Design and Implementation of the lwIP TCP/IP Stack

Swedish Institute of Computer Science

February 20, 2001

Adam Dunkels

adam@sics.se

Abstract

lwIP is an implementation of the TCP/IP protocol stack. The focus of the lwIP stack is to reduce memory usage and code size, making lwIP suitable for use in small clients with very limited resources such as embedded systems. In order to reduce processing and memory demands, lwIP uses a tailor made API that does not require any data copying.

This report describes the design and implementation of lwIP. The algorithms and data structures used both in the protocol implementations and in the sub systems such as the memory and buffer management systems are described. Also included in this report is a reference manual for the lwIP API and some code examples of using lwIP.

Contents

1 Introduction

Over the last few years, the interest for connecting computers and computer supported devices to wireless networks has steadily increased. Computers are becoming more and more seamlessly integrated with everyday equipment and prices are dropping. At the same time wireless networking technologies, such as Bluetooth [HNI+98] and IEEE 802.11b WLAN [BIG+97], are emerging. This gives rise to many new fascinating scenarios in areas such as health care, safety and security, transportation, and processing industry. Small devices such as sensors can be connected to an existing network infrastructure such as the global Internet, and monitored from anywhere.

The Internet technology has proven itself flexible enough to incorporate the changing network environments of the past few decades. While originally developed for low speed networks such as the ARPANET, the Internet technology today runs over a large spectrum of link technologies with vastly different characteristics in terms of bandwidth and bit error rate. It is highly advantageous to use the existing Internet technology in the wireless networks of tomorrow since a large amount of applications using the Internet technology have been developed. Also, the large connectivity of the global Internet is a strong incentive.

Since small devices such as sensors are often required to be physically small and inexpensive, an implementation of the Internet protocols will have to deal with having limited computing resources and memory. This report describes the design and implementation of a small TCP/IP stack called lwIP that is small enough to be used in minimal systems.

This report is structured as follows. Sections 2, 3, and 4 give an overview of the lwIP stack, Section 5 describes the operating system emulation layer, Section 6 describes the memory and buffer management. Section 7 introduces the network interface abstraction of lwIP, and Sections 8, 9, and 10 describe the implementation of the IP, UDP and TCP protocols. The Sections 11 and 12 describe how to interface with lwIP and introduce the lwIP API. Sections 13 and 14 analyze the implementation. Finally, Section 15 provides a reference manual for the lwIP API and Sections 17 and 18 show various code examples.

2 Protocol layering

The protocols in the TCP/IP suite are designed in a layered fashion, where each protocol layer solves a separate part of the communication problem. This layering can serve as a guide for designing the implementation of the protocols, in that each protocol can be implemented separately from the other. Implementing the protocols in a strictly layered way can however, lead to a situation where the communication overhead between the protocol layers degrades the overall performance [Cla82a]. To overcome these problems, certain internal aspects of a protocol can be made known to other protocols. Care must be taken so that only the important information is shared among the layers.

Most TCP/IP implementations keep a strict division between the application layer and the lower protocol layers, whereas the lower layers can be more or less interleaved. In most operating systems, the lower layer protocols are implemented as a part of the operating system kernel with entry points for communication with the application layer process. The application program is presented with an abstract view of the TCP/IP implementation, where network communication differs only very little from inter-process communication or file I/O. The implications of this is that since the application program is unaware of the buffer mechanisms used by the lower layers, it cannot utilize this information to, e.g., reuse buffers with frequently used data. Also, when the application sends data, this data has to be copied from the application process' memory space into internal buffers before being processed by the network code.

The operating systems used in minimal systems such as the target system of lwIP most often do not maintain a strict protection barrier between the kernel and the application processes. This allows using a more relaxed scheme for communication between the application and the lower layer protocols by the means of shared memory. In particular, the application layer can be made aware of the buffer handling mechanisms used by the lower layers. Therefore, the application can

more efficiently reuse buffers. Also, since the application process can use the same memory as the networking code the application can read and write directly to the internal buffers, thus saving the expense of performing a copy.

3 Overview

As in many other TCP/IP implementations, the layered protocol design has served as a guide for the design of the implementation of lwIP. Each protocol is implemented as its own module, with a few functions acting as entry points into each protocol. Even though the protocols are implemented separately, some layer violations are made, as discussed above, in order to improve performance both in terms of processing speed and memory usage. For example, when verifying the checksum of an incoming TCP segment and when demultiplexing a segment, the source and destination IP addresses of the segment has to be known by the TCP module. Instead of passing these addresses to TCP by the means of a function call, the TCP module is aware of the structure of the IP header, and can therefore extract this information by itself.

lwIP consists of several modules. Apart from the modules implementing the TCP/IP protocols (IP, ICMP, UDP, and TCP) a number of support modules are implemented. The support modules consists of the operating system emulation layer (described in Section 5), the buffer and memory management subsystems (described in Section 6), network interface functions (described in Section 7), and functions for computing the Internet checksum. lwIP also includes an abstract API, which is described in Section 12.

4 Process model

The process model of a protocol implementation describes in which way the system has been divided into different processes. One process model that has been used to implement communication protocols is to let each protocol run as a stand alone process. With this model, a strict protocol layering is enforced, and the communication points between the protocols must be strictly defined. While this approach has its advantages such as protocols can be added at runtime, understanding the code and debugging is generally easier, there are also disadvantages. The strict layering is not, as described earlier, always the best way to implement protocols. Also, and more important, for each layer crossed, a context switch must be made. For an incoming TCP segment this would mean three context switches, from the device driver for the network interface, to the IP process, to the TCP process and finally to the application process. In most operating systems a context switch is fairly expensive.

Another common approach is to let the communication protocols reside in the kernel of the operating system. In the case of a kernel implementation of the communication protocols, the application processes communicate with the protocols through system calls. The communication protocols are not strictly divided from each other but may use the techniques of crossing the protocol layering.

lwIP uses a process model in which all protocols reside in a single process and are thus separated from the operating system kernel. Application programs may either reside in the lwIP process, or be in separate processes. Communication between the TCP/IP stack and the application programs are done either by function calls for the case where the application program shares a process with lwIP, or by the means of a more abstract API.

Having lwIP implemented as a user space process rather than in the operating system kernel has both its advantages and disadvantages. The main advantage of having lwIP as a process is that is portable across different operating systems. Since lwIP is designed to run in small operating systems that generally do not support neither swapping out processes not virtual memory, the delay caused by having to wait for disk activity if part of the lwIP process is swapped or paged out to disk will not be a problem. The problem of having to wait for a scheduling quantum before getting a chance to service requests still is a problem however, but there is nothing in the design

of lwIP that precludes it from later being implemented in an operating system kernel.

5 The operating system emulation layer

In order to make lwIP portable, operating system specific function calls and data structures are not used directly in the code. Instead, when such functions are needed the operating system emulation layer is used. The operating system emulation layer provides a uniform interface to operating system services such as timers, process synchronization, and message passing mechanisms. In principle, when porting lwIP to other operating systems only an implementation of the operating system emulation layer for that particular operating system is needed.

The operating system emulation layer provides a timer functionality that is used by TCP. The timers provided by the operating system emulation layer are one-shot timers with a granularity of at least 200 ms that calls a registered function when the time-out occurs.

The only process synchronization mechanism provided is semaphores. Even if semaphores are not avaliable in the underlying operating system they can be emulated by other synchronization primitives such as conditional variables or locks.

The message passing is done through a simple mechanism which uses an abstraction called *mailboxes*. A mailbox has two operations: post and fetch. The post operation will not block the process; rather, messages posted to a mailbox are queued by the operating system emulation layer until another process fetches them. Even if the underlying operating system does not have native support for the mailbox mechanism, they are easily implemented using semaphores.

6 Buffer and memory management

The memory and buffer management system in a communication system must be prepared to accommodate buffers of very varying sizes, ranging from buffers containing full-sized TCP segments with several hundred bytes worth of data to short ICMP echo replies consisting of only a few bytes. Also, in order to avoid copying it should be possible to let the data content of the buffers reside in memory that is not managed by the networking subsystem, such as application memory or ROM.

6.1 Packet buffers — pbufs

A pbuf is lwIP's internal representation of a packet, and is designed for the special needs of the minimal stack. Pbufs are similar to the mbufs used in the BSD implementations. The pbuf structure has support both for allocating dynamic memory to hold packet contents, and for letting packet data reside in static memory. Pbufs can be linked together in a list, called a pbuf chain so that a packet may span over several pbufs.

Pbufs are of three types, PBUF_RAM, PBUF_ROM, and PBUF_POOL. The pbuf shown in Figure 1 represents the PBUF_RAM type, and has the packet data stored in memory managed by the pbuf subsystem. The pbuf in Figure 2 is an example of a chained pbuf, where the first pbuf in the chain is of the PBUF_RAM type, and the second is of the PBUF_ROM type, which means that it has the data located in memory not managed by the pbuf system. The third type of pbuf, PBUF_POOL, is shown in Figure 3 and consists of fixed size pbufs allocated from a pool of fixed size pbufs. A pbuf chain may consist of multiple types of pbufs.

The three types have different uses. Pbufs of type PBUF_POOL are mainly used by network device drivers since the operation of allocating a single pbuf is fast and is therefore suitable for use in an interrupt handler. PBUF_ROM pbufs are used when an application sends data that is located in memory managed by the application. This data may not be modified after the pbuf has been handed over to the TCP/IP stack and therefore this pbuf type main use is when the data is located in ROM (hence the name PBUF_ROM). Headers that are prepended to the data in a PBUF_ROM pbuf are stored in a PBUF_RAM pbuf that is chained to the front of of the PBUF_ROM pbuf, as in Figure 2.

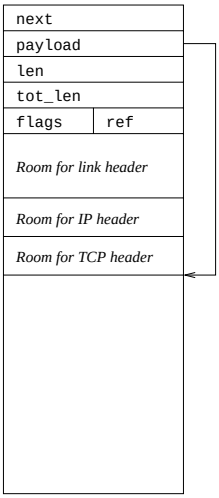

Figure 1. A PBUF_RAM pbuf with data in memory managed by the pbuf subsystem.

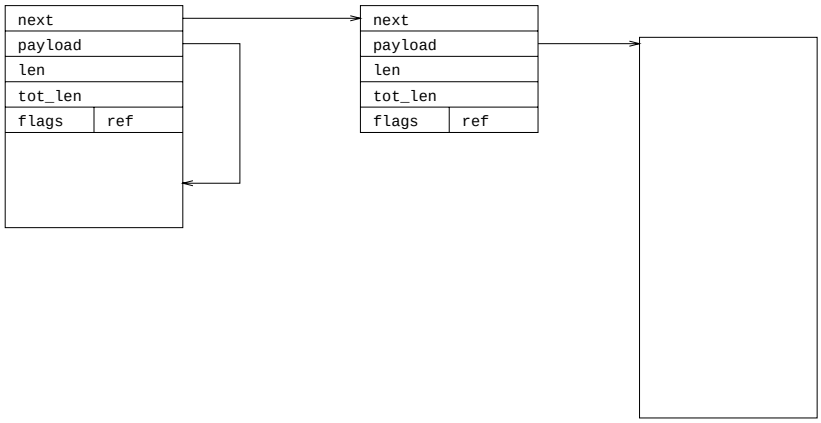

Figure 2. A PBUF_RAM pbuf chained with a PBUF_ROM pbuf that has data in external memory.

Pbufs of the PBUF_RAM type are also used when an application sends data that is dynamically generated. In this case, the pbuf system allocates memory not only for the application data, but also for the headers that will be prepended to the data. This is seen in Figure 1. The pbuf system cannot know in advance what headers will be prepended to the data and assumes the worst case. The size of the headers is configurable at compile time.

In essence, incoming pbufs are of type PBUF_POOL and outgoing pbufs are of the PBUF_ROM or PBUF_RAM types.

The internal structure of a pbuf can be seen in the Figures 1 through 3. The pbuf structure consists of two pointers, two length fields, a flags field, and a reference count. The `next` field is a pointer to the next pbuf in case of a pbuf chain. **The `payload` pointer points to the start of the data in the pbuf. The `len` field contains the length of the data contents of the pbuf. The `tot_len` field contains the sum of the length of the current pbuf and all `len` fields of following pbufs in the pbuf chain.** In other words, the `tot_len` field is the sum of the `len` field and the value of the `tot_len` field in the following pbuf in the pbuf chain. **The `flags` field indicates the type of the pbuf and the `ref` field contains a reference count.** The `next` and `payload` fields are native pointers and the size of those varies depending on the processor architecture used. The two length fields

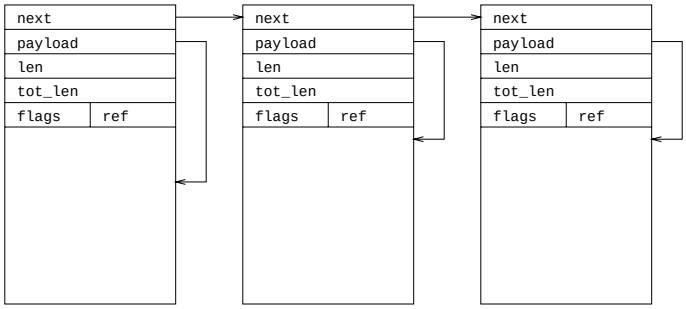

Figure 3. A chained PBUF_POOL pbuf from the pbuf pool.

are 16 bit unsigned integers and the `flags` and `ref` fields are 4 bit wide. The total size of the pbuf structure depends on the size of a pointer in the processor architecture being used and on the smallest alignment possible for the processor architecture. On an architecture with 32 bit pointers and 4 byte alignment, the total size is 16 bytes and on an architecture with 16 bit pointers and 1 byte alignment, the size is 9 bytes.

The pbuf module provides functions for manipulation of pbufs. Allocation of a pbuf is done by the function `pbuf_alloc()` which can allocate pbufs of any of the three types described above. The function `pbuf_ref()` increases the reference count. Deallocation is made by the function `pbuf_free()`, which first decreases the reference count of the pbuf. If the reference count reaches zero the pbuf is deallocated. The function `pbuf_realloc()` shrinks the pbuf so that it occupies just enough memory to cover the size of the data. The function `pbuf_header()` adjusts the `payload` pointer and the length fields so that a header can be prepended to the data in the pbuf. The functions `pbuf_chain()` and `pbuf_dechain()` are used for chaining pbufs.

6.2 Memory management

The memory manager supporting the pbuf scheme is very simple. It handles allocations and deallocations of contiguous regions of memory and can shrink the size of a previously allocated memory block. The memory manager uses a dedicated portion of the total memory in the system. This ensures that the networking system does not use all of the available memory, and that the operation of other programs is not disturbed if the networking system has used all of it's memory.

Internally, the memory manager keeps track of the allocated memory by placing a small structure on top of each allocated memory block. This structure (Figure 4) holds two pointers to the next and previous allocation block in memory. It also has a `used` flag which indicates whether the allocation block is allocated or not.

Memory is allocated by searching the memory for an unused allocation block that is large enough for the requested allocation. The first-fit principle is used so that the first block that is large enough is used. When an allocation block is deallocated, the `used` flag is set to zero. In order to prevent fragmentation, the `used` flag of the next and previous allocation blocks are checked. If any of them are unused, the blocks are combined into one larger unused block.

7 Network interfaces

In lwIP device drivers for physical network hardware are represented by a network interface structure similar to that in BSD. The network interface structure is shown in Figure 5. The network interfaces are kept on a global linked list, which is linked by the `next` pointer in the structure.

Each network interface has a name, stored in the `name` field in Figure 5. This two letter name identifies the kind of device driver used for the network interface and is only used when the

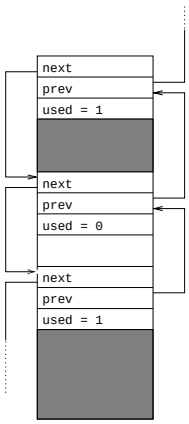

Figure 4. The memory allocation structure.

```
struct netif {
  struct netif *next;
  char name[2];
  int num;
  struct ip_addr ip_addr;
  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;
};
```

Figure 5. The `netif` structure.

interface is configured by a human operator at runtime. The name is set by the device driver and should reflect the kind of hardware that is represented by the network interface. For example, a network interface for a Bluetooth driver might have the name `bt` and a network interface for IEEE 802.11b WLAN hardware could have the name `wl`. Since the names not necessarily are unique, the `num` field is used to distinguish different network interfaces of the same kind.

The three IP addresses `ip_addr`, `netmask` and `gw` are used by the IP layer when sending and receiving packets, and their use is described in the next section. It is not possible to configure a network interface with more than one IP address. Rather, one network interface would have to be created for each IP address.

The `input` pointer points to the function the device driver should call when a packet has been received.

A network interface is connected to a device driver through the `output` pointer. This pointer points to a function in the device driver that transmits a packet on the physical network and it is called by the IP layer when a packet is to be sent. This field is filled by the initialization function of the device driver. The third argument to the `output` function, `ipaddr`, is the IP address of the host that should receive the actual link layer frame. It does not have to be the same as the destination address of the IP packet. In particular, when sending an IP packet to a host that is not on the local network, the link level frame will be sent to a router on the network. In this case, the IP address given to the `output` function will be the IP address of the router.

Finally, the `state` pointer points to device driver specific state for the network interface and

is set by the device driver.

8 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. For most applications this does not pose any problems.

8.1 Receiving packets

For incoming IP packets, processing begins when the `ip_input()` function is called by a network device driver. Here, the initial sanity checking of the IP version field and the header length is done, as well as computing and checking the header checksum. It is expected that the stack will not receive any IP fragments since the proxy is assumed to reassemble any fragmented packets, thus any packet that is an IP fragment is silently discarded. Packets carrying IP options are also assumed to be handled by the proxy, and are dropped.

Next, the function checks the destination address with the IP addresses of the network interfaces to determine if the packet was destined for the host. The network interfaces are ordered in a linked list, and it is searched linearly. The number of network interfaces is expected to be small so a more sophisticated search strategy than a linear search has not been implemented.

If the incoming packet is found to be destined for this host, the protocol field is used to decide to which higher level protocol the packet should be passed to.

8.2 Sending packets

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. When the outgoing network interface is determined, the packet is passed to `ip_output_if()` which takes the outgoing network interface as an argument. Here, all IP header fields are filled in and the IP header checksum is computed. The source and destination addresses of the IP packet is passed as an argument to `ip_output_if()`. The source address may be left out, however, and in this case the IP address of the outgoing network interface is used as the source IP address of the packet.

The `ip_route()` function finds the appropriate network interface by linearly searching the list of network interfaces. During the search the destination IP address of the IP packet is masked with the netmask of the network interface. If the masked destination address is equal to the masked IP address of the interface, the interface is chosen. If no match is found, a default network interface is used. The default network interface is configured either at boot-up or at runtime by a human operator[1]. If the network address of the default interface does not match the destination IP address, the `gw` field in the network interface structure (Figure 5) is chosen as the destination IP address of the link level frame. (Notice that the destination address of the IP packet and the destination address of the link level fram will be different in this case.) This primitive form of routing glosses over the fact that a network might have many routers attached to it. For the most basic case, where a local network only has one router, this works however.

Since the transport layer protocols UDP and TCP need to have the destination IP address when computing the transport layer checksum, the outgoing network interface must in some cases be determined before the packet is passed to the IP layer. This is done by letting the transport layer functions call the `ip_route()` function directly, and since the outgoing network interface is known already when the packet reaches the IP layer, there is no need to search the network interface list again. Instead, those protocols call the `ip_output_if()` function directly. Since this function takes a network interface as an argument, the search for an outgoing interface is avoided.

[1] Human configuration of lwIP during runtime requires an application program that is able to configure the stack. Such a program is not included in lwIP.

8.3 Forwarding packets

If none of the network interfaces has the same IP address as an incoming packet's destination address, the packet should be forwarded. This is done by the function `ip_forward()`. Here, the TTL field is decreased and if it reaches zero, an ICMP error message is sent to the original sender of the IP packet and the packet is discarded. Since the IP header is changed, the IP header checksum needs to be adjusted. The is no need to recompute the entire checksum, however, since simple arithmetic can be used to adjust the original IP checksum [MK90, Rij94]. Finally, the packet is forwarded to the appropriate network interface. The algorithm used to find the appropriate network interface is the same that is used when sending IP packets.

8.4 ICMP processing

ICMP processing is fairly simple. ICMP packets received by `ip_input()` are handed over to `icmp_input()`, which decodes the ICMP header and takes the appropriate action. Some ICMP messages are passed to upper layer protocols and those are taken care of by special functions in the transport layer. ICMP destination unreachable messages can be sent by transport layer protocols, in particular by UDP, and the function `icmp_dest_unreach()` is used for this.

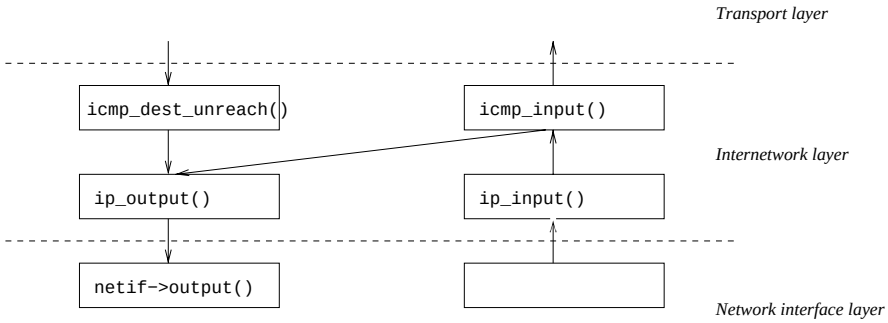

Figure 6. ICMP processing

Using ICMP ECHO messages to probe a network is widely used, and therefore ICMP echo processing is optimized for performance. The actual processing takes place in `icmp_input()`, and consists of swapping the IP destination and source addresses of the incoming packet, change the ICMP type to echo reply and adjust the ICMP checksum. The packet is then passed back to the IP layer for transmission.

9 UDP processing

UDP is a simple protocol used for demultiplexing packets between different processes. The state for each UDP session is kept in a PCB structure as shown in Figure 7. The UDP PCBs are kept on a linked list which is searched for a match when a UDP datagram arrives.

The UDP PCB structure contains a pointer to the next PCB in the global linked list of UDP PCBs. A UDP session is defined by the IP addresses and port numbers of the end-points and these are stored in the `local_ip`, `dest_ip`, `local_port` and `dest_port` fields. The `flags` field indicates what UDP checksum policy that should be used for this session. This can be either to switch UDP checksumming off completely, or to use UDP Lite [LDP99] in which the checksum covers only parts of the datagram. If UDP Lite is used, the `chksum_len` field specifies how much of the datagram that should be checksummed.

The last two arguments `recv` and `recv_arg` are used when a datagram is received in the session specified by the PCB. The function pointed to by `recv` is called when a datagram is received.

```
struct udp_pcb {
  struct udp_pcb *next;
  struct ip_addr local_ip, dest_ip;
  u16_t local_port, dest_port;
  u8_t flags;
  u16_t chksum_len;
  void (* recv)(void *arg, struct udp_pcb *pcb, struct pbuf *p);
  void *recv_arg;
};
```

Figure 7. The `udp_pcb` structure

Due to the simplicity of UDP, the input and output processing is equally simple and follows a fairly straight line (Figure 8). To send data, the application program calls `udp_send()` which calls upon `udp_output()`. Here the the necessary checksumming is done and UDP header fields are filled. Since the checksum includes the IP source address of the IP packet, the function `ip_route()` is in some cases called to find the network interface to which the packet is to be transmitted. The IP address of this network interface is used as the source IP address of the packet. Finally, the packet is turned over to `ip_output_if()` for transmission.

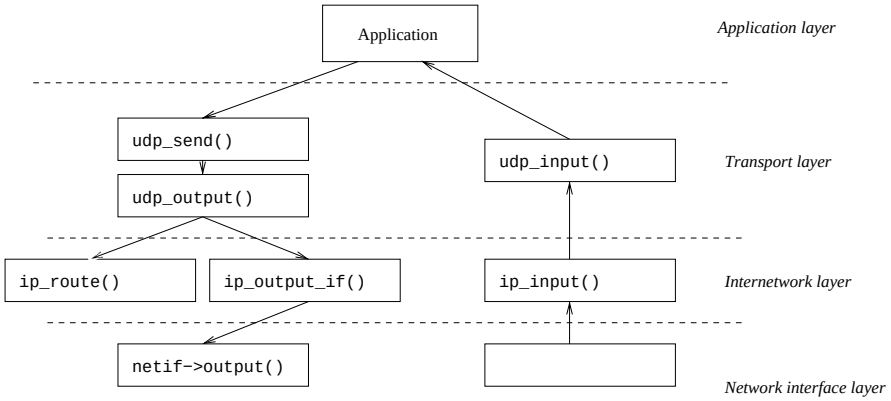

Figure 8. UDP processing

When a UDP datagram arrives, the IP layer calls the `udp_input()` function. Here, if checksumming should be used in the session, the UDP checksum is checked and the datagram is demultiplexed. When the corresponding UDP PCB is found, the `recv` function is called.

10 TCP processing

TCP is a transport layer protocol that provides a reliable byte stream service to the application layer. TCP is more complex than the other protocols described here, and the TCP code constitutes 50% of the total code size of lwIP.

10.1 Overview

The basic TCP processing (Figure 9) is divided into six functions; the functions `tcp_input()`, `tcp_process()`, and `tcp_receive()` which are related to TCP input processing, and `tcp_write()`, `tcp_enqueue()`, and `tcp_output()` which deals with output processing.

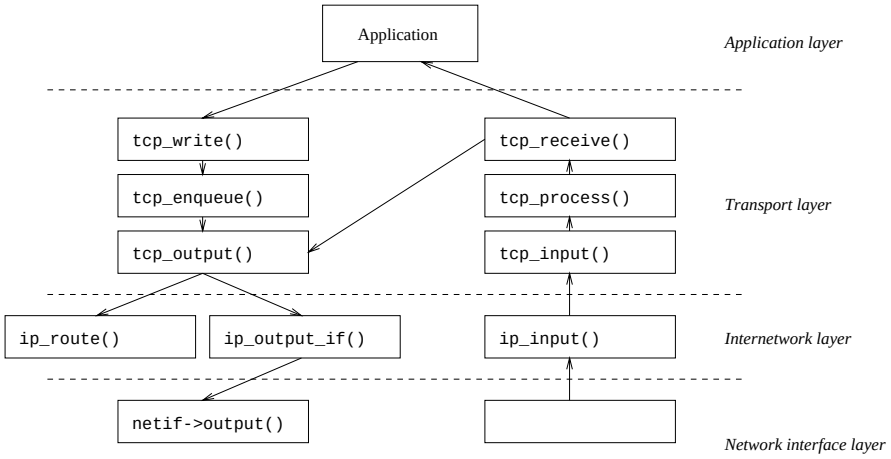

Figure 9. TCP processing

When an application wants to send TCP data, `tcp_write()` is called. The function `tcp_write()` passes control to `tcp_enqueue()` which will break the data into appropriate sized TCP segments if necessary 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()`.

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()`, which implements the TCP state machine, and any necessary state transitions are made. 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 the segment constitutes an ACK for unacknowledged (thus previously buffered) data, the data is removed from the buffers and its memory is reclaimed. Also, if an ACK for data was received the receiver might be willing to accept more data and therefore `tcp_output()` is called.

10.2 Data structures

The data structures used in the implementation of TCP are kept small due to the memory constraints in the minimal system for which lwIP is intended. There is a tradeoff between the complexity of the data structures and the complexity of the code that uses the data structures, and in this case the code complexity has been sacrificed in order to keep the size of the data structures small.

The TCP PCB is fairly large and is shown in Figure 10. Since TCP connections in the LISTEN and TIME-WAIT needs to keep less state information than connections in other states, a smaller PCB data structure is used for those connections. This data structure is overlaid with the full PCB structure, and the ordering of the items in the PCB structure in Figure 10 is therefore somewhat awkward.

The TCP PCBs are kept on a linked list, and the `next` pointer links the PCB list together. The `state` variable contains the current TCP state of the connection. Next, the IP addresses and port numbers which identify the connection are stored. The `mss` variable contains the maximum segment size allowed for the connection.

The `rcv_nxt` and `rcv_wnd` fields are used when receiving data. The `rcv_nxt` field contains the next sequence number expected from the remote end and is thus used when sending ACKs

to the remote host. The receiver's window is kept in `rcv_wnd` and this is advertised in outgoing TCP segments. The field `tmr` is used as a timer for connections that should be removed after a certain amount of time, such as connections in the TIME-WAIT state. The maximum segment size allowed on the connection is stored in the `mss` field. The `flags` field contains additional state information of the connection, such as whether the connection is in fast recovery or if a delayed ACK should be sent.

```
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;
  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;
};
```

Figure 10. The `tcp_pcb` structure

The fields `rttest`, `rtseq`, `sa`, and `sv` are used for the round-trip time estimation. The sequence number of the segment that is used for estimating the round-trip time is stored in `rtseq` and the time this segment was sent is stored in `rttest`. The average round-trip time and the round-trip time variance is stored in `sa` and `sv`. These variables are used when calculating the retransmission time-out which is stored in the `rto` field.

The two fields `lastack` and `dupacks` are used in the implementation of fast retransmit and fast recovery. The `lastack` field contains the sequence number acknowledged by the last ACK received and `dupacks` contains a count of how many ACKs that has been received for the sequence number in `lastack`. The current congestion window for the connection is stored in the `cwnd` field and the slow start threshold is kept in `ssthresh`.

The six fields `snd_ack`, `snd_nxt`, `snd_wnd`, `snd_wl1`, `snd_wl2` and `snd_lbb` are used when sending data. The highest sequence number acknowledged by the receiver is stored in `snd_ack` and the next sequence number to send is kept in `snd_nxt`. The receiver's advertised window is held in `snd_wnd` and the two fields `snd_wl1` and `snd_wl2` are used when updating `snd_wnd`. The `snd_lbb` field contains the sequence number of the last byte queued for transmission.

The function pointer `recv` and `recv_arg` are used when passing received data to the application layer. The three queues `unsent`, `unacked` and `ooseq` are used when sending and receiving data. 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`.

```
struct tcp_seg {
  struct tcp_seg *next;
  u16_t len;
  struct pbuf *p;
  struct tcp_hdr *tcphdr;
  void *data;
  u16_t rtime;
};
```

Figure 11. The `tcp_seg` structure

The `tcp_seg` structure in Figure 11 is the internal representation of a TCP segment. This structure starts with a `next` pointer which is used for linking when queuing segments. The `len` field contains the length of the segment in TCP terms. This means that the `len` field for a data segment will contain the length of the data in the segment, and the `len` field for an empty segment with the SYN or FIN flags set will be 1. The pbuf `p` is the buffer containing the actual segment and the `tcphdr` and `data` pointers points to the TCP header and the data in the segment, respectively. For outgoing segments, the `rtime` field is used for the retransmission time-out of this segment. Since incoming segments will not need to be retransmitted, this field is not needed and memory for this field is not allocated for incoming segments.

10.3 Sequence number calculations

The TCP sequence numbers that are used to enumerate the bytes in the TCP byte stream are unsigned 32 bit quantities, hence in the range $[0, 2^{32} - 1]$. Since the number of bytes sent in a TCP connection might be more than the number of 32-bit combinations, the sequence numbers are calculated modulo 2^{32}. This means that ordinary comparison operators cannot be used with TCP sequence numbers. The modified comparison operators, called $<_{seq}$ and $>_{seq}$, are defined by the relations

$$s <_{seq} t \Leftrightarrow s - t < 0$$

and

$$s >_{seq} t \Leftrightarrow s - t > 0,$$

where s and t are TCP sequence numbers. The comparison operators for $\leq$ and $\geq$ are defined equivalently. The comparison operators are defined as C macros in the header file.

10.4 Queuing and transmitting data

Data that is to be sent is divided into appropriate sized chunks and given sequence numbers by the `tcp_enqueue()` function. Here, the data is packeted into pbufs and enclosed in a `tcp_seg` structure. The TCP header is build in the pbuf, and filled in with all fields except the acknowledgment number, `ackno`, and the advertised window, `wnd`. These fields can change during the queuing time of the segment and are therefore set by `tcp_output()` which does the actual transmission of the segment. After the segments are built, they are queued on the `unsent` list in the PCB. The `tcp_enqueue()` function tries to fill each segment with a maximum segment size worth of data and when an under-full segment is found at the end of the `unsent` queue, this segment is appended with the new data using the pbuf chaining functionality.

After `tcp_enqueue()` has formatted and queued the segments, `tcp_output()` is called. It checks if there is any room in the current window for any more data. The current window is computed by taking the maximum of the congestion window and the advertised receiver's window. Next, it fills in the fields of the TCP header that was not filled in by `tcp_enqueue()` and transmits the segment using `ip_route()` and `ip_output_if()`. After transmission the segment is put on the `unacked` list, on which it stays until an ACK for the segment has been received.

When a segment is on the `unacked` list, it is also timed for retransmission as described in Section 10.8. When a segment is retransmitted the TCP and IP headers of the original segment is kept and only very little changes has to be made to the TCP header. The `ackno` and `wnd` fields of the TCP header are set to the current values since we could have received data during the time between the original transmission of the segment and the retransmission. This changes only two 16-bit words in the header and the whole TCP checksum does not have to be recomputed since simple arithmetic [Rij94] can be used to update the checksum. The IP layer has already added the IP header when the segment was originally transmitted and there is no reason to change it. Thus a retransmission does not require any recomputation of the IP header checksum.

10.4.1 Silly window avoidance

The Silly Window Syndrome [Cla82b] (SWS) is a TCP phenomena that can lead to very bad performance. SWS occurs when a TCP receiver advertises a small window and the TCP sender immediately sends data to fill the window. When this small segment is acknowledged the window is opened again by a small amount and sender will again send a small segment to fill the window. This leads to a situation where the TCP stream consists of very small segments. In order to avoid SWS both the sender and the receiver must try to avoid this situation. The receiver must not advertise small window updates and the sender must not send small segments when only a small window is offered.

In lwIP SWS is naturally avoided at the sender since TCP segments are constructed and queued without knowledge of the advertised receiver's window. In a large transfer the output queue will consist of maximum sized segments. This means that if a TCP receiver advertises a small window, the sender will not send the first segment on the queue since it is larger than the advertised window. Instead, it will wait until the window is large enough for a maximum sized segment.

When acting as a TCP receiver, lwIP will not advertise a receiver's window that is smaller than the maximum segment size of the connection.

10.5 Receiving segments

10.5.1 Demultiplexing

When TCP segments arrive at the `tcp_input()` function, they are demultiplexed between the TCP PCBs. The demultiplexing key is the source and destination IP addresses and the TCP port numbers. There are two types of PCBs that must be distinguished when demultiplexing a segment; those that correspond to open connections and those that correspond to connections that are half open. Half open connections are those that are in the LISTEN state and only have the local TCP port number specified and optionally the local IP address, whereas open connections have the both IP addresses and both port numbers specified.

Many TCP implementations, such as the early BSD implementations, use a technique where a linked list of PCBs with a single entry cache is used. The rationale behind this is that most TCP connections constitute bulk transfers which typically show a large amount locality [Mog92], resulting in a high cache hit ratio. Other caching schemes include keeping two one entry caches, one for the PCB corresponding to the last packet that was sent and one for the PCB of the last packet received [PP93]. An alternative scheme to exploit locality can be done by moving the most recently used PCB to the front of the list. Both methods have been shown [MD92] to outperform the one entry cache scheme.

In lwIP, whenever a PCB match is found when demultiplexing a segment, the PCB is moved to the front of the list of PCBs. PCBs for connections in the LISTEN state are not moved to the front however, since such connections are not expected to receive segments as often as connections that are in a state in which they receive data.

10.5.2 Receiving data

The actual processing of incoming segments is made in the function `tcp_receive()`. The acknowledgment number of the segment is compared with the segments on the `unacked` queue of the connection. If the acknowledgment number is higher than the sequence number of a segment on the `unacked` queue, that segment is removed from the queue and the allocated memory for the segment is deallocated.

An incoming segment is out of sequence if the sequence number of the segment is higher than the `rcv_nxt` variable in the PCB. Out of sequence segments are queued on the `ooseq` queue in the PCB. If the sequence number of the incoming segment is equal to `rcv_nxt`, the segment is delivered to the upper layer by calling the `recv` function in the PCB and `rcv_nxt` is increased by the length of the incoming segment. Since the reception of an in-sequence segment might mean that a previously received out of sequence segment now is the next segment expected, the `ooseq` queued is checked. If it contains a segment with sequence number equal to `rcv_nxt`, this segment is delivered to the application by a call to to `recv` function and `rcv_nxt` is updated. This process continues until either the `ooseq` queue is empty or the next segment on `ooseq` is out of sequence.

10.6 Accepting new connections

TCP connections that are in the LISTEN state, i.e., that are passively open, are ready to accept new connections from a remote host. For those connections a new TCP PCB is created and must be passed to the application program that opened the initial listening TCP connection. In lwIP this is done by letting the application register a callback function that is to be called when a new connection has been established.

When a connection in the LISTEN state receives a TCP segment with the SYN flag set, a new connection is created and a segment with the SYN and ACK flags are sent in response to the SYN segment. The connection then enters the SYN-RCVD state and waits for an acknowledgment for the sent SYN segment. When the acknowledgment arrives, the connection enters the ESTABLISHED state, and the accept function (the `accept` field in the PCB structure in Figure 10) is called.

10.7 Fast retransmit

Fast retransmit and fast recovery is implemented in lwIP by keeping track of the last sequence number acknowledged. If another acknowledgment for the same sequence number is received, the `dupacks` counter in the TCP PCB is increased. When `dupacks` reaches three, the first segment on the `unacked` queue is retransmitted and fast recovery is initialized. The implementation of fast recovery follows the steps laid out in [APS99]. Whenever an ACK for new data is received, the `dupacks` counter is reset to zero.

10.8 Timers

As in the the BSD TCP implementation, lwIP uses two periodical timers that goes off every 200 ms and 500 ms. Those two timers are then used to implement more complex logical timers such as the retransmission timers, the TIME-WAIT timer and the delayed ACK timer.

The fine grained timer, `tcp_timer_fine()` goes through every TCP PCB checking if there are any delayed ACKs that should be sent, as indicated by the `flag` field in the `tcp_pcb` structure (Figure 10). If the delayed ACK flag is set, an empty TCP acknowledgment segment is sent and the flag is cleared.

The coarse grained timer, implemented in `tcp_timer_coarse()`, also scans the PCB list. For every PCB, the list of unacknowledged segments (the `unacked` pointer in the `tcp_seg` structure in Figure 11), is traversed, and the `rtime` variable is increased. If `rtime` becomes larger than the current retransmission time-out as given by the `rto` variable in the PCB, the segment is retransmitted and the retransmission time-out is doubled. A segment is retransmitted only if allowed by the values of the congestion window and the advertised receiver's window. After retransmission, the congestion window is set to one maximum segment size, the slow start threshold is set to half of the effective window size, and slow start is initiated on the connection.

For connections that are in TIME-WAIT, the coarse grained timer also increases the `tmr` field in the PCB structure. When this timer reaches the $2 \times MSL$ threshold, the connection is removed.

The coarse grained timer also increases a global TCP clock, `tcp_ticks`. This clock is used for round-trip time estimation and retransmission time-outs.

10.9 Round-trip time estimation

The round-trip time estimation is a critical part of TCP since the estimated round-trip time is used when determining a suitable retransmission time-out. In lwIP round-trip times measurements are taken in a fashion similar to the BSD implementations. Round-trip times are measured once per round-trip and the smoothing function described in [Jac88] is used for the calculation of a suitable retransmission time-out.

The TCP PCB variable `rtseq` hold the sequence number of the segment for which the round-trip time is measured. The `rttest` variable in the PCB holds the value of `tcp_ticks` when the segment was first transmitted. When an ACK for a sequence number equal to or larger than `rtseq` is received, the round-trip time is measured by subtracting `rttest` from `tcp_ticks`. If a retransmission occurred during the round-trip time measurement, no measurement is taken.

10.10 Congestion control

The implementation of congestion control is surprisingly simple and consists of a few lines of code in the output and input code. When an ACK for new data is received the congestion window, `cwnd`, is increased either by one maximum segment size or by $mss^2/cwnd$, depending on whether the connection is in slow start or congestion avoidance. When sending data the minimum value of the receiver's advertised window and the congestion window is used to determine how much data that can be sent in each window.

11 Interfacing the stack

There are two ways for using the services provided by the TCP/IP stack; either by calling the functions in the TCP and UDP modules directly, or to use the lwIP API presented in the next section.

The TCP and UDP modules provide a rudimentary interface to the networking services. The interface is based on callbacks and an application program that uses the interface can therefore not operate in a sequential manner. This makes the application program harder to program and the application code is harder to understand. In order to receive data the application program registers a callback function with the stack. The callback function is associated with a particular connection and when a packet arrives in the connection, the callback function is called by the stack.

Furthermore, an application program that interfaces the TCP and UDP modules directly has to (at least partially) reside in the same process as the TCP/IP stack. This is due to the fact that a callback function cannot be called across a process boundary. This has both advantages and disadvantages. One advantage is that since the application program and the TCP/IP stack are in the same process, no context switching will be made when sending or receiving packets. The main disadvantage is that the application program cannot involve itself in any long running

computations since TCP/IP processing cannot occur in parallel with the computation, thus degrading communication performance. This can be overcome by splitting the application into two parts, one part dealing with the communication and one part dealing with the computation. The part doing the communication would then reside in the TCP/IP process and the computationally heavy part would be a separate process. The lwIP API presented in the next section provides a structured way to divide the application in such a way.

12 Application Program Interface

Due to the high level of abstraction provided by the BSD socket API, it is unsuitable for use in a minimal TCP/IP implementation. In particular, BSD sockets require data that is to be sent to be copied from the application program to internal buffers in the TCP/IP stack. The reason for copying the data is that the application and the TCP/IP stack usually reside in different protection domains. In most cases the application program is a user process and the TCP/IP stack resides in the operating system kernel. By avoiding the extra copy, the performance of the API can be greatly improved [ABM95]. Also, in order to make a copy, extra memory needs to be allocated for the copy, effectively doubling the amount of memory used per packet.

The lwIP API was designed for lwIP and utilizes knowledge of the internal structure of lwIP to achieve effectiveness. The lwIP API is very similar to the BSD API, but operates at a slightly lower level. The API does not require that data is copied between the application program and the TCP/IP stack, since the application program can manipulate the internal buffers directly.

Since the BSD socket API is well understood and many application programs have been written for it, it is advantageous to have a BSD socket compatibility layer. Section 17 presents the BSD socket functions rewritten using the lwIP API. A reference manual of the lwIP API is found in Section 15.

12.1 Basic concepts

From the application's point of view, data handling in the BSD socket API is done in continuous memory regions. This is convenient for the application programmer since manipulation of data in application programs is usually done in such continuous memory chunks. Using this type of mechanism with lwIP would not be advantageous, since lwIP usually handles data in buffers where the data is partitioned into smaller chunks of memory. Thus the data would have to be copied into a continuous memory area before being passed to the application. This would waste both processing time and memory since. Therefore, the lwIP API allows the application program to manipulate data directly in the partitioned buffers in order to avoid the extra copy.

The lwIP API uses a connection abstraction similar to that of the BSD socket API. There are very noticeable differences however; where an application program using the BSD socket API need not be aware of the distinction between an ordinary file and a network connection, an application program using the lwIP API has to be aware of the fact that it is using a network connection.

Network data is received in the form of buffers where the data is partitioned into smaller chunks of memory. Since many applications wants to manipulate data in a continuous memory region, a convenience function for copying the data from a fragmented buffer to continuous memory exists.

Sending data is done differently depending on whether the data should be sent over a TCP connection or as UDP datagrams. For TCP, data is sent by passing the output function a pointer to a continuous memory region. The TCP/IP stack will partition the data into appropriately sized packets and queue them for transmission. When sending UDP datagrams, the application program will to explicitly allocate a buffer and fill it with data. The TCP/IP stack will send the datagram immediately when the output function is called.

12.2 Implementation of the API

The implementation of the API is divided into two parts, due to the process model of the TCP/IP stack. As shown in Figure 12, parts of the API is implemented as a library linked to the application program, and parts are implemented in the TCP/IP process. The two parts communicate using the interprocess communication (IPC) mechanisms provided by the operating system emulation layer. The current implementation uses the following three IPC mechanisms:

- shared memory,
- message passing, and
- semaphores.

While these IPC types are supported by the operating system layer, they need not be directly supported by the underlying operating system. For operating systems that do not natively support them, the operating system emulation layer emulates them.

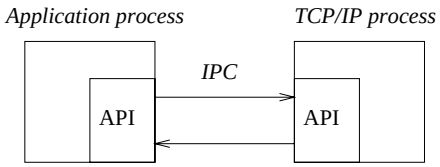

Figure 12. Division of the API implementation

The general design principle used is to let as much work as possible be done within the application process rather than in the TCP/IP process. This is important since all processes use the TCP/IP process for their TCP/IP communication. Keeping down the code footprint of the part of the API that is linked with the applications is not as important. This code can be shared among the processes, and even if shared libraries are not supported by the operating system, the code is stored in ROM. Embedded systems usually carry fairly large amounts of ROM, whereas processing power is scarce.

The buffer management is located in the library part of the API implementation. Buffers are created, copied and deallocated in the application process. Shared memory is used to pass the buffers between the application process and the TCP/IP process. The buffer data type used in communication with the application program is an abstraction of the pbuf data type.

Buffers carrying referenced memory, as opposed to allocated memory, is also passed using shared memory. For this to work, is has to be possible to share the referenced memory between the processes. The operating systems used in embedded systems for which lwIP is intended usually do not implement any form of memory protection, so this will not be a problem.

The functions that handle network connections are implemented in the part of the API implementation that resides in the TCP/IP process. The API functions in the part of the API that runs in the application process will pass a message using a simple communication protocol to the API implementation in the TCP/IP process. The message includes the type of operation that should be carried out and any arguments for the operation. The operation is carried out by the API implementation in the TCP/IP process and the return value is sent to the application process by message passing.

13 Statistical code analysis

This section analyzes the code of lwIP with respect to compiled object code size and number of lines in the source code. The code has been compiled for two processor architectures:

- The Intel Pentium III processor, henceforth referred to as the Intel x86 processor. The code was compiled with gcc 2.95.2 under FreeBSD 4.1 with compiler optimizations turned on.
- The 6502 processor [Nab, Zak83]. The code was compiled with cc65 2.5.5 [vB] with compiler optimizations turned on.

The Intel x86 has seven 32-bit registers and uses 32-bit pointers. The 6502, which main use today is in embedded systems, has one 8-bit accumulator as well as two 8-bit index registers and uses 16-bit pointers.

13.1 Lines of code

Table 1. Lines of code.

Module	Lines of code	Relative size
TCP	1076	42%
Support functions	554	21%
API	523	20%
IP	189	7%
UDP	149	6%
ICMP	87	3%
Total	2578	100%

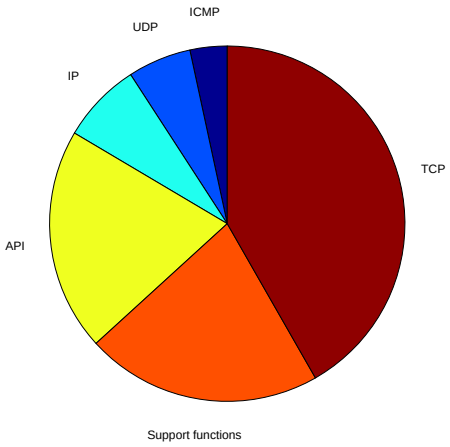

Figure 13. Lines of code.

Table 1 summarizes the number lines of source code of lwIP and Figure 13 shows the relative number of lines of code. The category "Support functions" include buffer and memory management functions as well as the functions for computing the Internet checksum. The checksumming functions are generic C implementations of the algorithm that should be replaced with processor specific implementations when actually deployed. The category "API" includes both the part of the API that is linked with the applications and the part that is linked with the TCP/IP stack. The operating system emulation layer is not included in this analysis since its size varies heavily with the underlying operating system and is therefore not interesting to compare.

For the purpose of this comparison all comments and blank lines have been removed from the source files. Also, no header files were included in the comparison. We see that TCP is vastly larger than the other protocol implementations and that the API and the support functions taken together are as large as TCP.

13.2 Object code size

Table 2. lwIP object code size when compiled for the Intel x86.

Module	Size (bytes)	Relative size
TCP	6584	48%
API	2556	18%
Support functions	2281	16%
IP	1173	8%
UDP	731	5%
ICMP	505	4%
Total	13830	100%

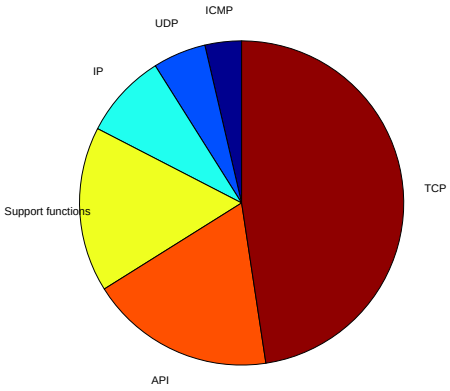

Figure 14. lwIP object code size when compiled for the x86.

Table 2 summarizes the sizes of the compiled object code when compiled for the Intel x86 and Figure 14 shows the relative sizes. We see that the order of the items are somewhat different from Table 1. Here, the API is larger than the support functions category, even though the support functions has more lines of code. We also see that TCP constitutes 48% of the compiled code but only 42% of the total lines of code. Inspection of the assembler output from the TCP module shows a probable cause for this. The TCP module involve large amounts of pointer dereferencing, which expands into many lines of assembler code, thus increasing the object code size. Since many pointers are dereferenced two or three times in each function, this could be optimized by modifying the source code so that pointers are dereferenced only once and placed in a local variable. While this would reduce the size of the compiled code, it would also use more RAM as space for the local variables is allocated on the stack.

Table 3. lwIP object code size when compiled for the 6502.

Module	Size (bytes)	Relative size
TCP	11461	51%
Support functions	4149	18%
API	3847	17%
IP	1264	6%
UDP	1211	5%
ICMP	714	3%
Total	22646	100%

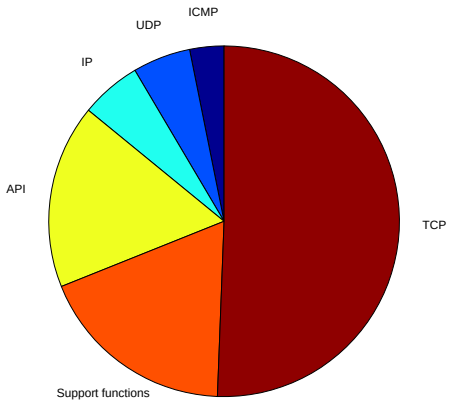

Figure 15. lwIP object code size when compiled for the 6502.

Table 3 shows the sizes of the object code when compiled for the 6502 and in Figure 14 the relative sizes are shown. We see that the TCP, the API, and the support functions are nearly twice as large as when compiled for the Intel x86, whereas IP, UDP and ICMP are approximately the same size. We also see that the support functions category is larger than the API, contrary to Table 2. The difference in size between the API and the support functions category is small though.

The reason for the increase in size of the TCP module is that the 6502 does not natively support 32-bit integers. Therefore, each 32-bit operation is expanded by the compiler into many lines of assembler code. TCP sequence numbers are 32-bit integers and the TCP module performs numerous sequence number computations.

The size of the TCP code can be compared to the size of TCP in other TCP/IP stacks, such as the popular BSD TCP/IP stack for FreeBSD 4.1 and the independently derived TCP/IP stack for Linux 2.2.10. Both are compiled for the Intel x86 with gcc and compiler optimizations turned on. The size of the TCP implementation in lwIP is almost 6600 bytes. The object code size of the TCP implementation in FreeBSD 4.1 is roughly 27000 bytes, which is four times as large as in lwIP. In Linux 2.2.10, the object code size of the TCP implementation is even larger and consists of 39000 bytes, roughly six times as much as in lwIP. The large difference in code size between lwIP and the two other implementations arise from the fact that both the FreeBSD and the Linux implementations contain more TCP features such as SACK [MMFR96] as well as parts of the implementation of the BSD socket API.

The reason for not comparing the sizes of the implementation of IP is that there is vastly more features in the IP implementations of FreeBSD and Linux. For instance, both FreeBSD and Linux includes support for firewalling and tunneling in their IP implementations. Also, those implementations support dynamic routing tables, which is not implemented in lwIP.

The lwIP API constitutes roughly one sixth of the size of lwIP. Since lwIP can be used without inclusion of the API, this part can be left out when deploying lwIP in a system with very little code memory.

14 Performance analysis

The performance of lwIP in terms of RAM usage or code efficiency have not been formally tested in this thesis, and this has been noted in future work. Simple tests have been conducted, however, and these have shown that lwIP running a simple HTTP/1.0 server is able to serve at least 10 simultaneous requests for web pages while consuming less than 4 kilobytes of RAM. In those tests, only the memory used by the protocols, buffering system, and the application program has been taken into account. Thus memory used by a device driver would add to the above figure.

15 API reference

15.1 Data types

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

- `netbuf`, the network buffer abstraction, and
- `netconn`, the abstraction of a network connection.

Each data type is repressented as a pointer to a C `struct`. Knowledge of the internal structure of the `struct` should not be used in application programs. Instead, the API provides functions for modifying and extracting necessary fields.

15.1.1 Netbufs

Netbufs are buffers that are used for sending and receiving data. Internally, a netbuf is associated with a pbuf as presented in Section 6.1. Netbufs can, just as pbufs, accomodate both allocated memory and referenced memory. Allocated memory is RAM that is explicitly allocated for holding network data, whereas referenced memory might be either application managed RAM or external ROM. Referenced memory is useful for sending data that is not modified, such as static web pages or images.

The data in a netbuf can be fragmented into differenly sized blocks. This means that an application must be prepared to accept fragmented data. Internally, a netbuf has a pointer to one of the fragments in the netbuf. Two functions, `netbuf_next()` and `netbuf_first()` are used to manipulate this pointer.

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.

15.2 Buffer functions

15.2.1 netbuf_new()

Synopsis

```
struct netbuf * netbuf_new(void)
```

Description
Allocates a netbuf structure. No buffer space is allocated when doing this, only the top level structure. After use, the netbuf must be deallocated with `netbuf_delete()`.

15.2.2 netbuf_delete()

Synopsis

```
void netbuf_delete(struct netbuf *)
```

Description
Deallocates a netbuf structure previosly allocated by a call to the `netbuf_new()` function. Any buffer memory allocated to the netbuf by calls to `netbuf_alloc()` is also deallocated.

Example This example shows the basic mechanisms for using netbufs.

```
int
main()
{
  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 */
}
```

15.2.3 netbuf_alloc()

Synopsis

```
void * netbuf_alloc(struct netbuf *buf, int size)
```

Description
Allocates buffer memory with `size` number of bytes for the netbuf `buf`. The function returns a pointer to the allocated memory. Any memory previously allocated to the netbuf `buf` is deallocated. The allocated memory can later be deallocated with the `netbuf_free()` function. Since protocol headers are expected to precede the data when it should be sent, the function allocates memory for protocol headers as well as for the actual data.

15.2.4 netbuf_free()

Synopsis

```
int netbuf_free(struct netbuf *buf)
```

Description
Deallocates the buffer memory associated with the netbuf `buf`. If no buffer memory has been allocated for the netbuf, this function does nothing.

15.2.5 netbuf_ref()

Synopsis

```
int netbuf_ref(struct netbuf *buf, void *data, int size)
```

Description
Associates the external memory pointer to by the `data` pointer with the netbuf `buf`. The size of the external memory is given by `size`. Any memory previously allocated to the netbuf is deallocated. The difference between allocating memory for the netbuf with `netbuf_alloc()` and allocating memory using, e.g., `malloc()` and referencing it with `netbuf_ref()` is that in the former case, space for protocol headers is allocated as well which makes processing and sending the buffer faster.

Example This example shows a simple use of the `netbuf_ref()` function.

```
int
main()
{
  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 */
  /* [...] */

  /* deallocate netbuf */
  netbuf_delete(buf);
}
```

15.2.6 netbuf_len()

Synopsis

```
int netbuf_len(struct netbuf *buf)
```

Description
Returns the total length of the data in the netbuf `buf`, even if the netbuf is fragmented. For a fragmented netbuf, the value obtained by calling this function is not the same as the size of the first fragment in the netbuf.

15.2.7 netbuf_data()

Synopsis

```
int netbuf_data(struct netbuf *buf, void **data, int *len)
```

Description
This function is used to obtain a pointer to and the length of a block of data in the netbuf `buf`. The arguments `data` and `len` are result parameters that will be filled with a pointer to the data and the length of the data pointed to. If the netbuf is fragmented, this function gives a pointer to one of the fragments in the netbuf. The application program must use the fragment handling functions `netbuf_first()` and `netbuf_next()` in order to reach all data in the netbuf.

See the example under `netbuf_next()` for an example of how use `netbuf_data()`.

15.2.8 netbuf_next()

Synopsis

```
int netbuf_next(struct netbuf *buf)
```

Description
This function updates the internal fragment pointer in the netbuf `buf` so that it points to the next fragment in the netbuf. The return value is zero if there are more fragments in the netbuf, > 0 if the fragment pointer now points to the last fragment in the netbuf, and < 0 if the fragment pointer already pointed to the last fragment.

Example This example shows how to use the `netbuf_next()` function. We assume that this is in the middle of a function and that the variable `buf` is a netbuf.

```
  /* [...] */
  do {
     char *data;
     int len;
```

```
    /* obtain a pointer to the data in the fragment */
    netbuf_data(buf, &data, &len);

    /* do something with the data */
    do_something(data, len);
  } while(netbuf_next(buf) >= 0);
  /* [...] */
```

15.2.9 netbuf_first()

Synopsis

```
void netbuf_first(struct netbuf *buf)
```

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

15.2.10 netbuf_copy()

Synopsis

```
void netbuf_copy(struct netbuf *buf, void *data, int len)
```

Description
Copies all of the data from the netbuf `buf` into the memory pointed to by `data` even if the netbuf `buf` is fragmented. The `len` parameter is an upper bound of how much data that will be copied into the memory pointed to by `data`.

Example This example shows a simple use of `netbuf_copy()`. Here, 200 bytes of memory is allocated on the stack to hold data. Even if the netbuf `buf` has more data that 200 bytes, only 200 bytes are copied into `data`.

```
void
example_function(struct netbuf *buf)
{
   char data[200];
   netbuf_copy(buf, data, 200);

   /* do something with the data */
}
```

15.2.11 netbuf_chain()

Synopsis

```
void netbuf_chain(struct netbuf *head, struct netbuf *tail)
```

Description
Chains the two netbufs `head` and `tail` together so that the data in `tail` will become the last fragment(s) in `head`. The netbuf `tail` is deallocated and should not be used after the call to this function.

15.2.12 netbuf_fromaddr()

Synopsis

```
struct ip_addr * netbuf_fromaddr(struct netbuf *buf)
```

Description
Returns the IP address of the host the netbuf `buf` was received from. If the netbuf has not been received from the network, the return the value of this function is undefined. The function `netbuf_fromport()` can be used to obtain the port number of the remote host.

15.2.13 netbuf_fromport()

Synopsis
```
unsigned short netbuf_fromport(struct netbuf *buf)
```

Description
Returns the port number of the host the netbuf `buf` was received from. If the netbuf has not been received from the network, the return the value of this function is undefined. The function `netbuf_fromaddr()` can be used to obtain the IP address of the remote host.

16 Network connection functions

16.0.14 netconn_new()

Synopsis
```
struct netconn * netconn_new(enum netconn_type type)
```

Description
Creates a new connection abstraction structure. The argument can be one of `NETCONN_TCP` or `NETCONN_UDP`, yielding either a TCP or a UDP connection. No connection is established by the call to this function and no data is sent over the network.

16.0.15 netconn_delete()

Synopsis
```
void netconn_delete(struct netconn *conn)
```

Description
Deallocates the netconn `conn`. If the connection is open, it is closed as a result of this call.

16.0.16 netconn_type()

Synopsis
```
enum netconn_type netconn_type(struct netconn *conn)
```

Description
Returns the type of the connection `conn`. This is the same type that is given as an argument to `netconn_new()` and can be either `NETCONN_TCP` or `NETCONN_UDP`.

16.0.17 netconn_peer()

Synopsis
```
int netconn_peer(struct netconn *conn,
struct ip_addr **addr, unsigned short port)
```

Description
The function `netconn_peer()` is used to obtain the IP address and port of the remote end of a connection. The parameters `addr` and `port` are result parameters that are set by the function. If the connection `conn` is not connected to any remote host, the results are undefined.

16.0.18 netconn_addr()

Synopsis
```
int netconn_addr(struct netconn *conn,
struct ip_addr **addr, unsigned short port)
```

Description
This function is used to obtain the local IP address and port number of the connection `conn`.

16.0.19 netconn_bind()

Synopsis
```
int netconn_bind(struct netconn *conn,
struct ip_addr *addr, unsigned short port)
```

Description
Binds the connection `conn` to the local IP address `addr` and TCP or UDP port `port`. If `addr` is `NULL` the local IP address is determined by the networking system.

16.0.20 netconn_connect()

Synopsis
```
int netconn_connect(struct netconn *conn,
struct ip_addr *remote_addr, unsigned short remote_port)
```

Description
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.

16.0.21 netconn_listen()

Synopsis
```
int netconn_listen(struct netconn *conn)
```

Description
Puts the TCP connection `conn` into the TCP LISTEN state.

16.0.22 netconn_accept()

Synopsis
```
struct netconn * netconn_accept(struct netconn *conn)
```

Description
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 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);
}
```

16.0.23 netconn_recv()

Synopsis

```
struct netbuf * netconn_recv(struct netconn *conn)
```

Description
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 recevied data is returned.

Example This is a small example that shows a suggested use of the `netconn_recv()` function. We assume that a connection has been established before the call to `example_function()`.

```
void
example_function(struct netconn *conn)
{
   struct netbuf *buf;

   /* receive data until the other host closes
      the connection */
   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);
}
```

16.0.24 netconn_write()

Synopsis

```
int netconn_write(struct netconn *conn, void *data,
int len, unsigned int flags)
```

Description
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 length of the data is given by `len`. There is no restriction on the length of the data. This function does not require the application to explicitly allocate buffers, as this is taken care of by the stack. The `flags` parameter has two possible states, as shown below.

```
#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. This allows the data to be modified directly after the call, but 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. This is useful when sending data that is located in ROM and therefore is immutable.

If greater control over the modifiability of the data is needed, a combination of copied and non-copied data can be used, as seen in the example below.

Example This example shows the basic usage of `netconn_write()`. Here, the variable `data` is assumed to be modified later in the program, and is therefore copied into the internal buffers by passing the flag `NETCONN_COPY` to `netconn_write()`. The `text` variable contains a string that will not be modified and can therefore be sent using references instead of copying.

```
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);
}
```

16.0.25 netconn_send()

Synopsis

```
int netconn_send(struct netconn *conn, struct netbuf *buf)
```

Description
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. The data should not be larger than the maximum transmission unit (MTU) of the outgoing network interface. Since there currently is no way of obtaining this value a careful approach sould be taken, and the netbuf should not contain data that is larger than some 1000 bytes.

No checking is made whether the data is sufficiently small and sending very large netbufs might give undefined results.

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);
}
```

16.0.26 netconn_close()

Synopsis

```
int netconn_close(struct netconn *conn)
```

Description
Closes the connection `conn`.

17 BSD socket library

This section provides a simple implementation of the BSD socket API using the lwIP API. The implementation is provided as a reference only, and is not intended for use in actual programs. There is for example no error handling.

Also, this implementation does not support the `select()` and `poll()` functions of the BSD socket API since the lwIP API does not have any functions that can be used to implement those. In order to implement those functions, the BSD socket implementation would have to communicate directly with the lwIP stack and not use the API.

17.1 The representation of a socket

In the BSD socket API sockets are represented as ordinary file descriptors. File descriptors are integers that uniquely identifies the file or network connection. In this implementation of the BSD socket API, sockets are internally represented by a `netconn` structure. Since BSD sockets are identified by an integer, the `netconn` variables are kept in an array, `sockets[]`, where the BSD socket identifier is the index into the array.

17.2 Allocating a socket

17.2.1 The `socket()` call

The `socket()` call allocates a BSD socket. The parameters to `socket()` are used to specify what type of socket that is requested. Since this socket API implementation is concerned only with network sockets, these are the only socket type that is supported. Also, 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;
}
```

17.3 Connection setup

The BSD socket API calls for setting up a connection are very similar to the connection setup functions of the minimal API. The implementation of these calls mainly include translation from the integer representation of a socket to the connection abstraction used in the minimal API.

17.3.1 The `bind()` call

The `bind()` call binds the BSD socket to a local address. In the call to `bind()` the local IP address and port number are specified. The `bind()` function is very similar to the `netconn_bind()` function in the lwIP API.

```
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;
}
```

17.3.2 The `connect()` call

The implementation of `connect()` is as straightforward as that of `bind()`.

```
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;
}
```

17.3.3 The `listen()` call

The `listen()` call is the equivalent of the LWIP API function `netconn_listen()` and can only be used for TCP connections. The only difference is that the BSD socket API allows the application to specify the size of the queue of pending connections (the backlog). This is not possible with LWIP and the `backlog` parameter is ignored.

```
int
listen(int s, int backlog)
{
  netconn_listen(sockets[s]);
  return 0;
}
```

17.3.4 The `accept()` call

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. The arguments to listen are result parameters that are set by the call to `accept()`. These are filled with the address of the remote host.

When the new connection has been established, the LWIP function `netconn_accept()` will return the connection handle for the new connection. After the IP address and port number of the remote host has been filled in, a new socket identifier is allocated and returned.

```
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;
}
```

17.4 Sending and receiving data

17.4.1 The `send()` call

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()`. For UDP sessions, the `send()` call resembles the `netconn_send()` function from the lwIP API, but since the lwIP API require the application to explicitly allocate buffers, a buffer must be allocated and deallocated within the `send()` call. Therefore, a buffer is allocated and the data is copied into the allocated buffer.

The `netconn_send()` function of the lwIP API cannot be used with TCP connections, so this implementation of the `send()` uses `netconn_write()` for TCP connections. In the BSD socket API, the application is allowed to modify the sent data directly after the call to `send()` and therefore the `NETCONN_COPY` flag is passed to `netconn_write()` so that the data is copied into internal buffers in the stack.

```
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;
}
```

17.4.2 The `sendto()` and `sendmsg()` calls

The `sendto()` and `sendmsg()` calls 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. Also, `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. An implementation of `sendmsg()` is not included.

```
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);
}
```

17.4.3 The `write()` call

In the BSD socket API, 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;
}
```

17.4.4 The `recv()` and `read()` calls

In the BSD socket API, the `recv()` and `read()` calls are used on a connected socket to receive data. They can be used for both TCP and UDP connections. A number of flags can be passed by the call to `recv()`. None of these are implemented here, and the `flags` parameter is ignored.

If the received message is larger than the supplied memory area, the excess data is silently discarded.

```
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);
}
```

17.4.5 The `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.

An implementation of `recvmsg()` is not included.

```
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;
  }
}
```

18 Code examples

18.1 Using the API

This section presents a simple web server written using the lwIP API. The application code is given below. The application implements only the bare bones of an HTTP/1.0 [BLFF96] server and is included only to show the principles in using the lwIP API for an actual application.

The application consists of a single process that accepts connections from the network, responds to HTTP requests, and closes the connection. There are two functions in the application; `main()`

which does the necessary initialization and connection setup, and `process_connection()` that implements the small subset of HTTP/1.0. The connection setup procedure a is fairly straightforward example of how connections are initialized using the minimal API. After the connection has been created using `netconn_new()` the connection is bound to TCP port 80 and put into the LISTEN state, in which it waits for connections. The call to `netconn_accept()` will return a `netconn` connection once a remote host has connected. After the connection has been processed by `process_connection()` the `netconn` must be deallocated using `netconn_delete()`.

In `process_connection()` a netbuf is received through a call to `netconn_recv()` and a pointer to the actual request data is obtained via `netbuf_data()`. This will return the pointer to the data in the first fragment of the netbuf, and we hope that it will contain the request. Since we only read the first seven bytes of the request, this is not an unreasonable assumption. If we would have wanted to read more data, the simplest way would have been to use `netbuf_copy()` and copy the request into a continuous memory and process it from there.

This simple web server only responds to HTTP GET requests for the file "/", and when the request has been checked the response it sent. We send the HTTP header for HTML data as well as the HTML data with two calls to the functions `netconn_write()`. Since we do not modify either the HTTP header or the HTML data, we can use the `NETCONN_NOCOPY` flag with `netconn_write()` thus avoiding any copying.

Finally, the connection is closed and the function `process_connection()` returns. The connection structure is deallocated after the call.

The C code for the application follows.

```
/* A simple HTTP/1.0 server using the minimal API. */

#include "api.h"

/* This is the data for the actual web page.
   Most compilers would place this in ROM. */
const static char indexdata[] =
"<html> \
<head><title>A test page</title></head> \
<body> \
This is a small test page. \
</body> \
</html>";

const static char http_html_hdr[] =
"Content-type: text/html\r\n\r\n";

/* This function processes an incomming connection. */
static void
process_connection(struct netconn *conn)
{
  struct netbuf *inbuf;
  char *rq;
  int len;

  /* Read data from the connection into the netbuf inbuf.
     We assume that the full request is in the netbuf. */
  inbuf = netconn_recv(conn);

  /* Get the pointer to the data in the first netbuf
     fragment which we hope contains the request. */
```

```
  netbuf_data(inbuf, &rq, &len);

  /* Check if the request was an HTTP "GET /\r\n". */
  if(rq[0] == 'G' && rq[1] == 'E' &&
     rq[2] == 'T' && rq[3] == ' ' &&
     rq[4] == '/' && rq[5] == '\r' &&
       rq[6] == '\n') {

      /* Send the header. */
      netconn_write(conn, http_html_hdr, sizeof(http_html_hdr),
                    NETCONN_NOCOPY);

      /* Send the actual web page. */
      netconn_write(conn, indexdata, sizeof(indexdata),
                    NETCONN_NOCOPY);

      /* Close the connection. */
      netconn_close(conn);
    }
  }
}


/* The main() function. */
int
main()
{
  struct netconn *conn, *newconn;

  /* Create a new TCP connection handle. */
  conn = netconn_new(NETCONN_TCP);

  /* Bind the connection to port 80 on any
     local IP address. */
  netconn_bind(conn, NULL, 80);

  /* Put the connection into LISTEN state. */
  netconn_listen(conn);

  /* Loop forever. */
  while(1) {
    /* Accept a new connection. */
    newconn = netconn_accept(conn);

    /* Process the incomming connection. */
    process_connection(newconn);

    /* Deallocate connection handle. */
    netconn_delete(newconn);
  }
  return 0;
}
```

18.2 Directly interfacing the stack

Since the basic web server machanism is very simple in that it only receives one request and services it by sending a file to the remote host, it is well suited to be implemented using the internal event based interface of the stack. Also, since there is no heavy computation involved, the TCP/IP processing is not delayed. The following example shows how to implement such an application. The implementation of the application is very similar to the above example.

```
/* A simple HTTP/1.0 server direcly interfacing the stack. */

#include "tcp.h"

/* This is the data for the actual web page. */
static char indexdata[] =
"HTTP/1.0 200 OK\r\n\
Content-type: text/html\r\n\
\r\n\
<html> \
<head><title>A test page</title></head> \
<body> \
This is a small test page. \
</body> \
</html>";

/* This is the callback function that is called
   when a TCP segment has arrived in the connection. */
static void
http_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p)
{
  char *rq;

  /* If we got a NULL pbuf in p, the remote end has closed
     the connection. */
  if(p != NULL) {

    /* The payload pointer in the pbuf contains the data
       in the TCP segment. */
    rq = p->payload;

    /* Check if the request was an HTTP "GET /\r\n". */
    if(rq[0] == 'G' && rq[1] == 'E' &&
       rq[2] == 'T' && rq[3] == ' ' &&
       rq[4] == '/' && rq[5] == '\r' &&
       rq[6] == '\n') {

      /* Send the web page to the remote host. A zero
         in the last argument means that the data should
         not be copied into internal buffers. */
      tcp_write(pcb, indexdata, sizeof(indexdata), 0);
    }

    /* Free the pbuf. */
    pbuf_free(p);
  }
```

```
  /* Close the connection. */
  tcp_close(pcb);
}

/* This is the callback function that is called when
   a connection has been accepted. */
static void
http_accept(void *arg, struct tcp_pcb *pcb)
{
  /* Set up the function http_recv() to be called when data
     arrives. */
  tcp_recv(pcb, http_recv, NULL);
}

/* The initialization function. */
void
http_init(void)
{
  struct tcp_pcb *pcb;

  /* Create a new TCP PCB. */
  pcb = tcp_pcb_new();

  /* Bind the PCB to TCP port 80. */
  tcp_bind(pcb, NULL, 80);

  /* Change TCP state to LISTEN. */
  tcp_listen(pcb);

  /* Set up http_accet() function to be called
     when a new connection arrives. */
  tcp_accept(pcb, http_accept, NULL);
}
```

References

[ABM95] B. Ahlgren, M. Björkman, and K. Moldeklev. The performance of a no-copy api for communication (extended abstract). In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Mystic, Connecticut, USA, August 1995.

[APS99] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, Internet Engineering Task Force, April 1999.

[BIG+97] C. Brian, P. Indra, W. Geun, J. Prescott, and T. Sakai. IEEE-802.11 wireless local area networks. *IEEE Communications Magazine*, 35(9):116–126, September 1997.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. RFC 1945, Internet Engineering Task Force, May 1996.

[Cla82a] D. D. Clark. Modularity and efficiency in protocol implementation. RFC 817, Internet Engineering Task Force, July 1982.

[Cla82b] D. D. Clark. Window and acknowledgement strategy in TCP. RFC 813, Internet Engineering Task Force, July 1982.

[HNI+98] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, and W. Allen. Bluetooth: Vision, goals, and architecture. *Mobile Computing and Communications Review*, 2(4):38–45, October 1998.

[Jac88] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Conference*, Stanford, California, August 1988.

[LDP99] L. Larzon, M. Degermark, and S. Pink. UDP Lite for real-time multimedia applications. In *Proceedings of the IEEE International Conference of Communications*, Vancouver, British Columbia, Canada, June 1999.

[MD92] Paul E. McKenney and Ken F. Dove. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–279, Baltimore, Maryland, August 1992.

[MK90] T. Mallory and A. Kullberg. Incremental updating of the internet checksum. RFC 1141, Internet Engineering Task Force, January 1990.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. RFC 2018, Internet Engineering Task Force, October 1996.

[Mog92] J. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.

[Nab] M. Naberezny. The 6502 microprocessor resource. Web page. 2000-11-30.
URL: *http://www.6502.org/*

[PP93] C. Partridge and S. Pink. A faster UDP. *IEEE/ACM Transactions in Networking*, 1(4):429–439, August 1993.

[Rij94] A. Rijsinghani. Computation of the internet checksum via incremental update. RFC 1624, Internet Engineering Task Force, May 1994.

[vB] U. von Bassewitz. cc65 - a freeware c compiler for 6502 based systems. Web page. 2000-11-30.
URL: *http://www.cc65.org/*

[Zak83] R. Zaks. *Programming the 6502*. Sybex, Berkeley, California, 1983.