Design and Implementation of the lwIP TCP/IP Stack

Lwip 协议栈的设计与实现

（中文版）

Swedish Institute of Computer Science

February 20, 2001

作者：Adam Dunkels adam@sics.se

翻译：果农(QQ: 10205001)

核桃(QQ: 329147)

佳旭(QQ:3232253)

整理：佳旭(QQ:3232253)

摘要

LWIP是TCP/IP协议栈的一种实现。LWIP的主要目的是减少存储器利用量和代码尺寸，使LWIP适合应用于小的、资源有限的处理器如嵌入式系统。为了减少处理器和存储器要求，lwIP可以通过不需任何数据拷贝的API进行裁减。

本文叙述了lwIP的设计与实现。叙述了协议实现及子系统中所使用的算法和数据结构如存储和缓冲管理系统。还包括LWIP API的参考手册和使用LWIP 的一些代码例子。

目录

1 序论

在过去的几年里，人们对计算机互连和计算机无线互连支持设备的兴趣不断的增长，计算机逐渐与日常使用的设备无缝结合，并且价格不断下降。同时，无线网络技术如Bluetooth [HNI+98] 和 IEEE 802.11b WLAN [BIG+97]不断显现。这也在一些领域譬如医疗保健、安全保卫、交通运输、加工业等引起了许多新引人入胜的情节。小的设备如传感器能被联入现有的网络如全球因特网，并可以在任何地方对其进行监控。

在过去的几年里，互联网技术证明自己具有足够的灵活性来合并不断改变的网络的环境。与当初为低速网络譬如ARPANET网而产生的互联网相比，今天的大范围连接的互联网技术在带宽和误码率方面都与原来有着巨大的差异。由于互联网的大量应用，把将来的无线互连网络应用于现有的互连网络将会给我们带来巨大的收益。并且，大面积互连的互联网也是一强劲趋势。

自从人们经常对像传感器这样的小设备有小的物理外形和便宜的价格的要求，实现一较少的处理和存储要求的互连协议就成为必须解决的问题。本文描述了一种称为LWIP的小到足以满足最小系统要求的TCP/IP协议栈的设计与实现。

本文结构如下编排：第2，3和4部分对lwIP栈作一个概述，第5部分叙述操作系统模拟层，第6部分叙述缓存和存储管理。第7部分介绍lwIP抽象的网络接口，第8，9，和10部分叙述IP，UDP，和TCP协议的实现。第11和12部分叙述怎样与lwIP进行接口并介绍lwIP API。第13和14部分分析了实现过程。最后，15部分提供了lwIP API用户参考手册，17和18部分展示了多种代码例子。

2 协议分层（Protocol layering）

TCP/IP协议被设计为分层结构，各协议层分别解决通信问题的一部份。这一分层对于协议的设计、实现可起一个指导作用，各个协议可分开实现。然而协议严格的按分层结构来实现，各层之间的通讯可能会导致总体性能的降低 [[Cla82a] 。 为克服这些问题，协议的某些内部方面可传达给其它协议共享，但必须注意，保证只有那些重要信息才在各层共享。

尽管底层协议或多或少可以进行交叉存取，大部分TCP/IP协议，还是在应用层协议与底层协议之间进行严格的区分。在大部分操作系统中，底层协议被作为与应用层程序具有通讯接口的操作系统内核的一部分。应用程序被看作是TCP/IP协议的抽象，网络通讯与进程间通讯或者文件I/O只有很小的差别。这意味着，因为应用程序不知道被底层协议所使用的缓冲机制，它不能利用缓冲机制对经常使用的数据进行缓冲。同样，当应用程序发送数据时，在数据被网络代码处理前，必须把这些数据从应用程序存储区被拷贝到内部缓冲区。

最小系统中使用的操作系统像lwIP的目标系统在内核和应用进程之间常常并不存在严格的保护屏障。这就允许应用程序和底层协议之间使用一种更宽松的方案，通过共享内存。特别地，应用层可以意识到底层协议所使用的缓存处理机制。因此，应用可以更有效地重用缓冲区。而且，

既然应用进程和网络代码可以使用相同的内存，应用可以直接读写内部缓存，因此节省了执行拷贝的开销。

3 总述（Overview）

正如其他TCP/IP协议的实现，分层协议的设计为LWIP的设计与实现提供一向导。每一个协议都作为一个模块来实现，提供一些与其他协议的接口函数。尽管各层分开实现，但正如上面所讨论的，为了同时提高处理速度和内存利用两方面的性能，一些层在设计时违背这一原则。例如：当检验一接收到的TCP段（segment）的校验和（checksum）和分解TCP段时，源和目的IP地址必须被告知TCP模块。LWIP实现时不是通过函数调用把IP地址传递给TCP，而是TCP模块通过获取IP报头的结构进而自己提取这一信息。

LWIP有几个模块组成，除了实现TCP/IP协议的各个模块(IP、ICMP、UDP、和 TCP)，同时设计了许多支持模块。这些支持模块组成了操作系统模拟层（第5章）、缓冲和存储管理子系统（第6章）、网络接口函数（第7章）和一些处理因特网校验和的函数。LWIP还包括关于API的摘要（第12章）。

4 进程模型（Process model）

协议实现的过程模型以把系统划分成为不同的过程的方法进行描述。用于实现通讯协议的过程模型使每个协议作为孤立的过程运行。这种模型使用严格的协议分层，协议之间的通讯结点必须被严格定义。虽然这种方法有其诸多优势如协议能在运行时被增加，代码一般容易理解和调试，但也有不利因素。 严格的分层，正如先前所述，并不总是实现协议的最好方法。 同时，更重要的，每跨越一层，必须做一次上下文切换。这将意味着，接受一个TCP段要进行三次上下文切换：从网络接口的驱动，到IP处理，再到TCP处理，最终到应用处理。根据网络接口的设备驱动程序，对于IP过程，对于TCP过程和最后。 在大多数操作系统中一个上下文切换所花的代价都是相当昂贵的。

另一个较普通的方法是把通信协议封装在操作系统的内核。 在这种内核实现通讯协议的情况下，应用程序通过系统调用完成通讯。 通讯协议之间不严格区分，但可以使用交叉协议分层技术。

lwIP所使用的过程模型是：把所以协议封装到一个单一的过程中，从而与操作系统内核分开。应用程序可能也驻留在lwIP处理过程中，或者在单独的过程中。 TCP/IP栈和应用程序之间的通信可以通过函数调用实现，也可以通过更为抽象的API。

以上两种LWIP的实现方法各有其优缺点。把LWIP作为一个过程的主要优点是便于在不同的操作系统上移植。由于LWIP的设计目标是面向小的操作系统，这些操作系统一般不支持进程外交换（swapping out processes）或者虚拟存储，这样由于LWIP处理过程交换或者翻页到磁盘而引起的不得不等待磁盘响应造成的延迟将不再是一个问题。尽管在获得服务响应前必须等待调度仍然是一个问题，但是，在LWIP设计时，这并没有妨碍它在一操作系统内核中实现。

5 操作系统模拟层

为了使lwIP便于移植，与操作系统有关的功能函数调用和数据结构没有在代码中直接使用。而是当需要这样的函数时，操作系统模拟层将加以使用。操作系统模拟层向诸如定时器、处理同步、消息传送机制等的操作系统服务提供一套统一的接口。原则上，移植lwIP到其他操作系统时，仅仅需要实现适合该操作系统的操作系统模拟层。

操作系统模拟层提供了由TCP使用的定时器功能。操作系统模拟层提供的定时器是一次性的定时器，当超时发生时，调用一个已注册函数至少要200ms的间隔。

进程同步机制仅提供了信号量。即使在操作系统底层中信号量不可用，也可以通过其他信号原语像条件变量或互锁来模拟。

信息传递的实现使用一种简单机制，用一种称为“邮箱”的抽象方法。邮箱做两种操作：邮寄和提取。邮寄操作不会阻塞进程；邮寄到邮箱的消息由操作系统模拟层排入队列直到另一个进程来提取它们。即使操作系统底层对邮箱机制不支持，也容易用信号量实现。

6 缓冲和存储管理

通讯系统中的存储和缓冲管理必须能够适应大小变化的缓冲区，从几百字节的包含完全大小TCP段的缓冲区到仅仅包含几个字节的短的ICMP回报。而且，为了避免拷贝它应当尽可能让缓冲区的数据内容驻留在内存中，网络子系统不管理像应用存储或ROM这样的内存。

6.1包缓冲器－pbufs

Pbuf在lwIP的内部表示一包，也是为了最小限度的使用栈这一特殊需要而设计。 Pbufs类似于用于BSD实现的mbufs。 pbuf结构既支持分配动态内存来保存包内容，也支持把包数据存储在静态存储区。Pbufs能在一张列表中一起被连在一起，称为一个pbuf链，这样一个包可以跨越若干个pbufs。

Pbufs具有三种类型，PBUF RAM，PBUF ROM，和PBUF POOL。 图1中pbuf描绘了PBUF RAM类型，和储存的被pbuf子系统管理的数据包。 图2中的pbuf是被链在一起的pbuf的一个例子，在其中链的第一个pbuf具有PBUF RAM类型（where the first pbufin the chain is of the PBUF RAM type），而第二个具有PBUF ROM类型，这意味着它具有不被pbuf系统管理的存储数据。第三种类型的pbuf ，PBUF POOL如图3所示，包括从共有的固定大小的pbufs分配的固定大小的pbufs （consists of fixed size pbufs allocated from a pool of fixedsize pbufs.）。 一个pbuf链可能包括多重类型的pbufs。

三种类型有不同的使用。 PBUF_POOL主要被网络设备驱动程序使用，因为对操作系统来说分配单一的pbuf速度较快并且适合用于中断管理（suitable foruse in an interrupt handler）。 当应用程序发送位于被应用程序管理的存储区的数据时，PBUF_ROM被使用。 在pbuf被移交到TCP/IP栈后，数据不能修改，因此这一pbuf类型，这类型主要用于数据位于ROM时(因此名称为PBUF_ROM)。 PBUF_ROM pbuf中的数据可能会用到的头存储在PBUF_RAM pbuf中，它链接在PBUF_ROM pbuf的前面，如图2所示。

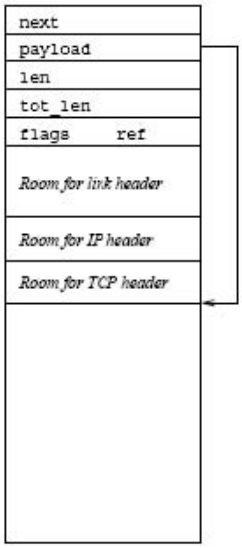


Figure 1. A PBUF_RAM pbuf with data in memory managed by the pbuf subsystem.

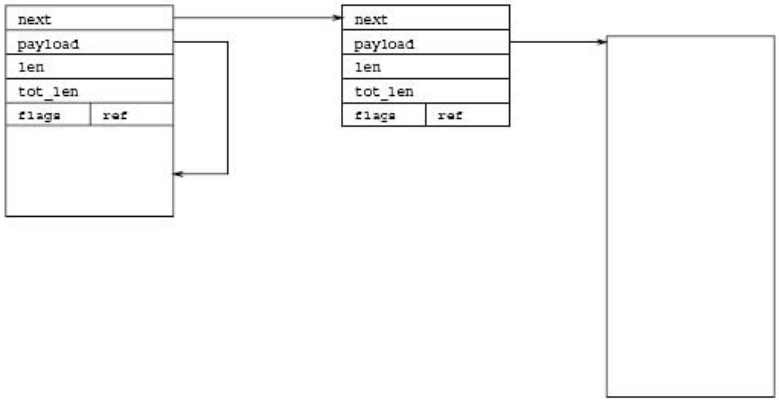


Figure 2. A PBUF_RAM pbuf chained with a PBUF_ROM pbuf that has data in external memory.

当应用程序发送动态地产生的数据时，PBUF_RAM类型的Pbufs也被使用。 在这种情况下，pbuf系统不光为应用数据分配存储空间，也为将要发送的数据的报头准备空间。如图1所示。pbuf系统不能预先知道为将要发送的数据准备什么样的报头，并且假定最坏的情况。报头大小在编译时可动态配置。

实质上,进入pbufs的是PBUF_POOL类型，离开pbufs的是PBUF_ROM或PBUF_RAM类型。

pbuf的内部的结构如图1~3。 pbuf结构包括两个指针，两长度域，一个flags域，和一参考计数。pbuf链中next域是一个指向下一个pbuf的指针。Payload指针指向pbuf中的数据的起始位置。len域包含pbuf的数据内容的长度。Tot_len域包含当前的pbuf的长度和在pbuf链中接下来的pbufs的所有len领域的总数。换句话说，tot_len域是len域和pbuf链中的随后的pbuf中的tot_len域的值的总和。flags域表明pbuf的类型，而ref领域包含一参考计数。 Next和payload域是内部指针和依赖于处理器体系结构的数据大小。两个长度域为16位无符号整形，flags和ref域均为4bit宽。pbuf结构整个的大小取决于所使用的处理器体系结构中一个指针的大小及可能的最小alignment的大小。 在带有32位指针和4个字节alignment的体系结构，整个的大小为16字节，16位指针和1个字节alignment的体系结构上，大小是9个字节。

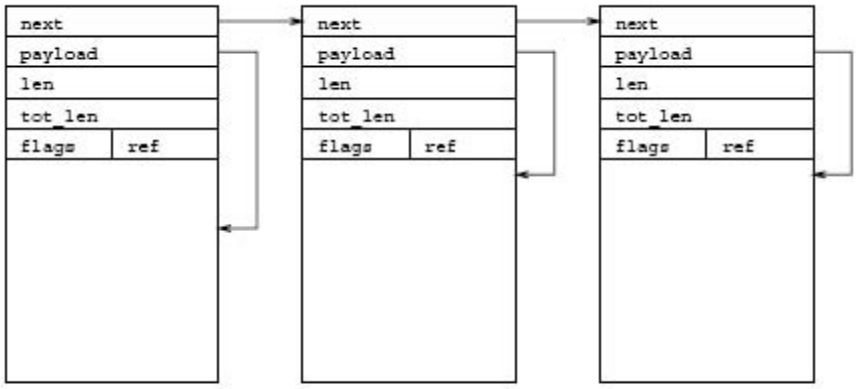


Figure 3. A chained PBUF_POOL pbuf from the pbuf pool.

pbuf模块为操纵pbufs提供了函数。函数pbuf_alloc() 完成分配一个pbuf的任务，它能够分配上面所说的三种pbuf中的任何一种。pbuf_ref()增加参考计数。 pbuf_free()完成释放分配的工作，它首先减少pbuf的参考计数。 如果参考计数到达零表示pbuf已经被释放。 函数pbuf_realloc()收缩pbuf使它刚好能够包含数据大小。 pbuf_header()调整payload 指针和长

度域，以便对pbuf中的数据的报头进行预先估计。 buf_chain() 和pbuf_dechain()用于用链接pbufs。

6.2内存管理

支持pbuf调度的存储管理非常简单。它处理内存中连续区域的分配和释放，可以紧缩一个预先分配的内存块。内存管理器使用系统中总内存的专用部分，这确保网络系统不会使用所有可利用内存，而且如果网络系统用了所有它自己的内存其他程序的操作也不会影响它。

在内部，内存管理通过将一种小的结构放置在每一被分配的内存块的顶端上来追踪分配的内存。这个结构（图4）中设置两个指针指向内存中下一个和前一个分配块，还有一个used标志用来指示这个分配块是否已经被分配。

通过搜索一个未使用的内存块来分配内存，这个内存块对于请求分配来说足够大。使用最先适用原则，因此第一块被使用的内存足够大。当一个分配块释放时，used标志被设为0。为了防止碎片，检测下一个和上一个分配块的used标志，如果它们还没被使用，几个块合并成一个大未使用块。

7网络接口

硬件设备驱动程序中，lwIP用一个类似于BSD的网络接口结构来描述物理硬件。 网络接口结构如图5所示。通过next指针，网络接口被连成一个全局链表（global linked list）。

每个网络接口有一个名字，存储在图5中的name字段。这个两个字符的名字识别用于网络接口中的设备驱动类型，而且当接口在运行时由人为操作来配置。这个名字由设备驱动设置，应当映射由网络接口表示的硬件类型。例如，蓝牙驱动网络接口可能使用名字btd，而IEEE802.11b WLAN硬件可能使用名字wl。由于这些名字不必是唯一的，num字段用来区分同类设备中的不同的网络接口。

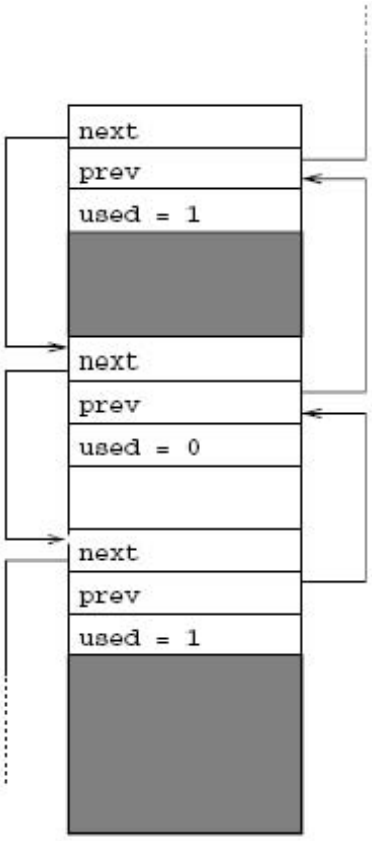


Figure 4. The memory allocation structure.

```
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.

当发送和接收包时，三个IP地址ip_addr，netmask和gw由IP层使用，它们的用途在下一节叙述。给网络接口配置多于一个IP地址是不允许的，一个网络接口应当为每一个IP地址创建。

当包接收到时，设备驱动应当调用input指针指向的函数。

网络接口通过output指针连接到设备驱动。这个指针指向设备驱动中的一个函数，它在物理网络上传送一个包，当一个包被发送时它由IP层调用。这个字段由设备驱动初始化函数填充。output函数的第三个参数ipaddr是主机的IP地址，它可以接收实际链路层的帧。它不应和IP包的目的地址相同。特别地，当发送一个IP包到不在本地网络的主机时，链路层帧被发送到网络

上一个路由。这种情况，给output函数的IP地址将是路由的IP地址。

最后，state指针指向设备驱动中网络接口的特定状态，由设备驱动设置。

8 IP处理

lwIP仅仅实现IP最基本的功能，它可以发送，接收和转发包，但不能发送或接收分割的IP包，也不能处理带IP选项的包。对于大多数应用来说这不会引起任何问题。

8.1接收包

对于接收的IP若干包，处理从ip_input( )函数被设备驱动程序调用开始。在这里，初始化工作将检查IP版本，同时确定报头长度，还会计算和检查报头checksum域。期望的情况是，自从Proxy服务器重组所有碎片（fragmented）包以来，堆栈就再没有收到碎片(fragments)，这样任何IP碎片的包都会被默默的丢弃。带有ip选项的包同样会被指定为由代理处理，并因此被丢掉。

接下来，函数通过网络接口的IP地址检验目的地址以确定包是否去往主机。网络接口已在链表中排序，可以线性查找。网络接口的序号指定为是小的号，因为比线性查找更巧妙的查找方法还没实现。

如果接收的包是主机指定的包，将使用protocol域来决定该包应该传给哪个更高层协议。

8.2发送包

一个要发送的包由函数ip_output( )处理，它使用函数ip_route( )寻找适当的网络接口来上传包。 当时发送包的网络接口被确定后，包被传递到ip_output_if()函数，该函数把发送网络接口作为一个函数自变量。 在这里，所有IP报头域被填补并且IP报头checksum被计算。 IP包的源和目的地址作为变量传递给ip_output_if()函数。源地址可能被略去（left out），然而，在这种情况下要发送的网络接口的IP地址将被用作包的来源IP地址。

ip_route()函数通过线性查找网络接口列表找到适合的网络接口。在查找IP包的目的IP地址期间，用网络接口的网络掩码进行掩码。如果目的地址等于经掩码的接口IP地址，则选择这个接口。如果找不到匹配的，则使用缺省网络接口。缺省网络接口由人工操作在启动时或运行时配置。如果缺省接口的网络地址和目的IP地址不匹配，则选择网络接口结构中的gw字段作为链路层帧的目的IP地址。（注意这各情况下IP包的目的地址和链路层帧的IP地址是不同的。）路由的原始形式忽略了这个事实：一个网络可能有许多路由器依附它。而工作时，对于一般情况下，一个本地网络只有一个路由器。

因为运输层协议UDP和TCP在计算运输层校验和时需要有目的IP地址，所以在包传给IP层前外发网络接口在某些情况下必须已确定。这可让运输层函数直接调用ip_route()函数完成，因为在包到达IP层时外发网络接口已经知道，没必要再查找网络接口列表。而是那些协议直接调用ip_output_if()函数。由于这个函数把网络接口作为参数，可避免外发接口的查找。

注：运行期间lwIP的手工配置要有一个能配置栈的应用程序，lwIP中不包含这样的程序。

8.3转交包

如果没有网络接口的IP地址和传进包的目的地址相同，这个包应当转发。这由函数ip_forward()完成。在这里，TTL字段减小，如果变为零，则ICMP错误信息被发送到IP包原始发送器并丢弃这个包。由于IP头被改变，有必要调整IP头校验和。然而不必重算完整的校验和，因为可用简单的算术来调整原始的IP校验和[MK90,Rij94]。最后，包被转发到适当的网络接口。用来寻找合适网络接口的算法和发送IP包时使用的一样。

8.4 ICMP处理

ICMP处理是相当简单的。 由ip_input()收到的ICMP包被移交到icmp_input()，它解析ICMP报头并且进行适当的处理。一些ICMP信息被传递到更高协议层并被传输层的一些特殊函数处理（Some ICMP messages are passed to upper layer protocols and those are taken care of by special functions in the transport layer）。ICMP目的地不能到达的信息能被运输层协议发送，尤其是UDP，和函数icmp_dest_unreach( )。

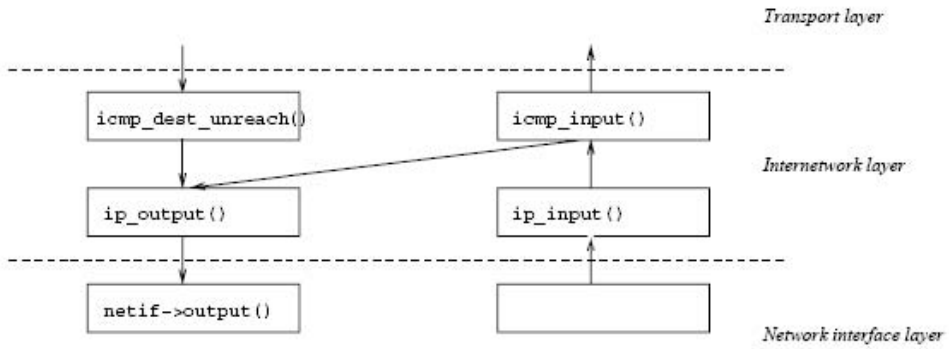


Figure 6. ICMP processing

使用ICMP ECHO信息来探测网络被广泛应用，因此对ICMPECHO进行了性能优化。 实际处理在函数icmp_input( )中发生，包括对接收的包的目的和源地址进行交换，改变ICMP类型为echo reply并且调整ICMP checksum。 然后包回送到IP层等待传送。

9 UDP 处理

UDP 是一个简单的协议，用来完成不同处理过程间的包分离。每一个UDP话路(session)的状态都被保留在一个PCB 结构中，如图7所示 。UDP PCBs 保存在一个链表中，当UDP datagram到达，则搜索该链表并进行匹配。

UDP PCB 结构中包含一个指向全局UDP PCB链表中的下一个 PCB的指针。UDP话路(session)由IP地址和端口号来定义，并且被存放在local_ip，dest_ip,local_port,dest_port域中。Flags域指出这一话路（session）将使用什么样的UDP 校验和策略。这可能既没关掉UDP checksumming 完全，或者使用UDP 轻便在哪个检验数字盖住只数据报的部分。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.
当接收到由PCB标明的session中的datagram时，最后二个参数 recv 和recv_arg将被使用。当接收到datagram时，recv所指向的函数被调用。

```
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

由于UDP较为简单，输入和输出处理也较简单，如图8所示。

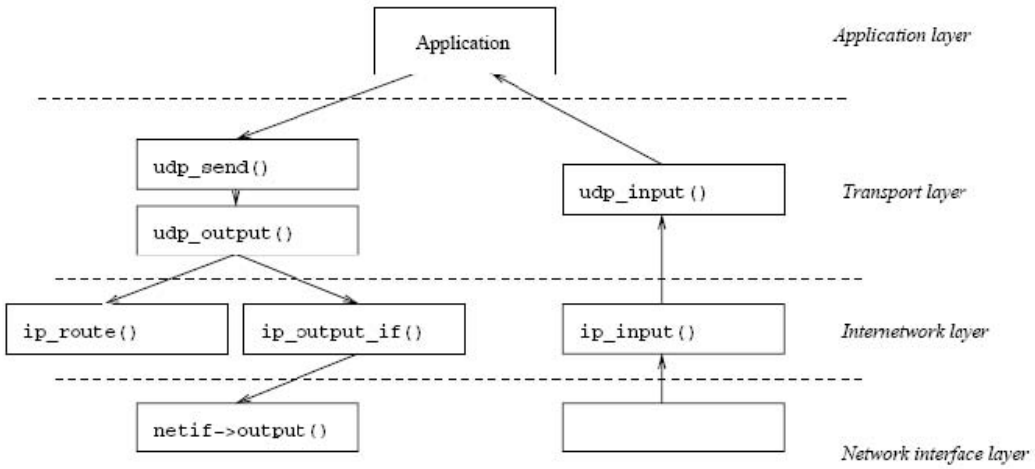


Figure 8. UDP processing

当收到一个UDP datagram 时，IP层调用udp_input()函数。这里，如果session应该使用checksumming，UDP checksum将被检查同时datagram被分离。当发现相应的UDP PCB，recv函数被调用。

10 TCP 处理

TCP 为传输层协议它为应用层提供可靠的二进制数据流服务。 TCP协议比这里描述的其它协议都要复杂，并且TCP 代码占lwIP总代码的50%。

10.1 总述

基本TCP 处理(图9) 被划分成六个函数；函数tcp_input()、tcp_process()、tcp_receive()与TCP 输入处理有关，tcp_write()、 tcp_enqueue()、tcp_output() 对输出进行处理。

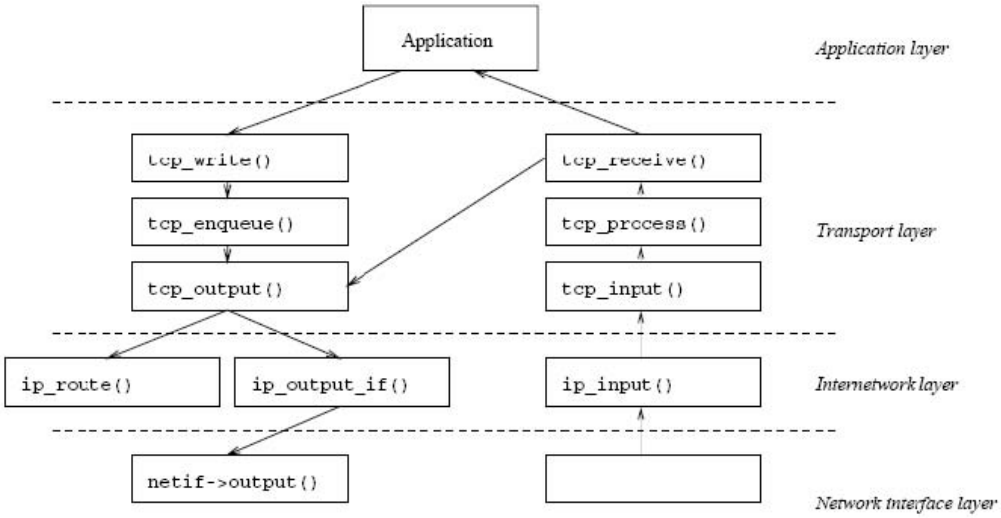


Figure 9. TCP processing

当应用程序想要发送TCP 数据，函数tcp_write()将被调用。 函数tcp_write() 将控制权交给tcp_enqueue()，该函数将数据分成合适大小的TCP段（如果必要），并放进发送队列。 接下来函数tcp_output()将检查数据是否可以发送。也就是说，如果接收器的窗口有足够的空间并且

拥塞窗口足够大,则使用ip_route()和ip_output_if() 两个函数发送数据 。

当ip_input()对IP报头进行检验且把TCP段移交给tcp_input()函数后，输入处理开始。 在该函数中将进行初始检验(也就是 ,checksumming 和TCP 剖析析)并决定该段属于哪个TCP连接。该段于是由tcp_process()处理,它实现TCP状态机和其他任何必须的状态转换。 。如果一个连接处于从网络接收数据的状态，函数tcp_receive() 将被调用。 如果那样，tcp_receive() 将把段上传给应用程序。 如果段构成未应答数据（先前放入缓冲区的）的ACK，数据将从缓冲被移走并且收回该存储区。 同样，如果接收到请求数据的ACK，接收者可能希望接收更多的数据，这时tcp_output() 将被调用。

10.2 数据结构

由于小型嵌入式系统内存的限制，LWIP所使用的数据结构被故意缩小。 这是在数据结构复杂度与使用数据结构的代码的复杂度之间的一个折衷。这样就因为要保证数据结构的小巧而使代码复杂性增加。

TCP PCB相当大，如图10 。 因为TCP 连接在处于监听（listen）和时间等待（TIME-WAIT）状态时比处于其他状态的连接需要保留较少状态信息，对于这些连接使用了一种更小的PCB 数据结构。 这种数据结构镶嵌在完整的PCB 结构中，在PCB 结构中的排列持续如图10， 因此有些笨拙。

TCP PCBs 被保留在一份链表中，并且next指针把PCB列表连接在一起。 状态变量包含当前连接的TCP状态。 其次，辨认连接的IP 地址和端口号被保存。 mss 变量保存连接所允许的最大段大小。

当接受数据时，rcv_nxt 和rcv_wnd域被使用。 rcv_nxt域包含期望从遥端的下个顺序编号（contains the next sequence number expected from the remote end），因而当发送ACKs 到远程主机时被使用。接收器的窗口被保留在rcv_wnd中,并且在将要发出的TCP 段中被告知。 tmr被作为定时器使用，在经过一特定时间后连接应该被取消，譬如连接在TIME-WAIT 状态。 连接所允许的最大段大小被存放在mss域中。 Flags域 包含连接的附加状态信息，譬如连接是否为快速恢复或被延迟的ACK是否被发送。

```
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

域rttest，rtseq，sa，和sv 被使用为round-trip时间估计。 用于估计的段顺序号存储在rtseq ，该段被发送的时间存放在rttest。 平均round-trip时间 和round-trip时间变化分别存放在sa和sv 。 这些变量被用来计算存放在rto域中的转播暂停（retransmission time-out）。

二个域lastack 和dupacks 被用来实现快速转播和快速重发。 lastack包含由最后接受到的ACK应答的顺序编号，dupacks 对接收到多少关于存储在lastack的顺序编号的ACK 进行计数。 当前连接的阻塞窗口被存放在cwnd域，并且缓慢的起动门限被保留在ssthresh 。

六个域snd_ack、 snd_nxt、snd_wnd、snd_wl1、snd_wl2 和snd_lbb 在发送数据时使用。 由接收器应答的最高的顺序编号被存放在snd_ack， 并且下个被发送的顺序编号保存在snd_nxt 。接收器的广告窗口(advertised window)保存在snd_wnd，两域 snd_wl1 和snd_wl2在更新snd_wn时使用。 Snd_lbb域包含发送队列最后字节的顺序编号。

当传递接受的数据到应用层时将使用函数指针recv 和recv_arg。 三个队列unsent，unacked和ooseq当发送和接受数据时被使用。 已经从应用接收但未被发送的数据由队列unsent进行排队，已发送但还没有被远程主机应答（acknowledged）的数据由unacked 存储。 接收的序列外面的数据由ooseq进行缓冲。

```
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

表11中的Tcp_seg 结构是TCP 段的内部表示方法。 这个结构由一个next指针开始，该指针用于连接排队段。 len域包含段的长度。 这意味着一个数据段的len域将包含段中数据的长度，具有SYN或FIN标志的空段的len被设置为1 。 pbuf结构类型指针p 是包含实际段、tcphdr、指向TCP头的数据指针和数据段的缓冲。 分别的，对于将要外发的段，rtime域被用于该段的转播暂停。 因为接收的段不会需要被转播，对于接收段来说这个域并不需要并且也不为该域分配内存。

10.3 顺序编好计算（Sequence number calculations）

用于枚举TCP二进制数据流的TCP顺序编号为无符号32位数据， 因此其范围是 $[0, 2^{32} - 1]$ 。因为在TCP 连接中要发送字节的数量可能会比32 位组合的数量更多，顺序编号以2^{32}为模数进行计算 。 这意味着，普通的比较操作符无法被TCP 顺序编号使用。 修改过的比较操作符，叫做$<_{seq}$和$>_{seq}$，由下面关系定义：

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

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

这里s 和t 是TCP 顺序编号。 比较操作符≦≧也等效地被定义。 比较操作符作为C 宏指令被定义在标头文件。

10.4 队列和发送数据

将要发送的数据被划分成适当的大小的大块并tcp_enqueue()指定顺序编号。这里，数据被打包进pbufs 结构并附加进tcp_seg 结构。 在pbuf内，TCP头被建立，并且除应答数字，ackno和广播窗口，wnd以外的所有域被填充。 这些域在段排队时可以被改变，并由tcp_output()进行设置，该函数完成段的实际传输。 在段被建立之后，他们被送往PCB里的unsent列表进行排队 。 函数tcp_enqueue()设法用最大段大小的数据填装各段直到在unsent 队列的末端发现段under-full，该段使用pbuf chaining functionality功能被添附以新数据。

在tcp_enqueue() 格式化和排队了段之后，tcp_output()函数被调用。它检查在当前的窗口中是否还有空间来存储更多的数据。当前窗口的数值是通过窗口数据拥挤的最大量和接收窗口的广播。（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）。 其次，它填装由tcp_enqueue() 未填装的TCP头的域，并且使用ip_route()和 ip_output_if()传送段。 在段被放入传输后unacked表后,停留直到该段的ACK被接收到。 当段放入unacked 表的同时，如在10.8部分所描述同时也为重发计时。 当段需重发时原来的TCP、IP头被保留，只需对TCP 头做少量变化。 TCP头的ackno 和wnd域被设置为当前值，因为在段的原始传输和重发期间我们可能接收了数据。 这只改变报头里的二个16 位字和整体TCP checksum不必要重新计算，因为简单的算术[Rij94] 可以用来更新checksum。当段最初被传送时 ，IP 层已经增加了IP 头，并且没有理由改变它。 因而重发不要求IP头的checksum.的任何重发计算。

Silly Window Syndrome [Cla82b] (SWS)综合症状[Cla82b] (SWS) 是一个可能导致非常坏的性能的TCP 现象。当TCP 接收器广播一个小窗口并且TCP 发送者立刻发送数据填装窗口时SWS发生。 当这小段被应答窗口再次打开并且发送者将再发送小段填装窗口。 这导致TCP 数据流包括许多非常小段的情况发生。 为了避免SWS 发送者和接收者都必须设法避免这个情况。 接收者不能给小窗口更新做广播并且当只提供一个小窗口时发送者不能送小段。

在lwIP， SWS 在发送端自然地被避免，因为TCP 段被建立和排队时没有做广播接收器窗口的应答。 在大数据量传输时队列将包括最大尺寸大小的段。 这意味着如果TCP 接收器广播一个小窗口，发送者不会送队列中的第一个段，因为它比广播的窗口大。相反，它将等待直到窗口是足够大以至于能容纳最大大小的段。 当作为TCP 接收器时，lwIP 不会给比连接的最大段大小小的接收器的窗口做广播。

10.5 接收段

10.5.1 复用

当TCP 段到达tcp_input() 函数时，他们将在TCP PCBs之间被解析（demultiplexed） 。 解析的关键是来源和目的地IP 地址和TCP 端口数。 当解析段的时候有种二类型的PCBs必须突出的（distinguished）：那些对应于开放连接的和那些对应于连接是半的打开。 半开放连接是指那些处于监听状态和只有本地TCP端口号被指定且本地IP 地址为任意值的连接，但是开放连接有指定的两个IP 地址和两个端口号。

许多TCP 的实现，譬如早期的BSD 实现，使用具有单一入口缓存的PCBs链表技术。 在这之后基本理论是，多数的TCP 连接构成批量传送，它典型地显示一个大批量的位置[Mog92]，造成一个高缓冲命中比率。 另一个方案包括两个具有单一入口的缓冲器，一个为对应于被送的最后的包的PCB，一个为最后接受的包的PCB [PP93] 。 通过移动最近被使用的PCB 到列表的前端，一份供选择的方案可以实施。 两种方法 [MD92]都胜过单一入口的缓冲方案。

在lwIP，每当PCB 匹配被发现，解析段时，PCB都将被移动向PCBs 列表的前端。 但是，在听状态的连接所用的PCBs并不被移动，因为这种连接并不期望接收段。

10.5.2 接收数据

对接收到的段的实际处理是在函数tcp_receive()里进行的。 段的应答数字与在unacked队列中的段比较。 如果应答数字比段在unacked 队列的顺序编号高，该段从队列中被移走并且为段分配的内存也被收回。

如果接收的段的顺序编号比PCB中rcv_nxt变量高，该段将脱离序列。序列外的段在PCB中的ooseq 队列进行排队。 如果接收的段的顺序编号与rcv_nxt 是相等的，通过调用在PCB中的函数recv，段被转交给上层，并且关于接收段的长度的rcv_nxt域被加进来。 因为在序列内的段的接收也许意味着先前被接受的在序列外的段是被期望的下一个段，ooseq 队列被检查。 如果它包含顺序编号与rcv_nxt顺序编号相等的段，通过调用函数recv该段被转交给应用程序并且更新rcv_nxt。 这个过程持续到ooseq 队列为空或ooseq的下一个段脱离序列。

10.6 接受新的连接（Accepting new connections）

处于听状态，也就是说已经被动地开放的TCP 连接，，已经准备从远程主机接受新的连接。为了那些连接，必须建立新的TCP PCB ，并传递给打开初始听连接的应用程序。在lwIP里， 这个步骤是通过使用回调函数来完成，这个函数当一个新的连接被建立时被调用。

当处于听状态的连接接收一个具有SYN标志的TCP段时（When a connection in the LISTEN state receives a TCP segment with the SYN flag set），一个新的连接就建立了，并且一个带有SYN和ACK的标志的段被送出以回应那个SYN段。这个时候这个连接就进入了SYN-RCVD状态，并等待发送的SYN段的应答，当应答信息收到后，这个连接就进入了ESTABLISHED阶段，接受函数（在PCB中的accept域如图10）会被调用 。

10.7 快速转发（Fast retransmit）

IWIP里拥有快速转发和快速恢复的功能。该功能通过明确最后被应答的顺序编号实现的。如果收到同一顺序编号的另外个应答信号，TCP PCB 里的 dupacks 计数建立。当DUPACKS到3的时候，

在未应答序列中的第一个段将要被重新送出，快速恢复被初始化。快速恢复的步骤完成以后的动作 [ASP99]：无论什么时候收到一个新数据的应答信号，dupacks计数都复位为0。

10.8 定时器（Timers）

和在BSD TCP 中实现一样，lwIP 使用二个周期性定时器，周期分别为500ms和200ms。 这二个定时器又被用来实现更加复杂的逻辑定时器，如转发定时器、TIME-WAIT 定时器和delayed ACK 定时器。

fine grained timer定时器，如果有应该被发送的任一被延迟的ACKs，tcp_timer_fine() 将在定时结束时检查TCP PCB，正如在tcp pcb 结构中的flag域所表明的（图10）。 如果delayed ACK 标志被设置，空的TCP 应答段被发送并且标志被清除。

coarse grained timer定时器，在tcp_timer_coarse()里实现，并且扫描PCB 列表。 对于任何一个PCB，未应答段列表(在tcp_seg 结构中的unacked指针,如表11)，被否认(is traversed)，并且rtime 变量被增加。 如果rtime 比当前的转发暂停时间（由PCB结构中的rto 变量给出）大， 段被转发并且转发暂停被加倍。只有在拥塞窗口和广播接收器的窗口的值允许的情况下段才被转发。 在转发以后，拥塞窗口被设置成一最大段大小，slow start threshold被设置为有效的窗口大小的一半，并且slow start在连接中被初始化。

对于在TIME-WAIT状态的连接，coarse grained timer定时器也会在PCB 结构中增加tmr域。当这个定时器到达2×MSL门限，连接被取消。 coarse grained timer定时器同时也增加一个全局TCP 时钟，tcp_ticks。 这个时钟用来估计round-trip时间转播暂停（time－out）。

10.9 Round-trip 时间估计（Round-trip time estimation）

round-trip时间估计是TCP 的一个重要部份，因为估计的round-trip时间被用来确定适当的转发暂停。在lwIP，round-trip时间测量的实现与BSD 相似。每一次round-trip往返 round-trip时间就被测量一次，并且使用smoothing函数（在[Jac88]中描述） 对适当的转发暂停进行计算。

TCP PCB的变量rtseq 保存着往返时间被测量的段的顺序编号。 PCB中的Rttest变量保存着当段第一次被传送时tcp_ticks的值。 当一个顺序编号等于或大于rtseq的ACK被接收到时，round-trip时间通过从tcp_ticks减去rttest被测量。 如果转发发生在round-trip时间测量期间，测量不被采取。

10.10拥塞控制（Congestion control）

拥塞控制的实现是出乎意料的简单，仅仅包括几行代码。 当一个新数据的ACK被拥塞窗口接受，cwnd，被mss^2/cwnd 增加或被一最大段大小增加，取决于连接是缓慢起动还是拥塞避免（depending on whether the connection is in slow start or congestion avoidance）。 当发送数据时，接收器的广播窗口和拥塞窗口的最小值用来确定每个窗口能够发送数据的多少。

11栈接口（Interfacing the stack）

使用由TCP/IP协议栈提供的服务有二种方式；一种是直接调用在TCP 和UDP 模块中的函数,另一种就是使用lwIP API。

TCP 和UDP 模块提供一个网络服务的基本接口。 该接口基于回调，因此使用它的应用程序可能因此不必以连续方式进行操作。 这使应用程序的编程更加困难并且应用代码更难理解。 为了接受数据，应用程序登记一个协议栈的回调函数。 回调函数同一个特定的连接联系在一起，当该连接的包到达时，回调函数被协议栈调用。

此外，与TCP 和UDP 模块直接接口地应用程序，必须(至少部份地)保留在像TCP/IP协议栈这样的处理过程中。 这归结于回调函数无法横跨处理界限调用的事实。 这既有好处也有不足。好处是应用程序和TCP/IP 协议是在同一个处理过程中，发送和接收包时不用上下文切换。 主要不足是，在任何长的连续计算过程中应用程序无法介入自己，因为TCP/IP 处理无法与计算平行发生， 因而丧失通讯性能。 通过把应用程序分成两部分可以克服这一缺点，一部分应付通信一部分应付计算。 负责通讯的部分驻留在TCP/IP 过程中，负责计算的部份将是一个单独的过

程。 将要在下一节介绍的lwIP API 提供了一个结构化的方式，用这样方式来划分应用程序。

TCP/IP的处理不应该与其他运算并行处理,这样将会降低通讯的性能,所以我们把应用程序分解为两个部分,一部分专注于处理通讯,另一部分做其他的运算。通讯的部分将包含在TCP/IP进层中，而其他的繁杂运算则作为一个独立的进程。下一节将介绍LWIP 的 API 提供的分开的结构来应用的办法。

12 应用程序接口 API

作为高级别的BSD socket API,它是不适宜用于一个最小限度的TCP/IP执行的.特别BSD SOCKETS要求在TCP/IP协议栈中将要发送的数据从应用程序拷贝到内部缓存.需要拷贝数据的原因是通常TCP/IP协议栈和应用程序一般都处在不同的保护领域.大多数时候应用程序是位于户进程而TCP/IP却在操作系统内核中。通过避免这额外的拷贝就可以大幅度的提高API的性能[ABM95]。同样地,这样的拷贝需要分配额外的内存，每个信息包都浪费了双倍的内存。

LWIP API是专为LWIP设计并利用LWIP的内部结构达成效果，LWIP API 和 BSD API非常类似，但操作相对低级。LWIP API不需要TCP/IP和应用程序之间的相互拷贝数据，应用程序可以巧妙的直接处理内部缓存。

由于BSD SOCKET API 很容易理解且已经有很多人为它写过应用程序，LWIP API很有必要有与BSD SOCKET 的兼容层面。17节中介绍了如何用LWIP API 去重写BSD SOCKET 函数，15节中有LWIP API 的一个参考手册。

12.1 基本概念

从应用程序去看，BSD SOCKET API的数据处理都是在一个连续的内存区域完成的。这是因为应用程序通常也是在这样的一个连续大内存块中完成数据处理的。LWIP采用这样的机制是没有优势的，因为IWIP通常处理的数据缓存都被分割成了小的内存块。在通过应用程序前这些数据就会不得不要复制到一个连续的内存区，这将浪费双方的处理时间和内存,因此LWIP API允许应用程序巧妙地直接处理分离的缓存去避免额外的拷贝。

LWIP API 尽管非常类似 BSD SOCKET API，可是却有着值得注意的不同的地方，应用程序使用BSD SOCKET API 时候不需要知道普通文件和网络连接之间的差别，但使用LWIP API的时候就必须要知道确实在使用网络连接。

网络数据被接收到分离的内存块的时候是以缓存的形式出现的,由于很多应用程序都希望在一个连续的内存区域处理数据，这就要有个函数去把这些缓存碎片复制到连续的内存空间中。

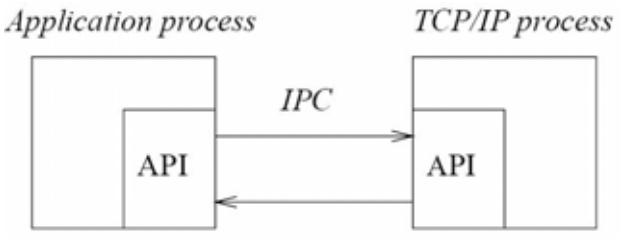
发送出去的网络数据根据是TCP连接还是UDP是不同地处理的。在TCP连接时，数据通过一个指向连续内存区的指针发送，TCP/IP协议为传送区分适当大小的数据包和数据队列。在发送UDP数据的时候，应用程序将明确地分配缓存并填充数据，在输出函数被调用的时候由TCP/IP协议栈马上发送出去。

12.2 API的执行

API是分做两个部分执行的,在图12中我们可以看到，一部分位于在TCP/IP的进程模块中,一部分当作连接库在应用程序中执行,这两部分的API通过由操作系统仿真层提供的进程间通讯(IPC)传达信息.当前使用的IPC机制有以下3种：

共享内存
消息
信号

当操作系统支持这些IPC类型的时候，并不意味着他它们得到了操作系统的最底层的支持，因为操作系统并不是一开始就支持它们,只是操作系统的的仿真层仿真了它们。



图示12． 分开为两部分的API执行

一般的设计原理都是尽可能地提高TCP/IP进程里的API的工作能力更胜于应用程序里API.这很重要,因为大部分的进程都用TCP/IP进程进行它们的TCP/IP通讯。遵循API部分的代码足迹,和应用程序连接的这部分并不是很重要.这些代码可以在进程间共享,即使操作系统并不支持共享连接库。这些代码还可以保存在ROM中，嵌入系统一般尽管处理能力不高却拥有很大的ROM空间。

缓存管理器位于API执行库里，buffer在应用程序进程中被建立,复制和分配.应用程序和TCP/IP进程间使用共享内存来传递buffer，应用程序中用于通讯的buffer数据类型是一种提取于pbuf的类型。

buffer传输引用到的内存，和分配的内存不同,它是可以利用共享内存的.所以可以进程间共享引用的内存.运行LWIP的嵌入式操作系统一般有意不做任何形式的内存保护,所以不会有问题。

操作网络连接的函数位于TCP/IP进程的API中.应用程序的API函数会通过一个简单的通讯协议传递一个信息个TCP/IP进程的API,这信息包含操作的类型,和操作的相关变量.这个操作由TCP/IP的API传输后用信息给应用程序一个返回值。

13 代码统计分析

本节分析了LWIP的源代码行的数量和编译后结果大小的关系.代码被两种不同的处理器结构编译：

Intel Pentium III处理器,在FreeBSD 4.1下面用GCC 2.95.2 编译,开启了编译优化选项。

6520处理器[Nab, Zak83].用cc65 2.5.5 [vB] 编译,开启了编译优化选项。

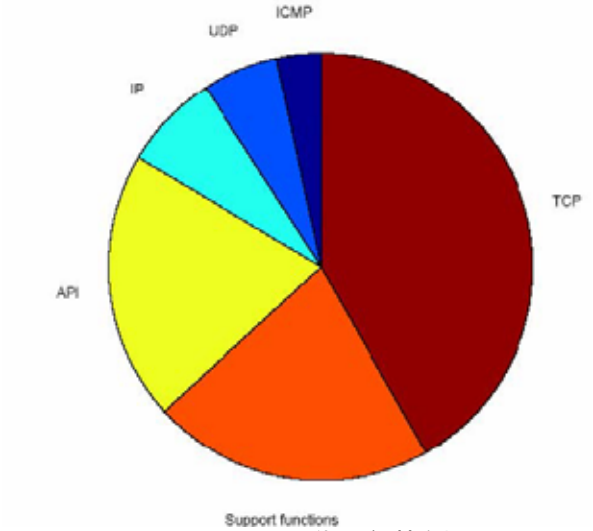
Intel x86 用了7个32位寄存器和32位指针。 6502主要用于嵌入系统,具有一个8位累加器两个8位索引寄存器和16位指针。

13.1 代码行

表1

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%



图示13　代码行比例

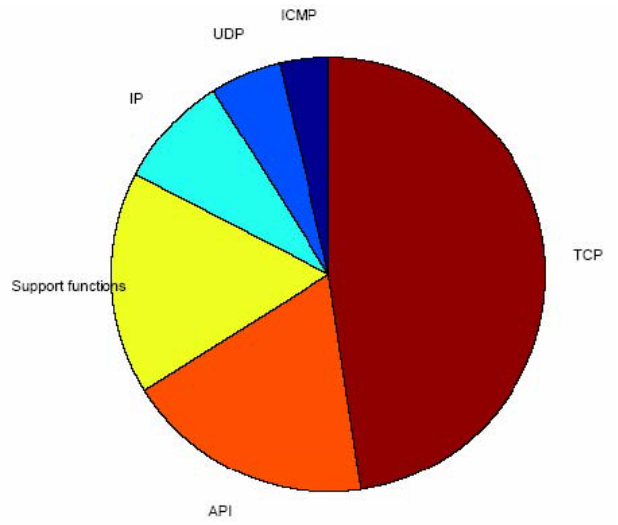
表1介绍了LWIP代码行的数量图示13显示了各部分的代码比例。"sunpport functions"类别包括了缓存和内存管理器还有校验和处理函数，在事实配置支持的情况下，校验和函数应该用普通C执行而不用处理器特殊的运算法则。"API"类别包括了应用程序的API和TCP/IP协议栈的API.操作系统的仿真层在这里没有分析显示，因为它在操作系统的底层而且大小有很大不定性，在这里就没必要比较它了。

作为比较，这里忽略了所有的注释，空白行和头文件.我们可以看到TCP这一块比其它的执行协议都大得多，,而API和support functions 加起来就和TCP差不多大。

13.2 目标结果代码大小

表2. LWIP在Intel x86下编译后代码的大小

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%

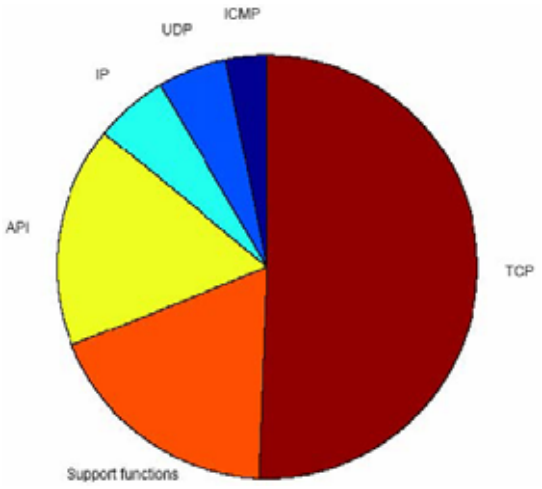


图示14. LWIP在X86下编译后代码的大小。

表 2总结了在Intel x86下编译后代码的大小，图示14 显示了相对大小比率.我们可以看到项目的排序有了点小变化。这里API大于support functions 类别，尽管support functions的代码也增加了很多。我们也可以看到TCP的编译后大小占了总大小的48%但源代码行却只占总源代码行数的42%。经检查TCP模块的汇编输出发现了一种可能性，TCP模块里包括了大量的解除参照指针(原文pointer dereferencing)在很多汇编代码行里，因此增加了编译后代码的大小。很多这些解除参照指针被每个函数用了2或3次。通过修改源代码让这些指针在一个地方只解除参照(dereferenced)一次然后放进一个本地变量里就应该可以达到优化目标代码的目的。当减少了编译后目标代码的大小后,可能要为堆栈分配更多的内存来存放这些本地变量。

表 3. LWIP 在6502下编译后的目标代码的大小。

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%



图示 15. LWIP 在6502下编译后目标代码的大小比例

表 3 列出了在6502下编译后目标代码的大小。图示 14 显示了各部分的大小比率。我们可以看到TCP，API和support functions几乎是在Intel x86下编译的两倍，然尔IP，UDP和ICMP却几乎一样的大小。我们也可以看到和表2相反过来support functions比API还大,API和support functions的代码大小的差别变小了。

TCP模块大小增加的原因主要是6502本来并不支持32位的整数，因此编译器把每个32位的操作展开为多行汇编代码。TCP序列号就是一个32位的整数而TCP模块执行了多次使用序列号的计算。

我们可以将LWIP里面TCP的代码大小和其他的TCP/IP栈的TCP代码进行比较，比如和现在很流行的Free BSD4.1下的BSD TCP/IP和Linux2.2.10下的TCP相比较。在Intel x86下用打开优化选项用gcc编译，LWIP的TCP的执行代码接近6600bytes,FreeBSD 4.1的TCP的目标执行代码大概27000bytes,几乎是LWIP的4倍。在Linux 2.1.10下，TCP的编译后目标执行代码高达39000bytes，大概是LWIP的6倍。这样大的差别事实上是因为FreeBSD和Linux2.2.10包含了很多TCP的特性象SACK[MMFR96]与BSD的socket API.

在这里我们不再做IP的执行代码的大小比较，因为在FreeBSD和linux中包含了大量的IP执行特性，比如：FreeBSD和linux支持防火墙和IP执行隧道，还有动态路由表等，这些都是LWIP中没有的。

LWIP的API的代码量大概是LWIP总代码量的六份之一，因为LWIP可以在没有API的情况下运行，系统配置的时候可以略掉这部分而只占用很小的代码空间。

14 性能分析

LWIP的内存使用和代码效率性能没有在本文中做非常正式的测试，这将在以后大家使用中体会，我们做了个简单测试，我们用LWIP运行一个简单的HTTP/1.0的WEB服务器，在少于4K的RAM下可以响应至少10个的同步网页请求。内存用于协议，系统缓存，应用程序都计算在内。因此设备驱动使用的内存将增加以上的数字.

15 API 参考手册

15.1 数据类型

以下是LWIP API用到的两种数据类型：

netbuf,网络缓存
netconn,网络连接
每种类型都是一个C结构体的指针，由这个结构体的内部结构我们知道它不应该在应用程序中使用，API取代它提供了修改和提取必要数据域的功能函数。

15.1.1 Netbufs

Netbufs 是用于发送接受数据的缓存，在6.1小节中介绍了netbuf和pbuf之间的内部关联。Netbufs可以当做pbufs容纳分配的内存和引用的内存。分配的内存是专门分配给持有网络数据的RAM，然而引用的内存可能不是应用程序管理的RAM就是外部的ROM。引用的内存对发送数据是很有用的，它不能修改，比如静态的网页和图片。

数据在netbuf里面可以是不同大小的碎片块，这意味着应用程序必须准备好接受零碎的数据。netbuf内部有一个指针指向netbuf里面的一个碎片，netbuf_next() 和 netbuf_first() 两个函数就是利用了这个指针。

从网络中接受的Netbufs包含了发送数据包来源的IP地址，端口号。然后可以用函数来提取了那些存在的值。

15.2 Buffer 函数

15.2.1 netbuf new()

摘要

描述

分配一个netbuf结构体，执行这里并没有分配buffer空间，只是建立结构体的开始，执行后，必须用netbuf_delete()来删除netbuf。

15.2.2 netbuf delete()

摘要

描述

删除先前由netbuf_new()分配的netbuf结构体，由netbuf_alloc()分配的任何buffer也会同时释放。

例子　这个例子显示了使用netbufs的最基本结构。

```
int
main()
{
struct netbuf *buf;
buf = netbuf_new(); /*  建立一个新的netbuf */
netbuf_alloc(buf, 100); /*  分配  100 bytes  作为  buffer */

struct netbuf * netbuf new(void)

void netbuf delete(struct netbuf *)
/*  使用netbuf */
/* [...] */
netbuf_delete(buf); /*  删除  netbuf */
}
```

15.2.3 netbuf alloc()

摘要

描述
以字节为单位为netbuf buf分配buffer，这个函数返回一个指针指向被分配的内存，先前曾经分配给过netbuf buf的空间会重新分配。分配的内存以后可以用netbuf_free() 函数来重新分配。因为发送数据的时候会先发协议头，所以这函数为协议头分配空间的时候也为实际的数据分配了空间。

15.2.4 netbuf free()
摘要

描述
重新分配和netbuf buf关联的buffer空间，如果buffer空间已经为netbuf buf分配过，这函数将会不做任何操作。

15.2.5 netbuf ref()
摘要

描述
将外部存储器的指针和netbuf buf的数据指针关联起来。外部存储器的大小由参数size提供。先前分配给netbuf的内存空间被重新分配。在用netbuf_alloc()为netbuf分配空间和用malloc()分配空间和用netbuf_ref()涉及它不同的是，为协议头分配的空间会让处理和发送buffer的速度更快。

例子　这个例子显示了netbuf_ref()函数的简单用法。

```
int
main()
{
struct netbuf *buf;
char string[] = "一个字符串";
/*  建立一个新的netbuf */
buf = netbuf_new();


void * netbuf alloc(struct netbuf *buf, int size)

int netbuf free(struct netbuf *buf)

int netbuf ref(struct netbuf *buf, void *data, int size)

/*  关联字符串*/
netbuf_ref(buf, string, sizeof(string));
/*  使用netbuf */
/* [...] */
/*  重新分配  netbuf */
netbuf_delete(buf);
}
```

15.2.6 netbuf len()
摘要

描述
返回netbuf buf中数据的总长度值，不管netbuf是不是碎片状的。netbuf是碎片的时候，返回值和netbuf里第一个碎片的大小值是不一样的。

15.2.7 netbuf data()
摘要

描述

此函数一般用于获得一个指针和netbuf 里面数据块的长度。变量data和len分别是指向数据的指针和指向数据的长度。如果netbuf是碎片状的，函数将指针指向netbuf里面的一个碎片。应用程序必须用netbuf_first()和netbuf_next()碎片处理函数来访问netbuf里所有的数据。
看下面netbuf_next()的例子如何使用netbuf_data()。

15.2.8 netbuf next()
摘要

描述
这个函数更新了指向netbuf buf内部碎片数据的指针，让指针指向netbuf里下一个碎片。如果netbuf里还有数据片段存在，那么返回值将>0 如果当前指针已经指向了最后一个数据片，那么返回值<0.

例子 这个例子显示如何使用netbuf_next()函数。我们假设函数中的buf变量是一个netbuf数据类型。

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

int netbuf len(struct netbuf *buf)

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

int netbuf next(struct netbuf *buf)


/*  获得一个指向数据片段的指针  */
netbuf_data(buf, &data, &len);
/*  使用这个数据*/
do_something(data, len);
} while(netbuf_next(buf) >= 0);
/* [...] */
```

15.2.9 netbuf first()

摘要

描述
复位netbuf buf里面指向数据片段的指针，让它指向第一个数据片段。

15.2.10 netbuf copy()
摘要

描述
复制所有netbuf buf里面的数据到data指针指向的内存空间，即使netbuf buf是碎片状的。len参数是复制到data指向内存空间的上限值。
例子　本例介绍了netbuf_copy()的简单用法，这里，在堆栈里分配给数据的是200字节的空间。即使netbuf buf里面的数据是超过了200字节，也只能复制200个字节给data指针。

```
void example_function(struct netbuf *buf)
{
char data[200];
netbuf_copy(buf, data, 200);
/*  使用数据  */
}
```

15.2.11 netbuf chain()
摘要

描述
把两个netbufs的头和尾相连起来，因此这个的数据的尾将变成下一个数据片段的头。在这个函数被调用后，原来netbuf的尾部将被重新分配不能再使用。

15.2.12 netbuf fromaddr()
摘要

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

void netbuf copy(struct netbuf *buf, void *data, int len)

void netbuf chain(struct netbuf *head, struct netbuf *tail)

struct ip addr * netbuf fromaddr(struct netbuf *buf)
```

描述
返回netbuf buf接收来自主机的IP地址。如果并没有从网络接收到netbuf，那么将返回一个未定义的值。调用netbuf_fromport()函数可以获得远程主机的端口号。

15.2.13 netbuf fromport()
摘要

描述

返回netbuf buf接受来自主机的端口号。如果并没有从网络接收到netbuf，那么将返回一个未定义的值。调用netbuf_fromaddr()函数可以获得远程主机的IP地址。

16 网络连接函数

16.0.14 netconn new()
摘要

描述

建立一个新的连接数据结构。变量可以是根据是TCP连接还是UDP连接分别为NETCONN_TCP 或 NETCONN_UDP。在连接没被确定前调用此函数将不能从网络发送数据。

16.0.15 netconn delete()
摘要

描述

删除连接netconn conn 。如果调用此函数的时候这个连接已经打开，那么结果就会关闭这个连接。

16.0.16 netconn type()
摘要

描述

返回连接conn的类型。这个类型将和netconn_new()的变量一样不是NETCONN_TCP就是NETCONN_UDP。

16.0.17 netconn peer()
摘要

描述

unsigned short netbuf fromport(struct netbuf *buf)

struct netconn * netconn new(enum netconn type type)

void netconn delete(struct netconn *conn)

enum netconn type netconn type(struct netconn *conn)

int netconn peer(struct netconn *conn,
struct ip addr **addr, unsigned short port)

在网络连接建立后调用本函数将获得远程连接的IP地址和端口。参数addr和port是函数设置的结果参数。如果连接conn没有和任何远程主机连接，将返回一个未定义的结果。

16.0.18 netconn addr()
摘要

```
int netconn addr(struct netconn *conn struct ip addr **addr unsigned short port)
```

描述
这个函数功能是用来获取连接conn的本地IP地址和端口号。
16.0.19 netconn bind()
摘要

```
int netconn bind(struct netconn *conn struct ip addr *addr unsigned short port)
```

描述
把连接conn和本地IP 地址addr 和TCP或者UDP 端口结合起来。 如果addr是无效的，本地IP 地址由网络系统确定。
16.0.20 netconn connect()
摘要

```
int netconn connect(struct netconn *conn
                    struct ip addr *remote addr
                    unsigned short remote port)
```

如果UDP是通过远程连接的接收者发送UDP消息来给出remote_addr和remote_port，那么对于TCP来说，netconn_connect()通过远程主机来打开连接。
16.0.21 netconn listen()
摘要

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

描述
让TCP连机conn进入TCP监听状态。
16.0.22 netconn accept()
摘要

```
struct netconn * netconn accept(struct netconn *conn)
```

描述
阻止进程直到接收到远程主机发送TCP连接conn的连接请求。连接必须处在监听状态，所以netconn_listen()的调用优先级必须比netconn_accept()高。当一个连接被远程主机建立时，一个新连接结构被返回。
例 这个例子来描述如何在2000端口上打开TCP服务器

```
int main()
{
    struct netconn *conn, *newconn;          /*创建一个连接结构体 */
    conn = netconn_new(NETCONN_TCP);
/*在任何本地IP地址上赋值连接2000端口*/
netconn_bind(conn, NULL, 2000); /* 通知连接来监听增加的连接请求*/
netconn_listen(conn);
```

```
/* block直到我们获取新的连接*/
newconn = netconn_accept(conn);
/*  使用newconn */
process_connection(newconn);
/*分配两个连接  */
netconn_delete(newconn);
netconn_delete(conn);
}
```

16.0.23 netconn recv()

摘要

```
Struct netbuf * netconn recv(struct netconn *conn)
```

描述

在等数据在连接conn上到达时终止进程。 如果远程主机已经关闭连接，返回NULL，否则返回的是在netbuf中接收到的数据。

示例：通过下面一个小例子来说明如何使用函数netconn_ recv()。我们假设在调用函数example_function()之前连接已经建立。

```
void
example_function(struct netconn *conn)
{
struct netbuf *buf;
/*  接收数据直到主机关闭连接*/
while((buf = netconn_recv(conn)) != NULL) {
do_something(buf);
}
/*  连接在另一端被关闭，所以我们这端也关闭连接  */
netconn_close(conn);
}
```

16.0.24 netconn write()

摘要

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

描述

这个功能只用于TCP 连接。 它签订被TCP 连接conn的关于输出队列的数据指向的数据。 镜子给数据的长度。 没有对数据的长度的限制。 当这被这个堆照顾时，这个功能不要求申请明确地分配bu &-#174;ers。 旗参数有两个可能的国家，象如下所示的那样。

这个函数仅用于TCP连接。It puts the data pointed to by data on the output queue for the TCP connection conn. 数据的长度由len给出。这里没有对数据的长度做限定。这个函数不需要应用程序明确分配缓冲区，这由堆栈来解决。参数flags有两种可能的状态，如下面所示：

```
#define NETCONN_NOCOPY 0x00
#define NETCONN_COPY 0x01
```

当使用NETCONN_COPY时 flag数据被复制到为这个数据分配的内部缓冲区。这种情况下允许数据在调用后直接修改，但是在运行的时间和内存的使用上效率是很低的。如果使用flag NETCONN_NOCOPY 数据就不是被复制而是引用。数据在调用后是不允许修改的，因为数据被放在连接的重发队列中，并保存在那里对于那些不确定时间的。（stay there for an indeterminate amount of time.）当要发送的数据是固定不变存在ROM里时这种方法就非常有用。

如果有大量的数据需要更改的话，可以是用复制和不复制数据的结合，如下面的例子所示。

例 这个例子说明了函数netconn_write()的基本用法。在这里那些可变的数据假设在程序运行后被修改，因此数据将被复制到内部缓冲区通过定义flag NETCONN_COPY。变量text包含一字符串将不被修改，这样的话就可以使用引用来代替复制。

```
int
main()
{
struct netconn *conn;
char data[10];
char text[] = "Static text";
int i;
/*建立连接conn */
/* [...] */
/*创建一些随机数据 */
for(i = 0; i < 10; i++)
data[i] = i;
netconn_write(conn, data, 10, NETCONN_COPY);
netconn_write(conn, text, sizeof(text), NETCONN_NOCOPY);
/*可以修改的数据 */
for(i = 0; i < 10; i++)
data[i] = 10 - i;
/*记下连接 take down the connection conn */
netconn_close(conn);
}
```

16.0.25 netconn send()

摘要

```
int netconn send(struct netconn *conn struct netbuf *buf)
```

描述

使用UDP连接conn发送数据到netbuf buf。在netbuf里的数据不能太大，因为没有使用IP分段。数据不能超过流出网络接口的最大传输单元（MTU）。因为当前的数据无法通过一个恰当的途径保存，netbuf不能存储大于1000字节的数据。

无校验使得不论发送的数据非常小或是非常大，netbuf可能给出不确定的结果。

例 这个例子说明如何通过IP地址10.10.0.1的远程主机的UDP端口7000发送UDP数据报。

```
int
main()
{
struct netconn *conn;
struct netbuf *buf;
struct ip_addr addr;
char *data;
char text[] = "A static text";
int i;
/*  创建一个新的连接*/
conn = netconn_new(NETCONN_UDP);
/*  设置远程主机的IP地址*/
addr.addr = htonl(0x0a000001);
/*  连接远程主机*/
netconn_connect(conn, &addr, 7000);
/*  创建一新的netbuf */
buf = netbuf_new();
data = netbuf_alloc(buf, 10);
/*创建一些立即数*/
for(i = 0; i < 10; i++)
data[i] = i;
/*  发送立即数*/
netconn_send(conn, buf);
netbuf_ref(buf, text, sizeof(text));
/*  发送text */
netconn_send(conn, buf);
/*分配connection  和  netbuf */
netconn_delete(conn);
netconn_delete(buf);
```

16.0.26 netconn close()

摘要

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

描述

关闭连接conn。

17 BSD socket 库

这一段给出了一个简单的BSD socket API使用LWIP API的应用。这个例子仅仅提供一个参考，不是为实际程序使用的。仅是个示例无错误处理。

而且这个应用例子也不支持BSD socket API的select()和poll()函数，因为在LWIP API里没有任何函数可以在这个应用中使用。为了应用这些函数，BSD socket不得不直接和LWIP堆栈通信而不使用API。

17.1 socket的表示方法

在BSD socket API里套接字被表示为普通的文件描述符，文件描述符是一个独一无二的标识，表示文件或网络连接的整型数。在这个BSD socket API应用中，套接字由netconn结构体来表示。因为BSD sockets表示符是整型数，所以变量netconn放在sockets[]数组里，BSD socket

标识索引放在数组中。

17.2 分配socket

17.2.1 socket()函数的调用

调用socket（）函数来分配一BSD套接字。函数socket（）的参数用来规定何中套接字的类型。考虑到这个socket API的应用仅于网络套接字有关，所以在这里仅支持一种套接字类型。而且仅仅UDP (SOCK DGRAM)或TCP (SOCK STREAM)套接字可以使用。

```
int
socket(int domain, int type, int protocol)
{
struct netconn *conn;
int i;
/* 创建netconn */
switch(type) {
case SOCK_DGRAM:
conn = netconn_new(NETCONN_UDP);
break;
case SOCK_STREAM:
conn = netconn_new(NETCONN_TCP);
break;
}
/*寻找 sockets[]表中的空元素*/
for(i = 0; i < sizeof(sockets); i++) {
if(sockets[i] == NULL) {
sockets[i] = conn;
return i;
}
}
return -1;
}
```

17.3 连接安装

BSD socket API 对设置连接的调用函数与最低程度的API的连接设置函数十分相似.那些调用函数的执行主要包括将socket的整数表示转化为在最低程度的API中使用的连接抽象化.

17.3.1 bind()函数调用

bind()调用函数将BSD socket绑定到本地(本机)地址上.在调用bind()时,本地IP地址和端口号将被指定.bind()函数与在 lwIP API中的netconn_bind()函数十分相似.

```
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 connect()函数调用
函数connect()的应用和bind()函数一样简单。（见bind()函数）

```
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 listen()函数调用
listen() 调用函数 相当于lwIP API中的netconn_listen()函数,它只能在TCP连接时使用.唯一的不同是BSD socket API允许应用程序指明等待连接队列的大小.这对lwIP是不可能的并且等待事件的参数被忽略了.

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

17.3.4 accept()函数调用
accept()调用函数被用来等待TCP socket口上的输入连接.事前,这个TCP socket口通过调用listen()已经被设置成监听状态.对accept()调用一直被阻塞,直到与远程主机建立连接.监听的自变量是结果参数,这些参数通过调用accept()来设置.他们充满了远程主机的地址.
当新的连接建立时,lwIP函数netconn accept()因为新的连接将返回连接句柄.在远程主机的IP地址和端口被装满后,新的socket口的标识符被分配和返回.

```
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);
/*取得远程主机的IP地址和端口号 */
netconn_peer(conn, &addr, &port);
addr->sin_addr = *addr;
addr->sin_port = port;
/*分配新的套接字标识*/
for(i = 0; i < sizeof(sockets); i++) {
```

```
if(sockets[i] == NULL) {
sockets[i] = newconn;
return i;
}
}
return -1;
}
```

17.4 发送和接收数据

17.4.1 send()函数调用

在BSD socket API中,send()调用函数在UDP 和 TCP两种连接中被用来发送数据.在调用send()前,数据接收器必须被设置成正在使用connect().对UDP期间,send()调用函数类似lwIP API中的netconn send()函数,但是因为lwIP API需要应用程序明确地分配缓冲区,所以一个缓冲区必须在send()调用函数里分配和释放.因此,先分配缓冲区再把数据拷贝进缓冲区.lwIP API中的netconn send()函数不能在TCP连接中使用,因此在TCP连接中使用netconn write()来执行send()功能.在BSD socket API中,应用程序在调用send()后可以直接修改发送的数据,因此NETCONN COPY 标志位 被传递给netconn write()那样数据被装入了堆栈的内部缓冲区.

```
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:
/*创建一缓冲区 */
buf = netbuf_new();
/* 使得缓冲区指针指向要发送的数据*/
netbuf_ref(buf, data, size);
/*发送数据*/
netconn_send(sock->conn.udp, buf);
/*分配缓冲区*/
netbuf_delete(buf);
break;
case NETCONN_TCP:
netconn_write(conn, data, size, NETCONN_COPY);
break;
}
return size;
}
```

17.4.2 sendto() 和 sendmsg()函数调用

sendto() 和 sendmsg()调用函数与send()调用函数类似,但是在参数调用中他们允许应用程序指定数据接收器.同样,sendto() 和 sendmsg()仅能在UDP连接中使用.实现这功能要使用etconn connect()来设置数据包接收器.如果以前socket口被连接,因此必须重设远程IP地址和端口号.不包括sendmsg()的执行不包括.

```
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];
/*当前连接建立获取peer */
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);
/*复位远程连接的地址和端口号*/
netconn_connect(conn, addr, port);
}
```

17.4.3 write()函数调用

在BSD socket API中,write()调用函数通过连接来发送数据并且能在UDP 和 TCP连接中使用.对TCP连接来说,这个函数直接映射到lwIP API函数netconn write().对UDP连接来说,BSD socket函数write()等价于send()函数.

```
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 recv()和read()函数调用

在BSD socket API,通过函数recv() and read()调用来连接socket接收数据.可以用在TCP和UDP连接上.通过调用函数recv()来传递很多flags.在这里这些都用不着,因为flags参数是被忽略的.

如果接收到的消息大于提供的存储区间,多于的数据将自动丢弃.

```
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);
/* 拷贝接收缓冲区的内容到提供的内存指针 men指向的区域 */
netbuf_copy(buf, mem, len);
netbuf_delete(buf);
/* 如果接收数据长度大于len,数据被丢弃并返回len,否则返回接收到的实际数据的长度.*/
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

函数recvfrom()和recvmsg()的调用和recv()类似,区别在于前者可以通过调用函数获取数据发送者的IP地址和端口号.

recvmsg()的应用这里没有给出.

```
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);
/*拷贝接收缓冲区的内容到提供的内存指针 men指向的区域*/
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);
/*如果接收数据长度大于len,数据被丢弃并返回len,否则返回接收到的实际数据的长度.*/
if(len > buflen) {
return buflen;
} else {
return len;
}
}
```