

LwIP 协议栈分析

larkguo@gmail.com

2007-05-26

目录

1	简介.....	3
2	Architecture.....	3
3	数据结构.....	6
3.1	Pbuf.....	6
3.2	Netbuf.....	7
3.3	Netconn.....	8
3.4	udp_pcb.....	10
3.5	tcp_seg.....	10
3.6	tcp_pcb.....	11
3.7	tcp_pcb_listen.....	13
3.8	Mbox.....	13
3.8.1	tcpip_msg.....	14
3.8.2	api_msg.....	14
3.9	Socket.....	15
3.10	定时.....	16
4	API.....	17
4.1	TYPE 1.....	19
4.1.1	netconn_bind.....	20
4.1.2	netconn_listen.....	20
4.1.3	netconn_close.....	20
4.2	TYPE 2.....	20
4.2.1	netconn_connect.....	21
4.3	TYPE 3.....	22
4.3.1	netconn_send.....	22
4.3.2	netconn_write.....	22
4.4	TYPE 4.....	23
4.4.1	netconn_accept.....	23
4.4.2	netconn_recv.....	23
4.5	Server.....	24

4.6	Client.....	24
5	流程.....	25
5.1	TCP.....	25
5.1.1	TCP Send.....	25
5.1.2	TCP Receive.....	26
5.2	UDP.....	27
6	主线程.....	27

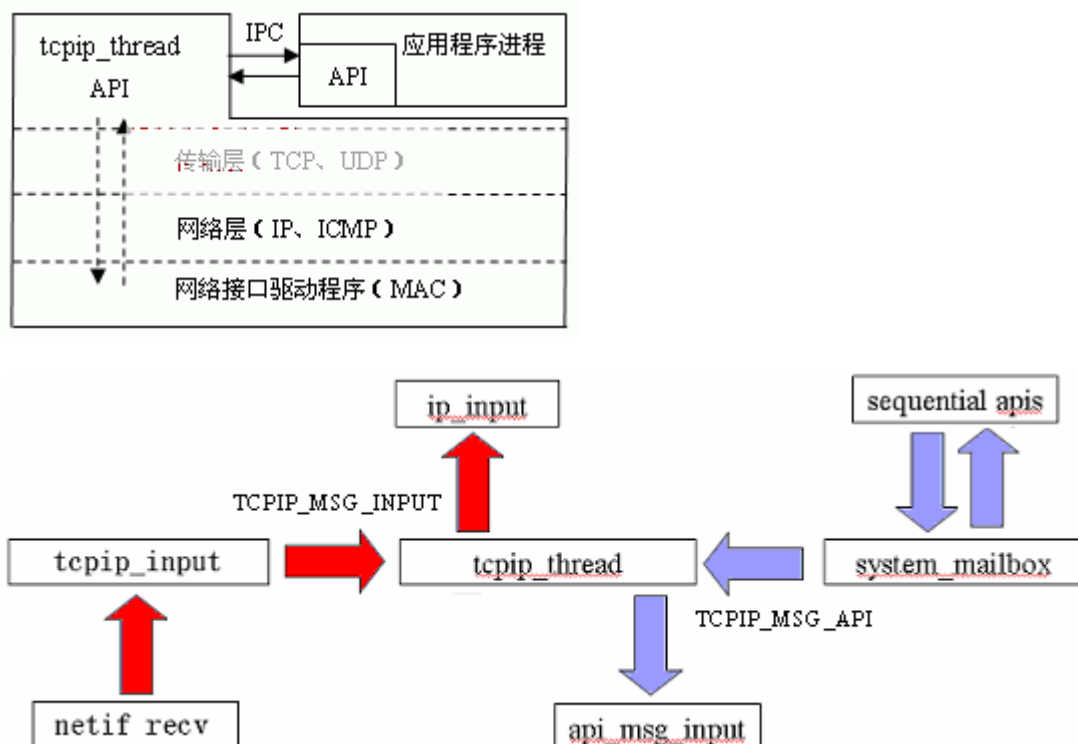
1 简介

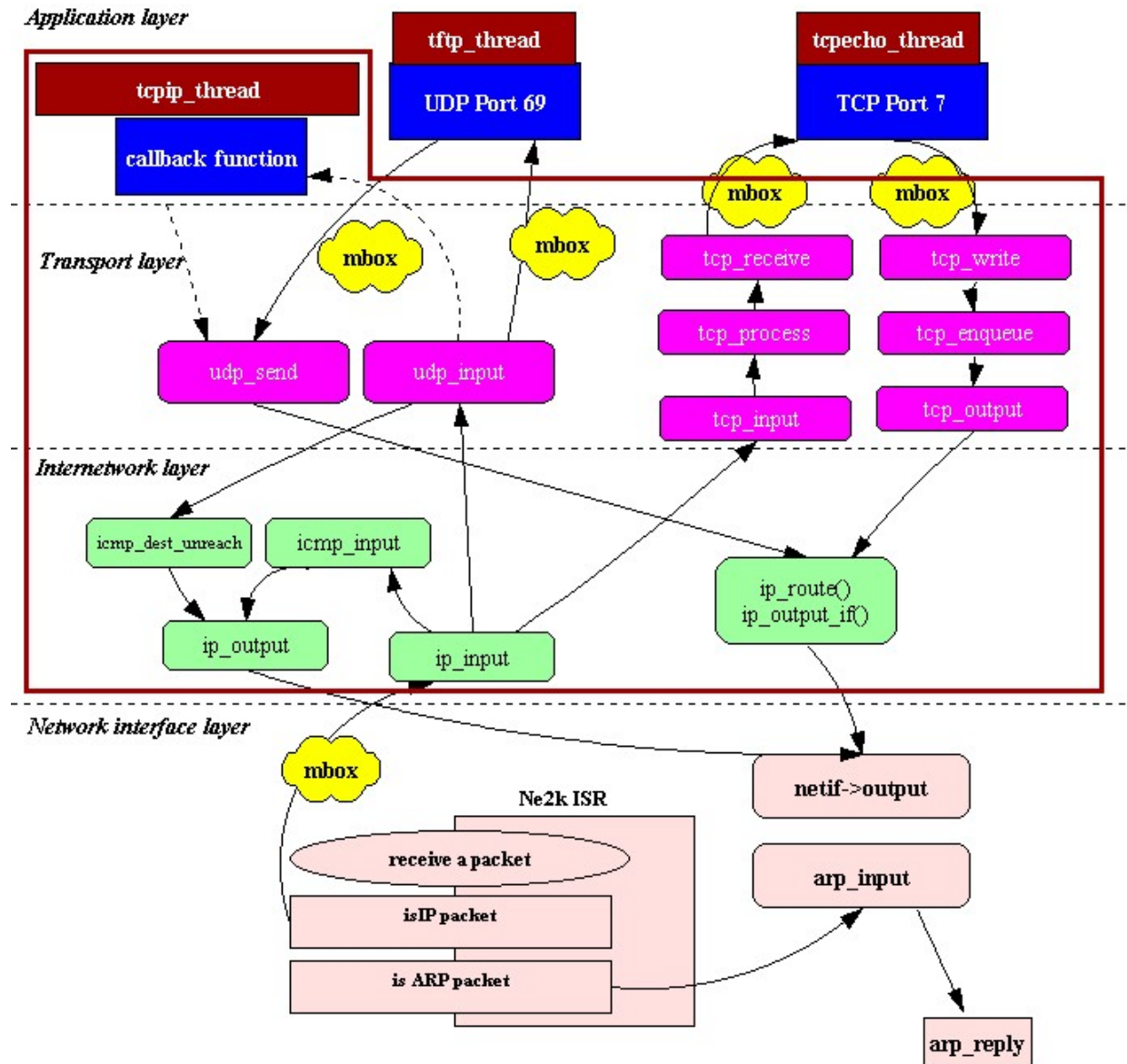
LwIP 是瑞士计算机科学院（Swedish Institute of Computer Science）的 Adam Dunkels 等开发的一套用于嵌入式系统的开放源代码 TCP/IP 协议栈。Lwip 既可以移植到操作系统上，又可以在无操作系统的情况下独立运行。

LwIP 的特性如下：

- 支持多网络接口下的 IP 转发
- 支持 ICMP 协议
- 包括实验性扩展的 UDP（用户数据报协议）
- 包括阻塞控制，RTT 估算和快速恢复和快速转发的 TCP（传输控制协议）
- 提供专门的内部回调接口（Raw API）用于提高应用程序性能
- 可选择的 Berkeley 接口 API（多线程情况下）
- 在最新的版本中支持 ppp
- 新版本中增加了 IP fragment 的支持。
- 支持 DHCP 协议,动态分配 ip 地址。
- 支持 IPv6

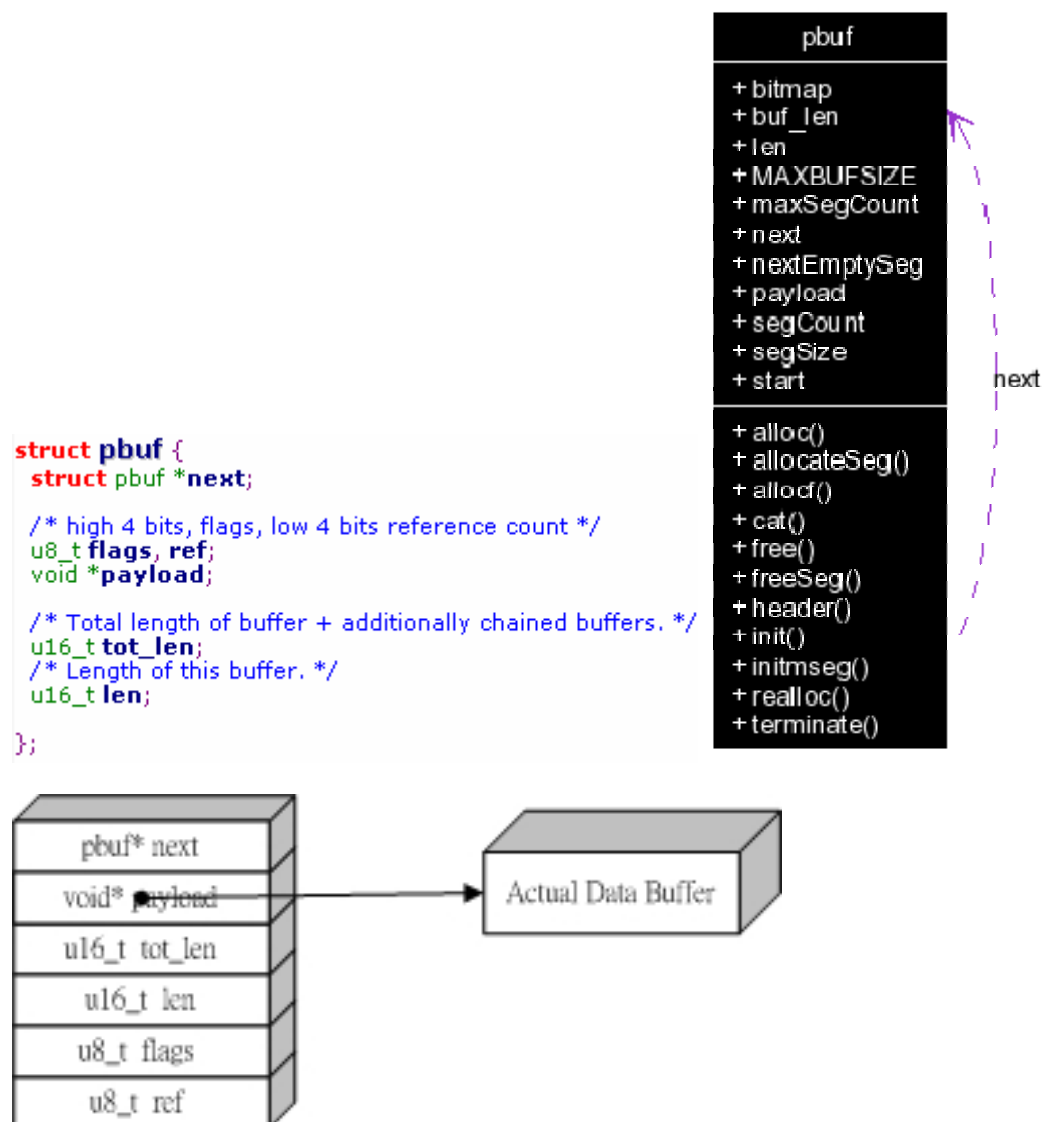
2 Architecture





3 数据结构

3.1 Pbuf



Pbuf是lwIP包的内部表示，被设计为最小化栈的特殊需要。Pbufs类似于BSD实现中的mbufs。Pbuf结构支持为包内容动态分配内存和让包数据驻留在静态内存中。Pbufs能被一个称为pbuf链的链接到一个链表中，以至一个包能跨越多个pbufs。

Pbufs 有三种类型:PBUF_RAM,PBUF_ROM 和 PBUF_POOL。图 1 表示 PBUF_RAM 类型，包含有存在内存中由 pbuf 子系统管理的包数据。图 2 显示了一个 pbuf 链表，第 1 个是 PBUF_RAM 类型，第 2 个是 PBUF_ROM 类型，意味着它包含有不被 pbuf 子系统管理的内存数据。图 3 描述了 PBUF_POOL，其包含有从固定大小 pbuf 池中分配来的 pbuf。一个 pbuf 链可以包含多个不同类型的 pbuf。

这三种类型有不同的用处。PBUF_POOL类型主要由网络设备驱动使用，因为分配单个pbuf 快速且适合中断句柄使用。PBUF_ROM类型由应用程序发送那些在应用程序内存空间中的

数据时使用。这些数据不会在pbuf递交给TCP/IP栈后被修改，因此这个类型主要用于当数据在ROM中时。PBUF_ROM中指向数据的头部被存在链表中其前一个PBUF_RAM类型的pbuf中，如图 2 所示。

PBUF_RAM 类型也用于应用程序发送动态产生的数据。这情况下，pbuf 系统不仅为应用程序数据分配内存，也为将指向（prepend）数据的头部分配内存。如图 1 所示。Pbuf 系统不能预知哪种头部将指向（prepend）那些数据，只假定最坏的情况。头部的大小在编译时确定。

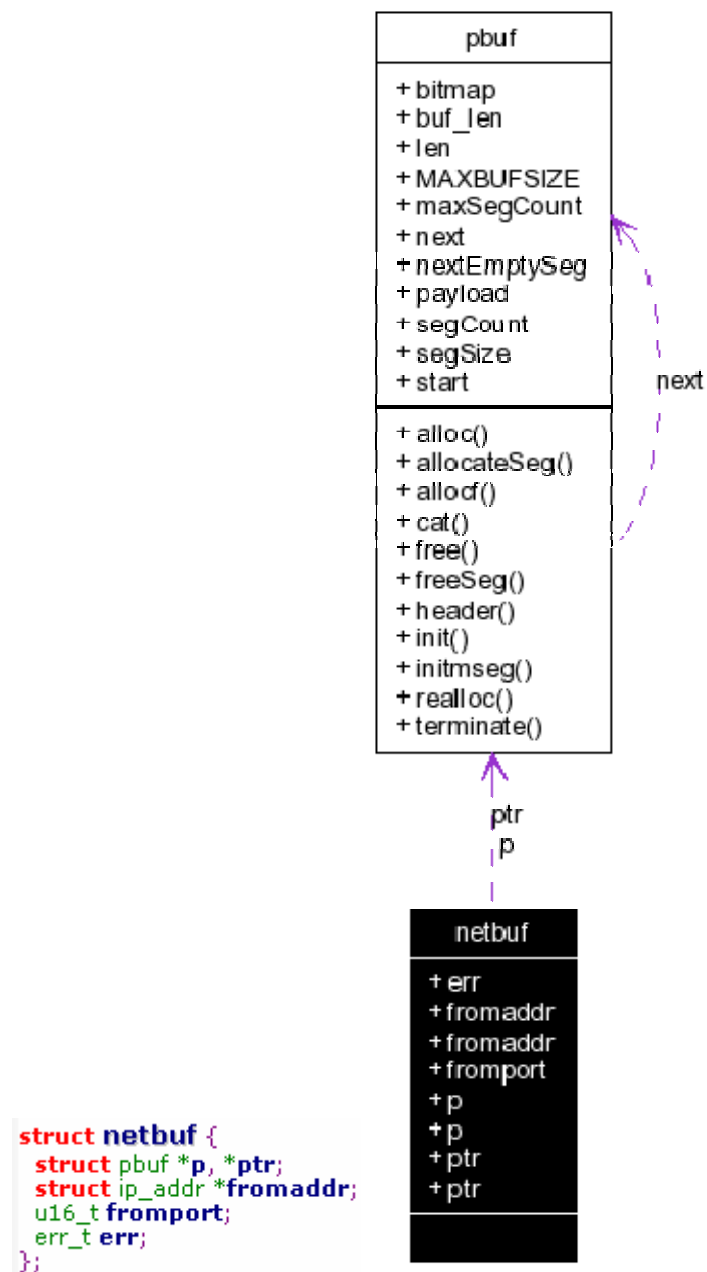
本质上，进来的 pbuf 是 PBUF_POOL 类型，而出去的 pbuf 是 PBUF_ROM 或 PBUF_RAM 类型。

从图 1，图 2 可以看出pbuf的内部结构。Pbuf结构包含有两个指针，两个长度字段，一个标志字段，和一个参考计数。Next字段指向[统一](#)链表中的下一个pbuf。有效载荷指针指向该pbuf中数据的起始点。Len字段包含有该pbuf数据内同的长度。Tot_len字段是当前pbuf和所有链表接下来中的len字段值的总和。简单说，tot_len字段是len字段及下一个pbuf中tot_len字段值的总和。Flags字段表示pbuf类型而ref字段包含一个参考计数。Next和payload字段是本地指针，其大小由处理器体系结构决定。两个长度字段是 16 位无符号整数，而flags和ref字段都是 4 比特大小。Pbuf的总大小决定于使用的处理器体系结构。在 32 位指针和 4 字节校正的体系结构上，总大小是 16 字节，而在 16 位指针和 1 自己校正的体系结构上，总大小是 9 字节。

Pbuf模块提供了操作pbuf的函数。Pbuf_alloc()可以分配前面提到的三种类型的pbuf。Pbuf_ref()增加引用计数,pbuf_free()释放分配的空间，它先减少引用计数，当引用计数为 0 时就释放pbuf。Pbuf_realloc([收缩](#))空间以使pbuf只占用刚好的空间保存数据。Pbuf_header()调整 payload 指针和长度字段，以使一个头部指向 pbuf 中的数据。Pbuf_chain() 和 pbuf_dechain()用于链表化pbuf。

3.2 Netbuf

描述网络缓存的数据类型



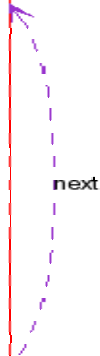
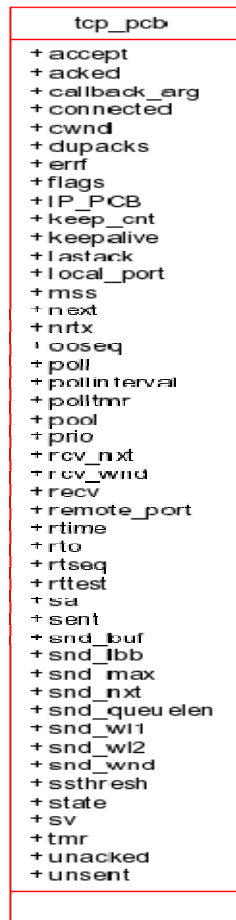
3.3 Netconn

描述网络连接的数据类型
与[socket](#)一一对应


```
enum netconn_type {
    NETCONN_TCP,
    NETCONN_UDP,
    NETCONN_UDPLITE,
    NETCONN_UDPNOCHKSUM
};
```

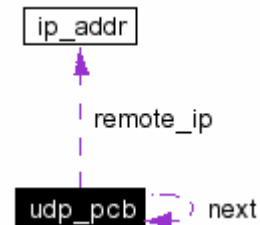
```
enum netconn_state {
    NETCONN_NONE,
    NETCONN_WRITE,
    NETCONN_ACCEPT,
    NETCONN_RECV,
    NETCONN_CONNECT,
    NETCONN_CLOSE
};
```

```
struct netconn {
    enum netconn_type type;
    enum netconn_state state;
    union {
        struct tcp_pcb *tcp;
        struct udp_pcb *udp;
    } pcb;
    err_t err;
    sys_mbox_t mbox;
    sys_mbox_t rcvmbox;
    sys_mbox_t acceptmbox;
    sys_sem_t sem;
};
```



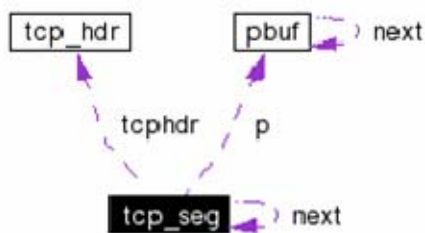
3.4 udp_pcb

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

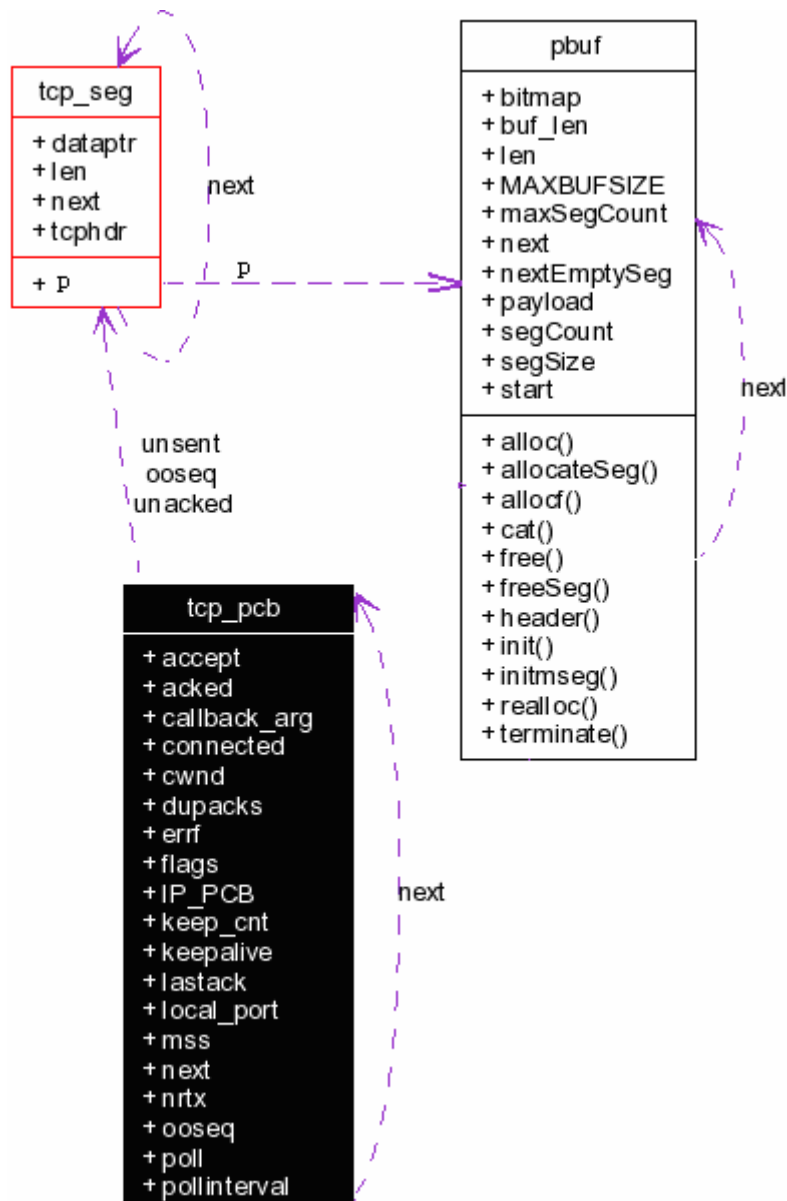


3.5 tcp_seg

```
/* This structure is used to represent TCP segments. */  
struct tcp_seg {  
    struct tcp_seg *next; /* used when putting segments on a queue */  
    struct pbuf *p; /* buffer containing data + TCP header */  
    void *dataptr; /* pointer to the TCP data in the pbuf */  
    u16_t len; /* the TCP length of this segment */  
    struct tcp_hdr *tcphdr; /* the TCP header */  
};
```



3.6 tcp_pcb



```

/* the TCP protocol control block */
struct tcp_pcb {
    struct tcp_pcb *next; /* for the linked list */
    enum tcp_state state; /* TCP state */
    void *callback_arg;

    /* Function to call when a listener has been connected. */
    err_t (*accept)(void *arg, struct tcp_pcb *newpcb, err_t err);

    struct ip_addr local_ip;
    u16_t local_port;

    struct ip_addr remote_ip;
    u16_t remote_port;

    /* receiver variables */
    u32_t rcv_nxt; /* next seqno expected */
    u16_t rcv_wnd; /* receiver window */

    u16_t tmr; /* Timers */
    u8_t rtime; /* Retransmission timer. */
    u16_t mss; /* maximum segment size */

    u8_t flags;
    #define TF_ACK_DELAY 0x01 /* Delayed ACK. */
    #define TF_ACK_NOW 0x02 /* Immediate ACK. */
    #define TF_INFR 0x04 /* In fast recovery. */
    #define TF_RESET 0x08 /* Connection was reset. */
    #define TF_CLOSED 0x10 /* Connection was successfully closed. */
    #define TF_GOT_FIN 0x20 /* Connection was closed by the remote end. */

    /* RTT estimation variables. */
    u16_t rttest; /* RTT estimate in 500ms ticks */
    u32_t rtseq; /* sequence number being timed */
    s32_t sa, sv;

    u16_t rto; /* retransmission time-out */
    u8_t nrtx; /* number of retransmissions */

    /* fast retransmit/recovery */
    u32_t lastack; /* Highest acknowledged seqno. */
    u8_t dupacks;

```

```

/* congestion avoidance/control variables */
u16_t cwnd;
u16_t ssthresh;

/* sender variables */
u32_t snd_nxt, /* next seqno to be sent */
snd_max, /* Highest seqno sent. */
snd_wnd, /* sender window */
snd_wl1, snd_wl2,
snd_lbb;

u16_t snd_buf; /* Available buffer space for sending. */
u8_t snd_queuelen;

/* Function to be called when more send buffer space is available. */
err_t (* sent)(void *arg, struct tcp_pcb *pcb, u16_t space);
u16_t acked;
/* Function to be called when (in-sequence) data has arrived. */
err_t (* rcv)(void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err);
struct pbuf *rcv_data;
/* Function to be called when a connection has been set up. */
err_t (* connected)(void *arg, struct tcp_pcb *pcb, err_t err);
/* Function which is called periodically. */
err_t (* poll)(void *arg, struct tcp_pcb *pcb);
/* Function to be called whenever a fatal error occurs. */
void (* errf)(void *arg, err_t err);

u8_t polltmr, pollinterval;

/* These are ordered by sequence number: */
struct tcp_seg *unsent; /* Unsent (queued) segments. */
struct tcp_seg *unacked; /* Sent but unacknowledged segments. */
#if TCP_QUEUE_OOSEQ
struct tcp_seg *ooseq; /* Received out of sequence segments. */
#endif /* TCP_QUEUE_OOSEQ */
} ? end tcp_pcb ? ;

```

3.7 tcp_pcb_listen

```

struct tcp_pcb_listen {
    struct tcp_pcb_listen *next; /* for the linked list */

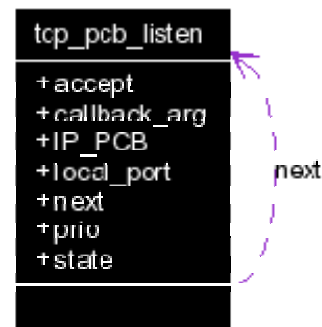
    enum tcp_state state; /* TCP state */

    void *callback_arg;

    /* Function to call when a listener has been connected. */
    void (* accept)(void *arg, struct tcp_pcb *newpcb);

    struct ip_addr local_ip;
    u16_t local_port;
};

```



3.8 Mbox

```

struct sys_mbox {
    u16_t first, last;
    void *msgs[SYS_MBOX_SIZE];
    struct sys_sem *mail;
    struct sys_sem *mutex;
};

```

3.8.1 tcpip_msg

```
enum tcpip_msg_type {
    TCPIP_MSG_API,
    TCPIP_MSG_INPUT
};

struct tcpip_msg {
    enum tcpip_msg_type type;
    sys_sem_t *sem;
    union {
        struct api_msg *apimsg;
        struct {
            struct pbuf *p;
            struct netif *netif;
        } inp;
    } msg;
};
```

TCPIP_MSG_API 表示从 API(上层)来的包,包括 api_msg 中的 api_msg_type 所有类型的包;
TCPIP_MSG_INPUT 表示从 IP(下层)来的包.

3.8.2 api_msg

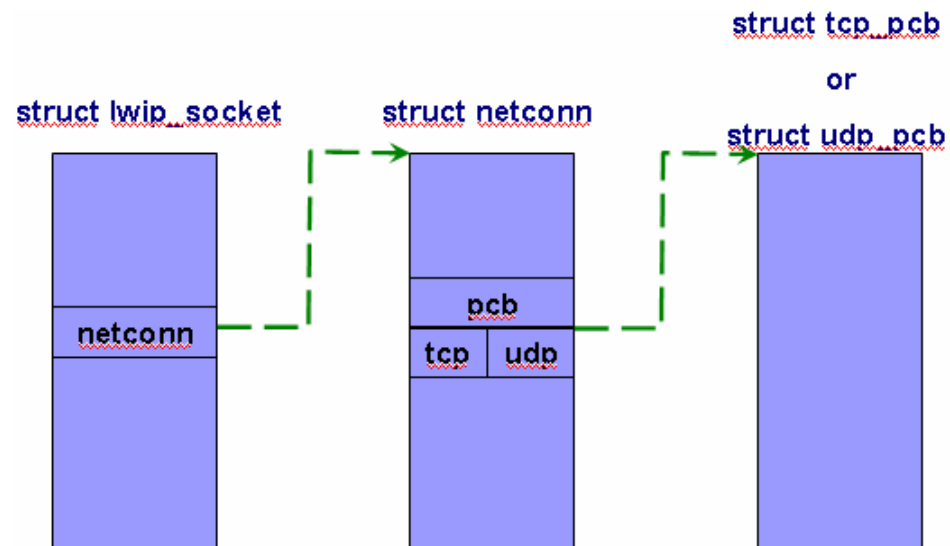
```
enum api_msg_type {
    API_MSG_NEWCONN,
    API_MSG_DELCONN,
    API_MSG_BIND,
    API_MSG_CONNECT,
    API_MSG_LISTEN,
    API_MSG_ACCEPT,
    API_MSG_SEND,
    API_MSG_RECV,
    API_MSG_WRITE,
    API_MSG_CLOSE,
    API_MSG_MAX
};

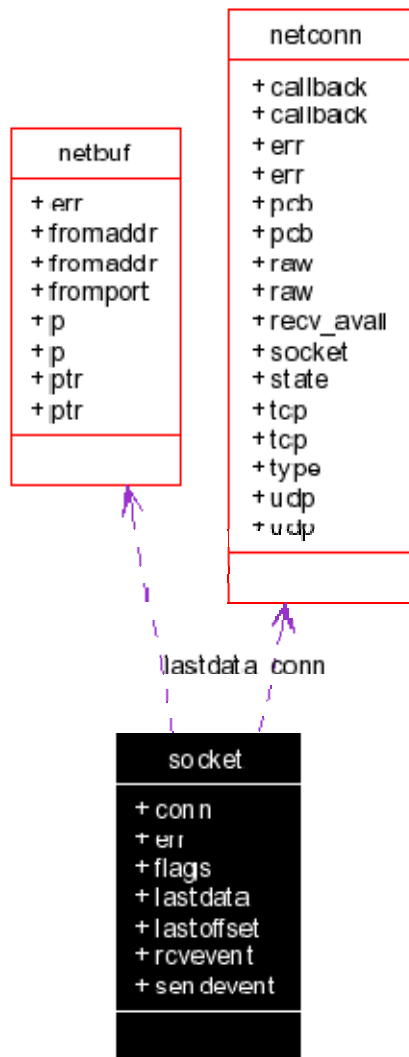
struct api_msg_msg {
    struct netconn *conn;
    enum netconn_type conntype;
    union {
        struct pbuf *p;
        struct {
            struct ip_addr *ipaddr;
            u16_t port;
        } bc;
        struct {
            void *dataptr;
            u16_t len;
            unsigned char copy;
        } w;
        sys_mbox_t mbox;
        u16_t len;
    } msg;
};

struct api_msg {
    enum api_msg_type type;
    struct api_msg_msg msg;
};
```

3.9 Socket

```
struct lwip_socket {  
  struct netconn *conn;  
  struct netbuf *lastdata;  
  u16_t lastoffset;  
};  
  
static struct lwip_socket sockets[NUM_SOCKETS];
```





3.10 定时

```

struct sys_timeout {
    struct sys_timeout *next;
    u16_t time;
    sys_timeout_handler h;
    void *arg;
};

struct sys_timeouts {
    struct sys_timeout *next;
};

```

在 tcp/ip 协议中很多时候都要用到定时,定时的实现也是 tcp/ip 协议栈中一个重要的部分.lwip 中定时事件的数据结构如下.

```

struct sys_timeout {
    struct sys_timeout *next;//指向下一个定时结构
    u32_t time;//定时时间
    sys_timeout_handler h;//定时时间到后执行的函数
    void *arg;//定时时间到后执行函数的参数.
};

```



```

struct sys_timeouts {
    struct sys_timeout *next;
};

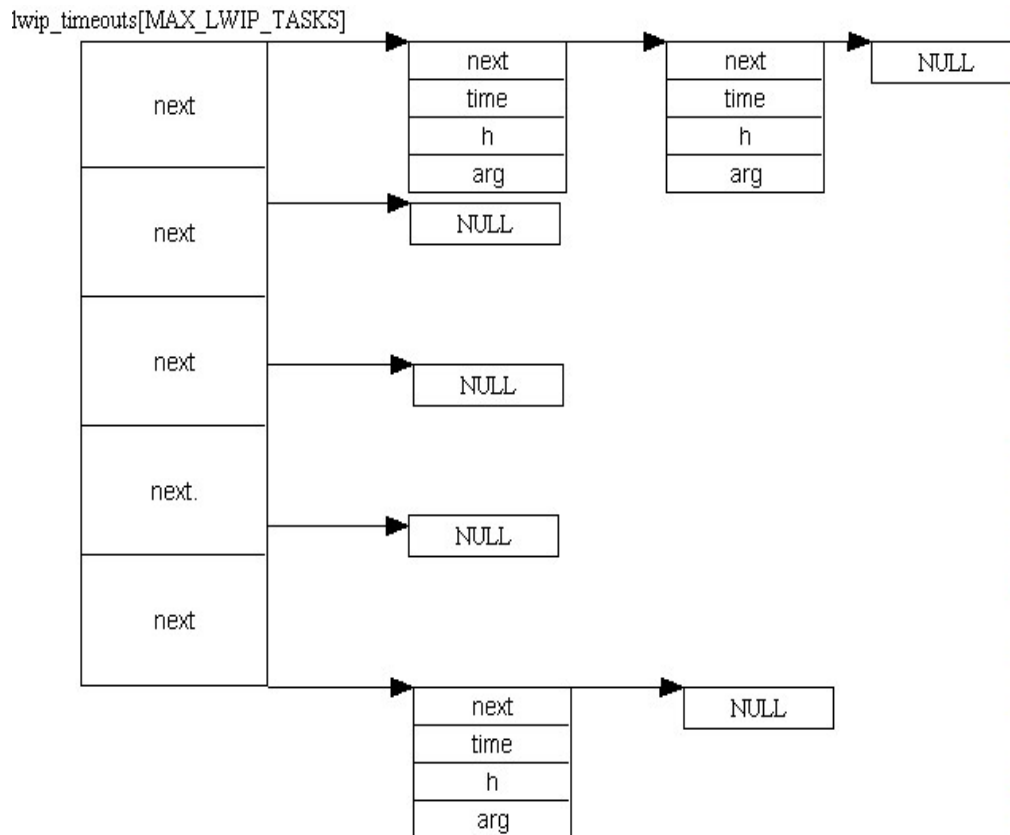
```

```

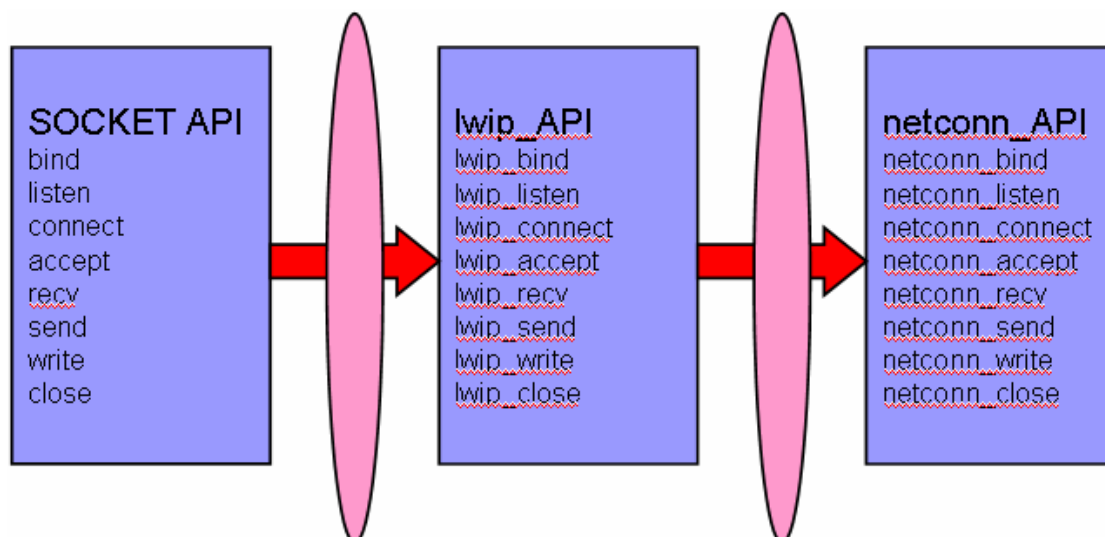
struct sys_timeouts lwip_timeouts[LWIP_TASK_MAX];

```

Lwip 中的定时事件表的结构如下图,每个和 tcp/ip 相关的任务的一系列定时事件组成一个单向链表.每个链表的起始指针存在 lwip_timeouts 的对应表项中.



4 API



	<u>netconn</u>	<u>netbuf</u>	<u>offset</u>
[0]			
[1]			
[2]			
[3]			
[4]			
[5]			
⋮			
[N-1]			
[N]			

TASK 1

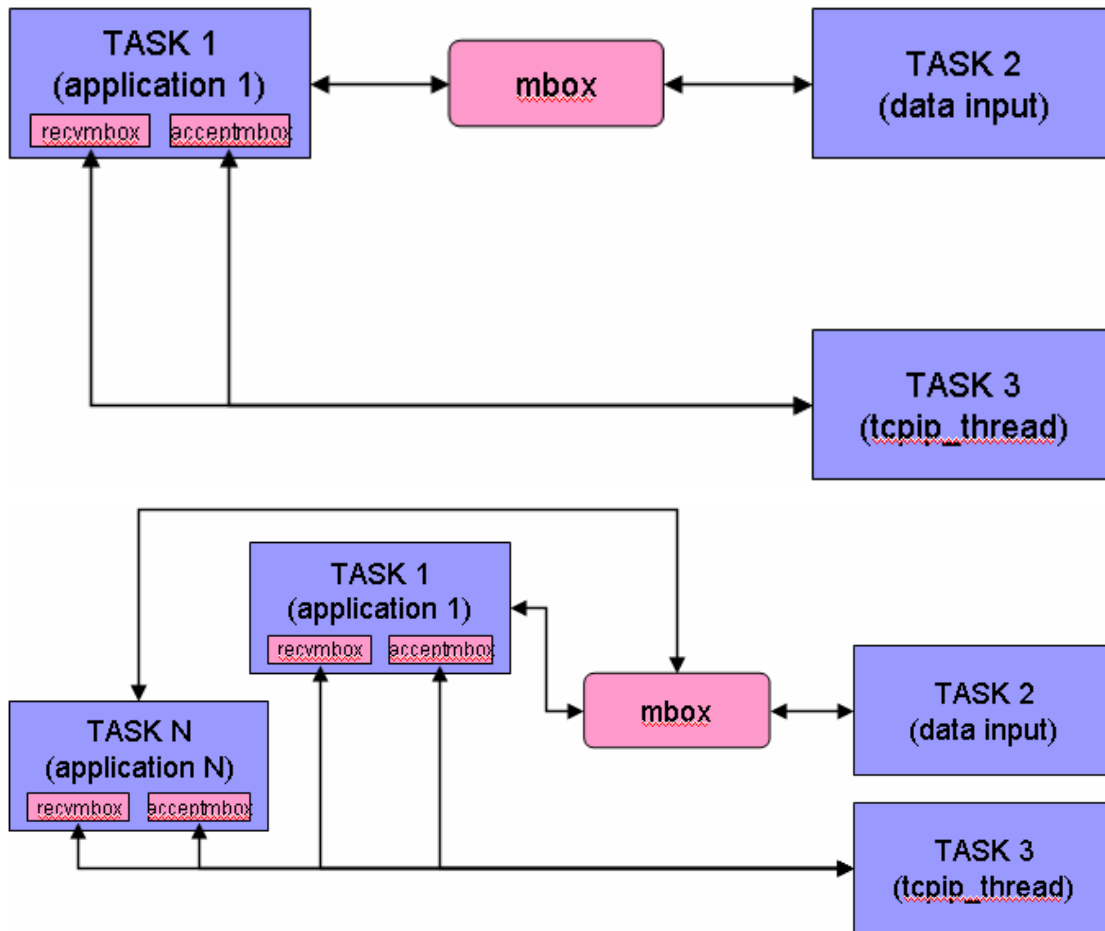
<u>application</u>
<u>socket API</u>
<u>lwip_bind ...</u>
<u>netconn_bind ...</u>

TASK 2

<u>tcpip_thread</u>
<u>api_msg_input</u>
<u>do_bind ...</u>

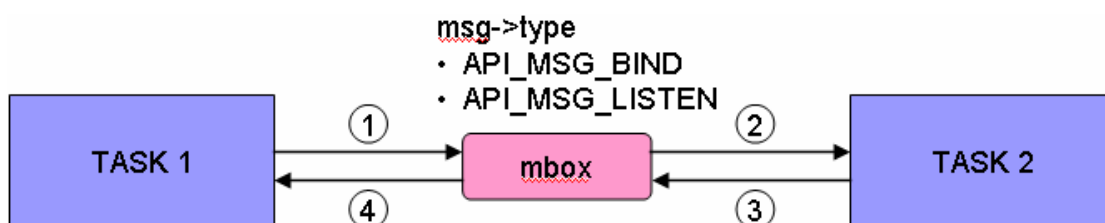
TASK 3

<u>driver</u>
<u>recv_buf_data</u>
<u>ethernetif_input</u>
<u>low_level_init</u>
<u>ip_input</u>
<u>tcp_input</u>
<u>tcp_process</u>

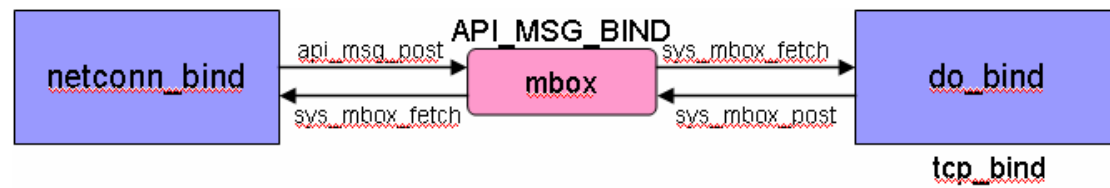


- TYPE 1
 - BIND, LISTEN, CLOSE
- TYPE 2
 - CONNECT
- TYPE 3
 - SEND, WRITE
- TYPE 4
 - ACCEPT, RECV

4.1 TYPE 1



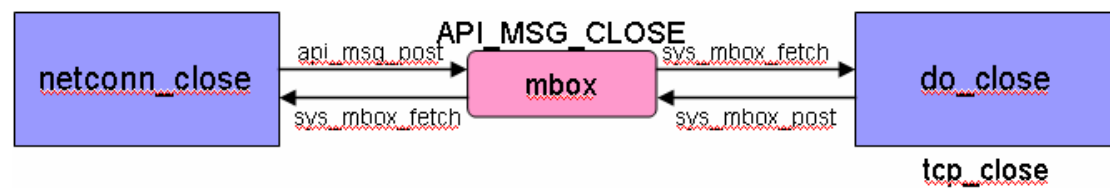
4.1.1 netconn_bind



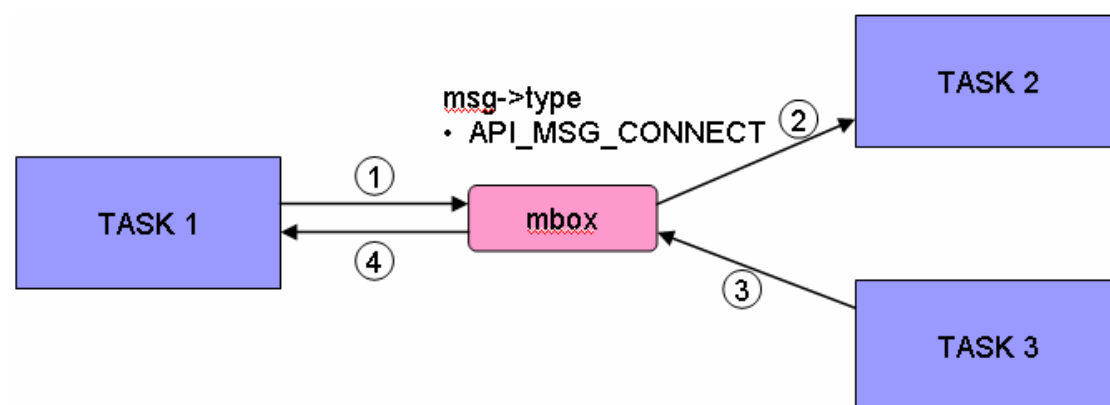
4.1.2 netconn_listen



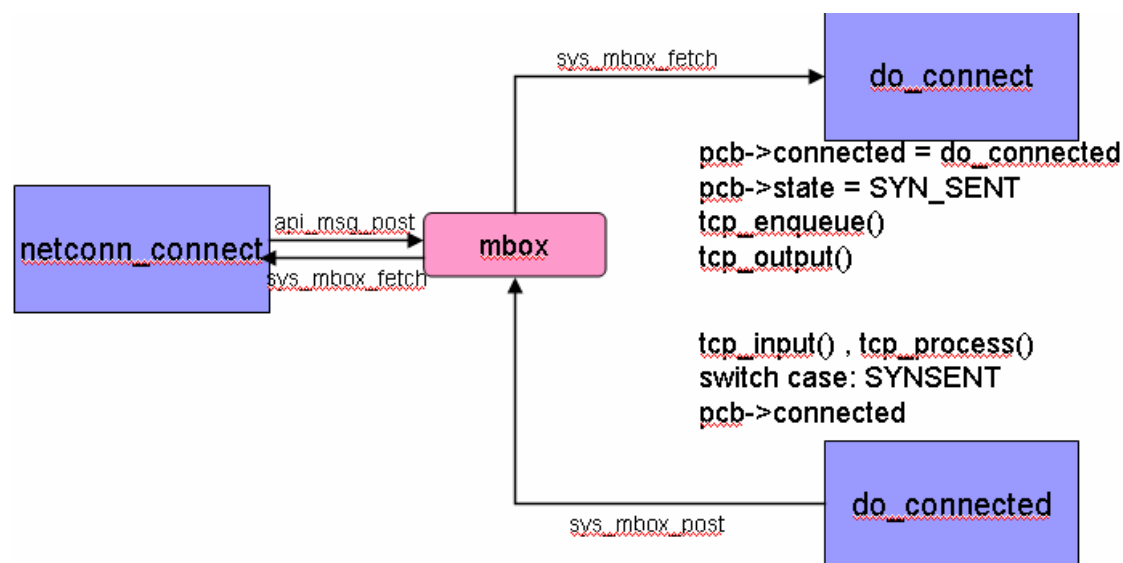
4.1.3 netconn_close



4.2 TYPE 2



4.2.1 netconn_connect



在 lwip 实现中，应用层调用 connect 函数进行主动连接，在该函数内部实际上是生成一个消息发送给 tcpip_thread 线程，在此消息里包含了 conn->mbox 信号量句柄，然后，该函数阻塞等待在此信号量上。一旦底层完成三次握手，连接成功，就会触发 TCP 已连接事件 TCP_EVENT_CONNECTED 回调 do_connect 释放此信号量，connect 随即退出阻塞。在应用层看来，connect 一直阻塞到连接成功，如果不成功就返回-1。

connect 运行在用户线程，实际连接运行在 tcpip_thread 线程，通过消息回调，使两个不同线程的函数建立了同步关系，虽然 TCP 协议时延动态范围很大，达到秒级，但这种消息驱动机制能很好地适应变化。

```

err_t
netconn_connect(struct netconn *conn, struct ip_addr *addr,
                u16_t port)
{
    struct api_msg *msg;

    if(conn == NULL)
        return ERR_VAL;

    if(conn->recvmbox == SYS_MBOX_NULL)
    {
        if((conn->recvmbox = sys_mbox_new()) == SYS_MBOX_NULL)
            return ERR_MEM;
    }

    if((msg = memp_malloc(MEMP_API_MSG)) == NULL)
        return ERR_MEM;

    msg->type = API_MSG_CONNECT;
    msg->msg.conn = conn;
    msg->msg.msg.bc.ipaddr = addr;
    msg->msg.msg.bc.port = port;
    api_msg_post(msg);

    /* 阻塞等待 do_connect 释放此信号量 */
    sys_mbox_fetch(conn->mbox, NULL);
    /* 释放 msg */
    memp_free(MEMP_API_MSG, msg);

    return conn->err;
} ? end netconn_connect ?

```

```

static err_t do_connected(void *arg, struct tcp_pcb *pcb, err_t err)
{
    struct netconn *conn;

    conn = arg;

    if(conn == NULL)
        return ERR_VAL;

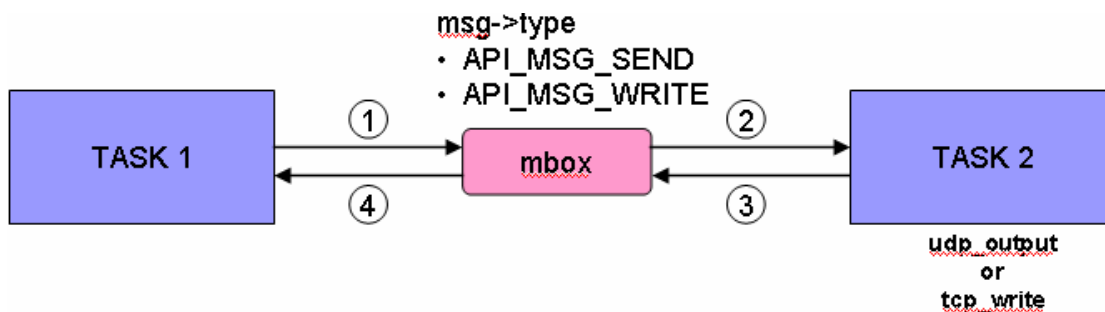
    conn->err = err;

    if(conn->type == NETCONN_TCP && err == ERR_OK)
        setup_tcp(conn);

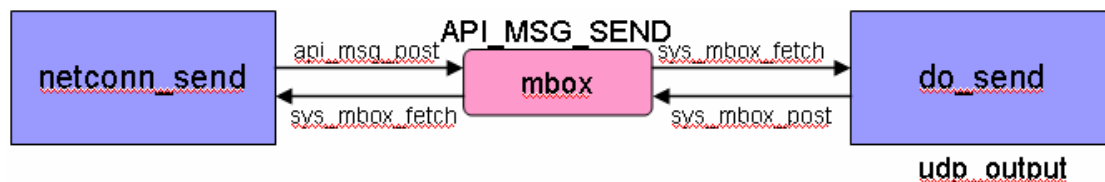
    /* 释放信号量,返回 netconn_connect */
    sys_mbox_post(conn->mbox, NULL);
    return ERR_OK;
}

```

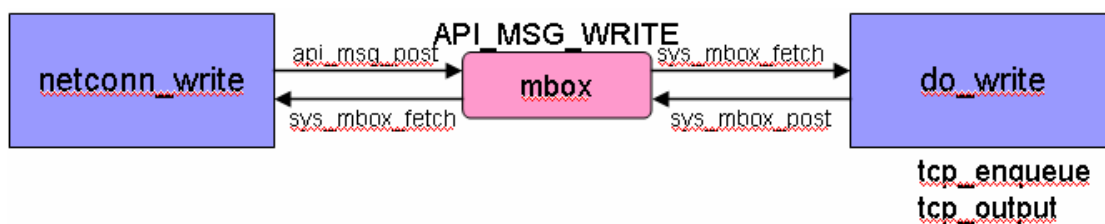
4.3 TYPE 3



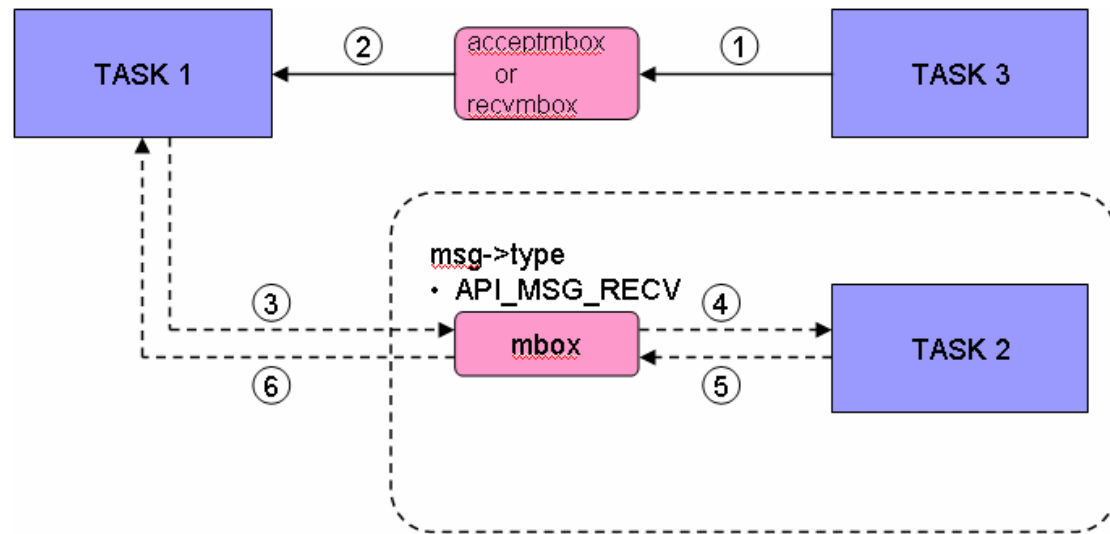
4.3.1 netconn_send



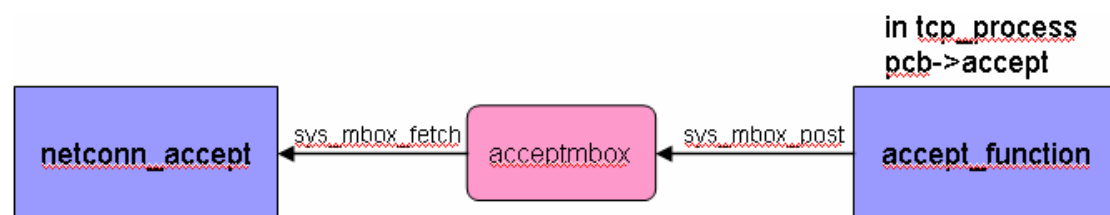
4.3.2 netconn_write



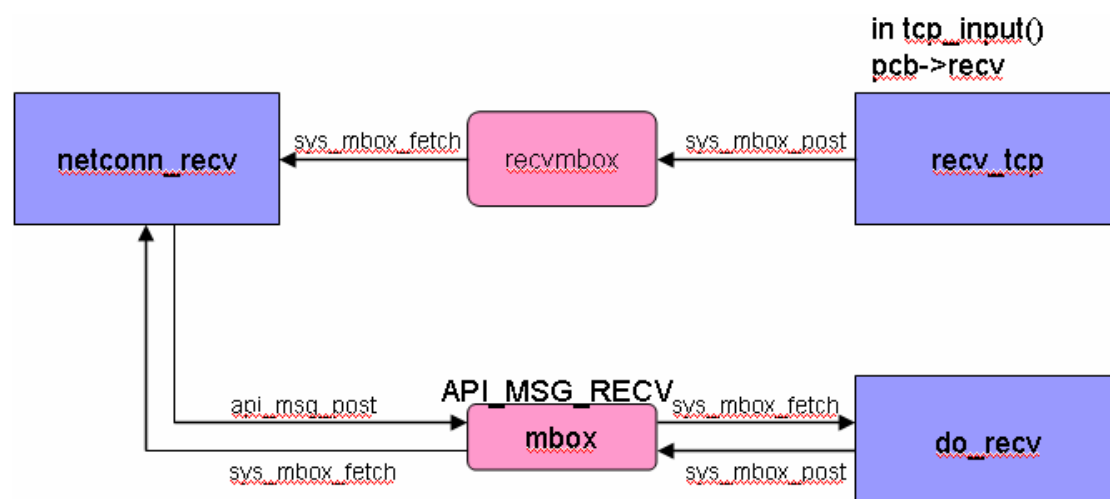
4.4 TYPE 4



4.4.1 netconn_accept

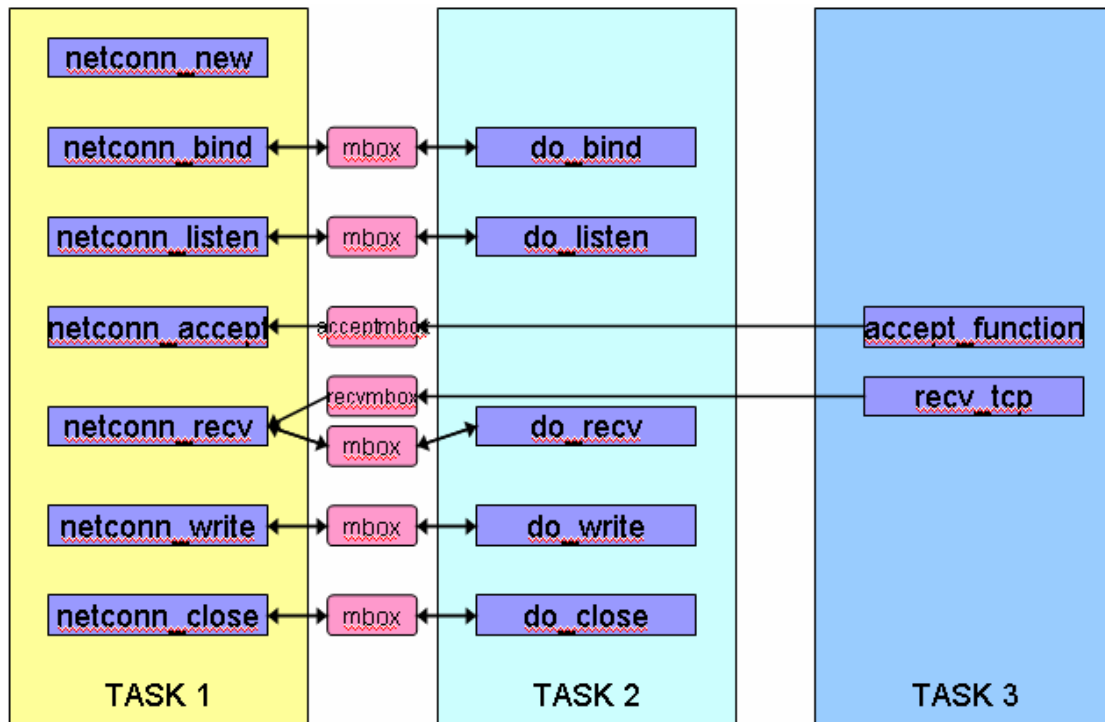


4.4.2 netconn_recv

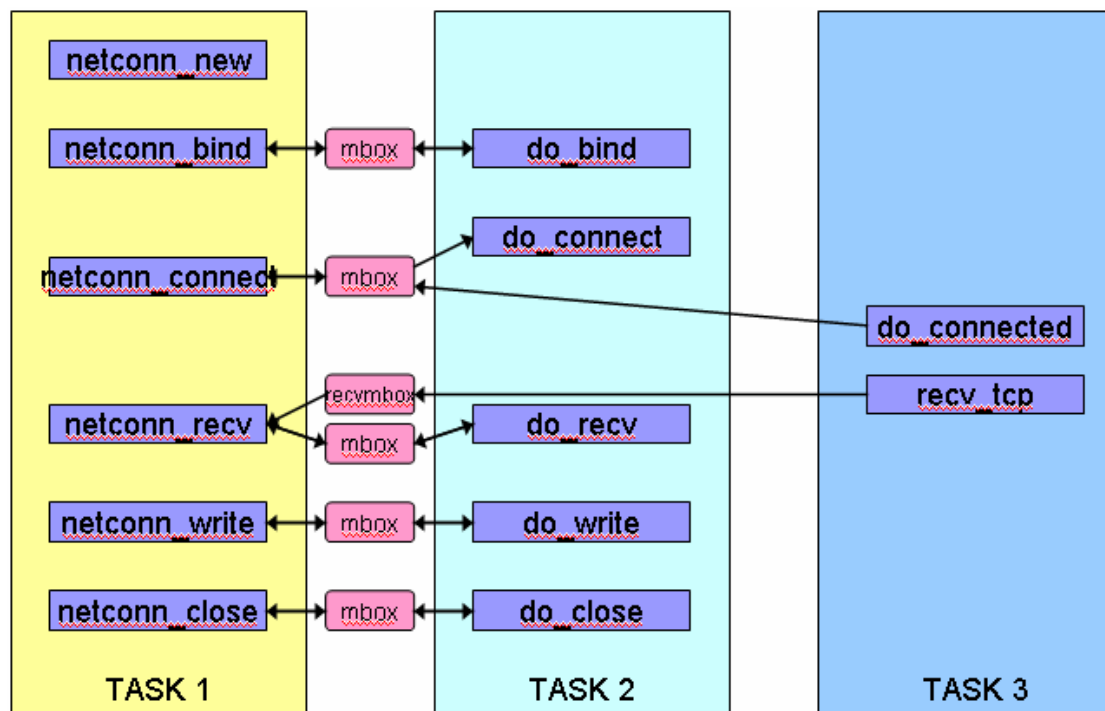


Let the stack know that we have taken the data

4.5 Server

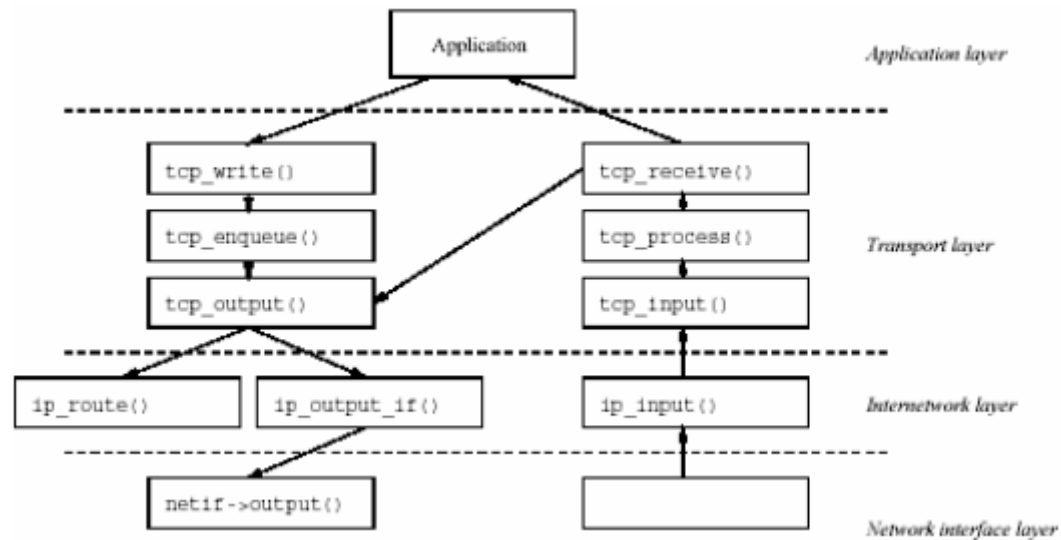


4.6 Client

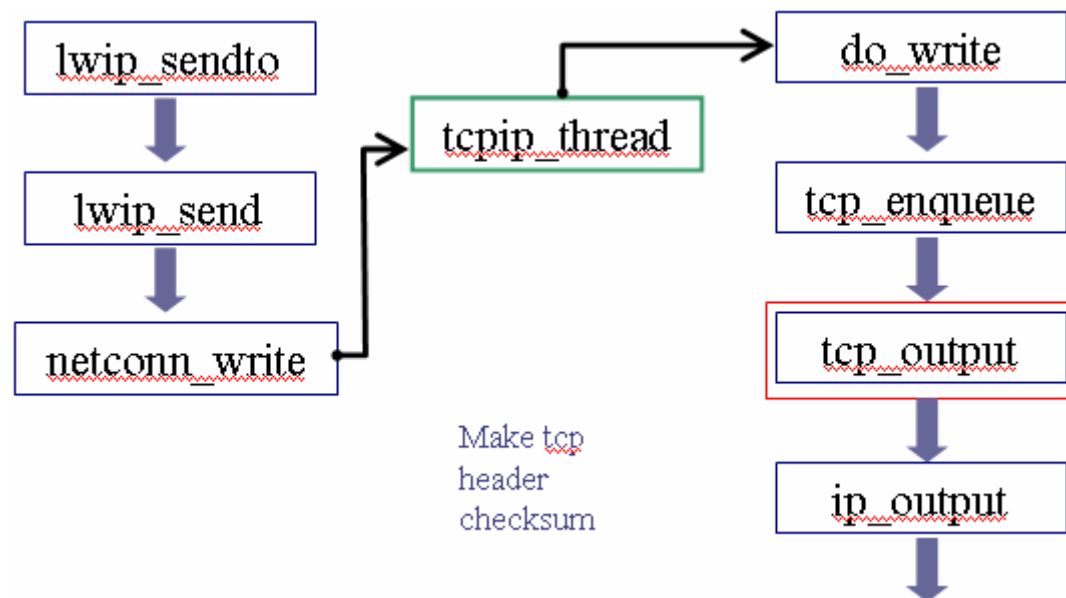


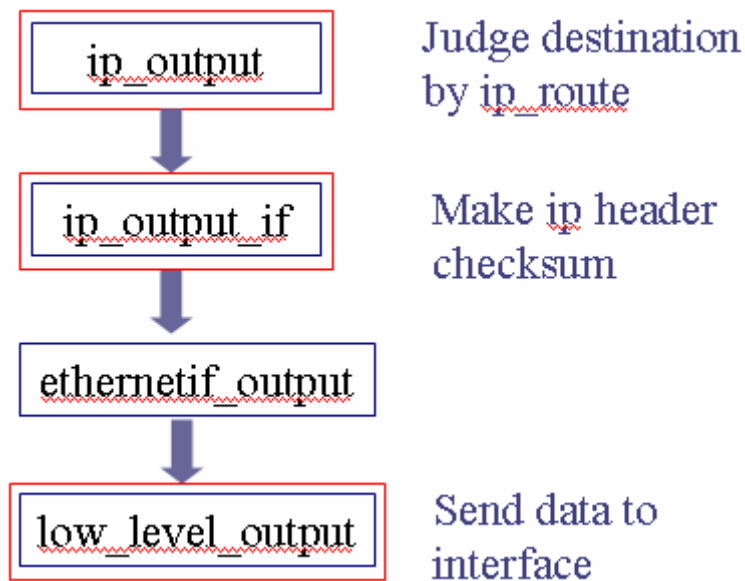
5 流程

5.1 TCP

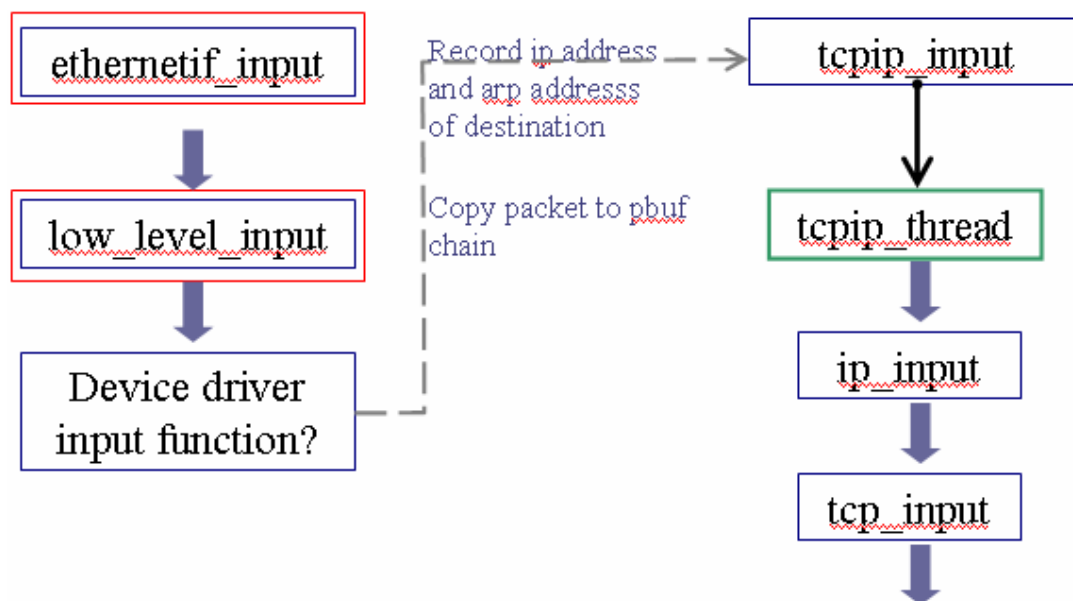


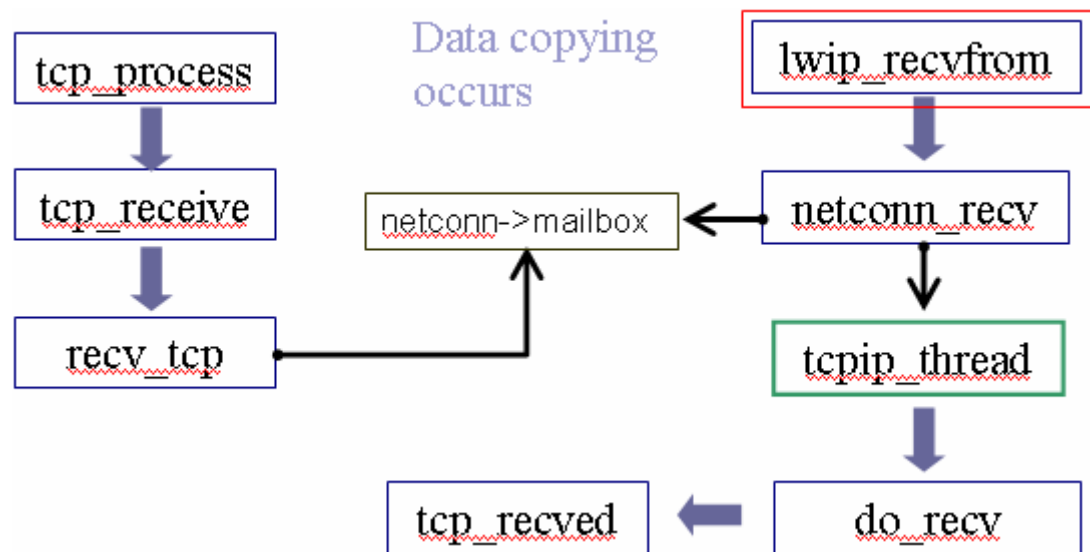
5.1.1 TCP Send



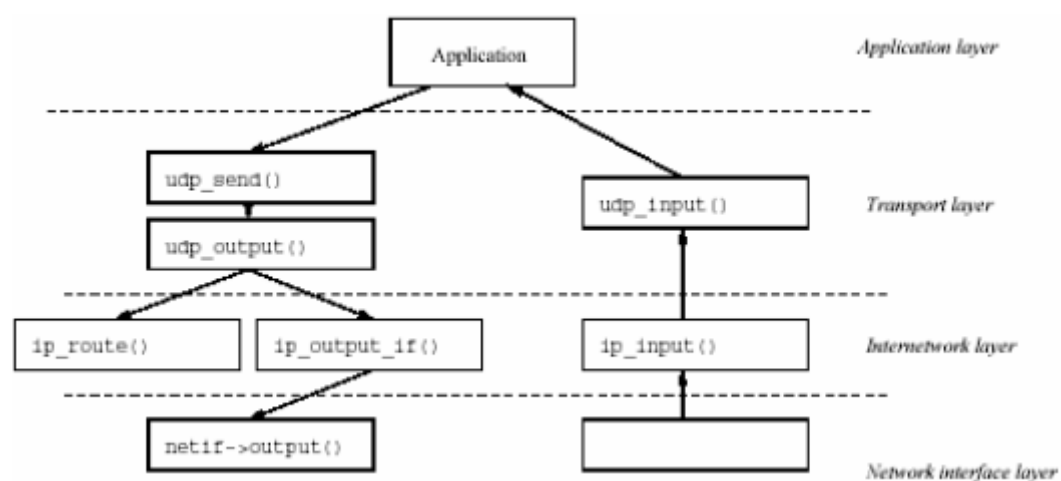


5.1.2 TCP Receive





5.2 UDP



6 主线程

tcpip_thread 是 LwIP 的主线程, 整个 tcp/ip 协议栈都在同一个任务(tcpip_thread)中.

```

static void
tcpip_thread(void *arg)
{
    struct tcpip_msg *msg;

    ip_init();
    udp_init();
    tcp_init();

    sys_timeout(TCP_TMR_INTERVAL, (sys_timeout_handler)tcpip_tcp_timer, NULL);

    if(tcpip_init_done != NULL) {
        tcpip_init_done(tcpip_init_done_arg);
    }

    while(1) {
        /* MAIN Loop */
        sys_mbox_fetch(mbox, (void *)&msg);
        switch(msg->type) {
            case TCPIP_MSG_API:
                DEBUGF(TCPIP_DEBUG, ("tcpip_thread: API message %p\n", msg));
                api_msg_input(msg->msg.apimsg);
                break;
            case TCPIP_MSG_INPUT:
                DEBUGF(TCPIP_DEBUG, ("tcpip_thread: IP packet %p\n", msg));
                ip_input(msg->msg.inp.p, msg->msg.inp.netif);
                break;
            default:
                break;
        }
        memp_freep(MEMP_TCPIP_MSG, msg);
    }
}
} ? end tcpip_thread ?

```