original connection and allocate a

new smaller memory block for the listening connection.

tcp\_listen() may return NULL if no memory was available for the

listening connection. If so, the memory associated with the pcb

passed as an argument to tcp\_listen() will not be deallocated.

- struct tcp\_pcb \*tcp\_listen\_with\_backlog(struct tcp\_pcb \*pcb, u8\_t backlog)

Same as tcp\_listen, but limits the number of outstanding connections

in the listen queue to the value specified by the backlog argument.

To use it, your need to set TCP\_LISTEN\_BACKLOG=1 in your lwipopts.h.

- void tcp\_accepted(struct tcp\_pcb \*pcb)

Inform lwIP that an incoming connection has been accepted. This would

usually be called from the accept callback. This allows lwIP to perform

housekeeping tasks, such as allowing further incoming connections to be

queued in the listen backlog.

ATTENTION: the PCB passed in must be the listening pcb, not the pcb passed

into the accept callback!

- void tcp\_accept(struct tcp\_pcb \*pcb,

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

err\_t err))

Specified the callback function that should be called when a new

connection arrives on a listening connection.

- err\_t tcp\_connect(struct tcp\_pcb \*pcb, ip\_addr\_t \*ipaddr,

u16\_t port, err\_t (\* connected)(void \*arg,

struct tcp\_pcb \*tpcb,

err\_t err));

Sets up the pcb to connect to the remote host and sends the

initial SYN segment which opens the connection.

The tcp\_connect() 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 below)

will be called.

The tcp\_connect() function can return ERR\_MEM if no memory is

available for enqueueing the SYN segment. If the SYN indeed was

enqueued successfully, the tcp\_connect() function returns ERR\_OK.

--- Sending TCP data

TCP data is sent by enqueueing the data with a call to

tcp\_write(). When the data is successfully transmitted to the remote

host, the application will be notified with a call to a specified

callback function.

- err\_t tcp\_write(struct tcp\_pcb \*pcb, const void \*dataptr, u16\_t len,

u8\_t apiflags)

Enqueues the data pointed to by the argument dataptr. The length of

the data is passed as the len parameter. The apiflags can be one or more of:

- TCP\_WRITE\_FLAG\_COPY: indicates whether the new memory should be allocated

for the data to be copied into. If this flag is not given, no new memory

should be allocated and the data should only be referenced by pointer. This

also means that the memory behind dataptr must not change until the data is

ACKed by the remote host

- TCP\_WRITE\_FLAG\_MORE: indicates that more data follows. If this is given,

the PSH flag is set in the last segment created by this call to tcp\_write.

If this flag is given, the PSH flag is not set.

The tcp\_write() 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() function.

The proper way to use this function is to call the function with at

most tcp\_sndbuf() bytes of data. If the function returns ERR\_MEM,

the application should wait until some of the currently enqueued

data has been successfully received by the other host and try again.

- void tcp\_sent(struct tcp\_pcb \*pcb,

err\_t (\* sent)(void \*arg, struct tcp\_pcb \*tpcb,

u16\_t len))

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.

--- Receiving TCP data

TCP data reception is callback based - an application specified

callback function is called when new data arrives. When the

application has taken the data, it has to call the tcp\_recved()

function to indicate that TCP can advertise increase the receive

window.

- void tcp\_recv(struct tcp\_pcb \*pcb,

err\_t (\* recv)(void \*arg, struct tcp\_pcb \*tpcb,

struct pbuf \*p, err\_t err))

Sets the callback function that will be called when new data

arrives. The callback function will be passed a NULL pbuf to

indicate that the remote host has closed the connection. If

there are no errors and the callback function is to return

ERR\_OK, then it must free the pbuf. Otherwise, it must not

free the pbuf so that lwIP core code can store it.

- void tcp\_recved(struct tcp\_pcb \*pcb, u16\_t len)

Must be called when the application has received the data. The len

argument indicates the length of the received data.

--- Application polling

When a connection is idle (i.e., no data is either transmitted or

received), lwIP will repeatedly poll the application by calling a

specified callback function. This can be used either as a watchdog

timer for killing connections that have stayed idle for too long, or

as a method of waiting for memory to become available. For instance,

if a call to tcp\_write() has failed because memory wasn't available,

the application may use the polling functionality to call tcp\_write()

again when the connection has been idle for a while.

- void tcp\_poll(struct tcp\_pcb \*pcb,

err\_t (\* poll)(void \*arg, struct tcp\_pcb \*tpcb),

u8\_t interval)

Specifies the polling interval and the callback function that should

be called to poll the application. The interval is specified in

number of TCP coarse grained timer shots, which typically occurs

twice a second. An interval of 10 means that the application would

be polled every 5 seconds.

--- Closing and aborting connections

- err\_t tcp\_close(struct tcp\_pcb \*pcb)

Closes the connection. The function may return ERR\_MEM if no memory

was available for closing the connection. If so, the application

should wait and try again either by using the acknowledgment

callback or the polling functionality. If the close succeeds, the

function returns ERR\_OK.

The pcb is deallocated by the TCP code after a call to tcp\_close().

- void tcp\_abort(struct tcp\_pcb \*pcb)

Aborts the connection by sending a RST (reset) segment to the remote

host. The pcb is deallocated. This function never fails.

ATTENTION: When calling this from one of the TCP callbacks, make

sure you always return ERR\_ABRT (and never return ERR\_ABRT otherwise

or you will risk accessing deallocated memory or memory leaks!

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() function.

- void tcp\_err(struct tcp\_pcb \*pcb, void (\* err)(void \*arg,

err\_t err))

The error callback function does not get the pcb passed to it as a

parameter since the pcb may already have been deallocated.

--- Lower layer TCP interface

TCP provides a simple interface to the lower layers of the

system. During system initialization, the function tcp\_init() has

to be called before any other TCP function is called. When the system

is running, the two timer functions tcp\_fasttmr() and tcp\_slowtmr()

must be called with regular intervals. The tcp\_fasttmr() should be

called every TCP\_FAST\_INTERVAL milliseconds (defined in tcp.h) and

tcp\_slowtmr() should be called every TCP\_SLOW\_INTERVAL milliseconds.

--- UDP interface

The UDP interface is similar to that of TCP, but due to the lower

level of complexity of UDP, the interface is significantly simpler.

- struct udp\_pcb \*udp\_new(void)

Creates a new UDP pcb which can be used for UDP communication. The

pcb is not active until it has either been bound to a local address

or connected to a remote address.

- void udp\_remove(struct udp\_pcb \*pcb)

Removes and deallocates the pcb.

- err\_t udp\_bind(struct udp\_pcb \*pcb, ip\_addr\_t \*ipaddr,

u16\_t port)

Binds the pcb to a local address. The IP-address argument "ipaddr"

can be IP\_ADDR\_ANY to indicate that it should listen to any local IP

address. The function currently always return ERR\_OK.

- err\_t udp\_connect(struct udp\_pcb \*pcb, ip\_addr\_t \*ipaddr,

u16\_t port)

Sets the remote end of the pcb. This function does not generate any

network traffic, but only set the remote address of the pcb.

- err\_t udp\_disconnect(struct udp\_pcb \*pcb)

Remove the remote end of the pcb. This function does not generate

any network traffic, but only removes the remote address of the pcb.

- err\_t udp\_send(struct udp\_pcb \*pcb, struct pbuf \*p)

Sends the pbuf p. The pbuf is not deallocated.

- void udp\_recv(struct udp\_pcb \*pcb,

void (\* recv)(void \*arg, struct udp\_pcb \*upcb,

struct pbuf \*p,

ip\_addr\_t \*addr,

u16\_t port),

void \*recv\_arg)

Specifies a callback function that should be called when a UDP

datagram is received.

--- System initalization

A truly complete and generic sequence for initializing the lwip stack

cannot be given because it depends on the build configuration (lwipopts.h)

and additional initializations for your runtime environment (e.g. timers).

We can give you some idea on how to proceed when using the raw API.

We assume a configuration using a single Ethernet netif and the

UDP and TCP transport layers, IPv4 and the DHCP client.

Call these functions in the order of appearance:

- stats\_init()

Clears the structure where runtime statistics are gathered.

- sys\_init()

Not of much use since we set the NO\_SYS 1 option in lwipopts.h,

to be called for easy configuration changes.

- mem\_init()

Initializes the dynamic memory heap defined by MEM\_SIZE.

- memp\_init()

Initializes the memory pools defined by MEMP\_NUM\_x.

- pbuf\_init()

Initializes the pbuf memory pool defined by PBUF\_POOL\_SIZE.

- etharp\_init()

Initializes the ARP table and queue.

Note: you must call etharp\_tmr at a ARP\_TMR\_INTERVAL (5 seconds) regular interval

after this initialization.

- ip\_init()

Doesn't do much, it should be called to handle future changes.

- udp\_init()

Clears the UDP PCB list.

- tcp\_init()

Clears the TCP PCB list and clears some internal TCP timers.

Note: you must call tcp\_fasttmr() and tcp\_slowtmr() at the

predefined regular intervals after this initialization.

- netif\_add(struct netif \*netif, ip\_addr\_t \*ipaddr,

ip\_addr\_t \*netmask, ip\_addr\_t \*gw,

void \*state, err\_t (\* init)(struct netif \*netif),

err\_t (\* input)(struct pbuf \*p, struct netif \*netif))

Adds your network interface to the netif\_list. Allocate a struct

netif and pass a pointer to this structure as the first argument.

Give pointers to cleared ip\_addr structures when using DHCP,

or fill them with sane numbers otherwise. The state pointer may be NULL.

The init function pointer must point to a initialization function for

your ethernet netif interface. The following code illustrates it's use.

err\_t netif\_if\_init(struct netif \*netif)

{

u8\_t i;

for(i = 0; i < ETHARP\_HWADDR\_LEN; i++) netif->hwaddr[i] = some\_eth\_addr[i];

init\_my\_eth\_device();

return ERR\_OK;

}

For ethernet drivers, the input function pointer must point to the lwip

function ethernet\_input() declared in "netif/etharp.h". Other drivers

must use ip\_input() declared in "lwip/ip.h".

- netif\_set\_default(struct netif \*netif)

Registers the default network interface.

- netif\_set\_up(struct netif \*netif)

When the netif is fully configured this function must be called.

- dhcp\_start(struct netif \*netif)

Creates a new DHCP client for this interface on the first call.

Note: you must call dhcp\_fine\_tmr() and dhcp\_coarse\_tmr() at

the predefined regular intervals after starting the client.

You can peek in the netif->dhcp struct for the actual DHCP status.

--- Optimalization hints

The first thing you want to optimize is the lwip\_standard\_checksum()

routine from src/core/inet.c. You can override this standard

function with the #define LWIP\_CHKSUM <your\_checksum\_routine>.

There are C examples given in inet.c or you might want to

craft an assembly function for this. RFC1071 is a good

introduction to this subject.

Other significant improvements can be made by supplying

assembly or inline replacements for htons() and htonl()

if you're using a little-endian architecture.

#define LWIP\_PLATFORM\_BYTESWAP 1

#define LWIP\_PLATFORM\_HTONS(x) <your\_htons>

#define LWIP\_PLATFORM\_HTONL(x) <your\_htonl>

Check your network interface driver if it reads at

a higher speed than the maximum wire-speed. If the

hardware isn't serviced frequently and fast enough

buffer overflows are likely to occur.

E.g. when using the cs8900 driver, call cs8900if\_service(ethif)

as frequently as possible. When using an RTOS let the cs8900 interrupt

wake a high priority task that services your driver using a binary

semaphore or event flag. Some drivers might allow additional tuning

to match your application and network.

For a production release it is recommended to set LWIP\_STATS to 0.

Note that speed performance isn't influenced much by simply setting

high values to the memory options.

For more optimization hints take a look at the lwIP wiki.

--- Zero-copy MACs

To achieve zero-copy on transmit, the data passed to the raw API must

remain unchanged until sent. Because the send- (or write-)functions return

when the packets have been enqueued for sending, data must be kept stable

after that, too.

This implies that PBUF\_RAM/PBUF\_POOL pbufs passed to raw-API send functions

must \*not\* be reused by the application unless their ref-count is 1.

For no-copy pbufs (PBUF\_ROM/PBUF\_REF), data must be kept unchanged, too,

but the stack/driver will/must copy PBUF\_REF'ed data when enqueueing, while

PBUF\_ROM-pbufs are just enqueued (as ROM-data is expected to never change).

Also, data passed to tcp\_write without the copy-flag must not be changed!

Therefore, be careful which type of PBUF you use and if you copy TCP data

or not!