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, struct ip_addr *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.

```
- 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, struct ip_addr *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, void *dataptr, u16_t len,
                  u8_t copy)

  Enqueues the data pointed to by the argument dataptr. The length of the data is passed as the len parameter. The copy argument is either 0 or 1 and indicates whether the new memory should be allocated for the data to be copied into. If the argument is 0, no new memory should be allocated and the data should only be referenced by pointer.

  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, struct ip_addr *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, struct ip_addr *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,
                                         struct ip_addr *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, struct ip_addr *ipaddr,
            struct ip_addr *netmask, struct ip_addr *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!