LWIP主进程工作

```
/* LWIP 协议模拟了 TCP/IP 协议的分层思想，
表面上看 LWIP 也是有分层思想的，但从实现上看， LWIP 只在一个进程内实现了各个层次
的所有工作。具体如下： LWIP 完成相关初始化后，会阻塞在一个邮箱上，等待接收数据进
行处理。这个邮箱内的数据可能来自底层硬件驱动接收到的数据包，也可能来自应用程序。
当在该邮箱内取得数据后， LWIP 会对数据进行解析，然后再依次调用协议栈内部上层相关
处理函数处理数据。处理结束后， LWIP 继续阻塞在邮箱上等待下一批数据。
*/
```

当数据在各层之间传递时，LWIP极力禁止数据的拷贝工作，因为会耗费大量的时间和内存

```
struct pbuf{
    struct pbuf *next;
    void *payload;  // 指向该pbuf管理的数据的起始地址：紧跟在pbuf结构后的RAM,或者ROM上的
某个地址
    u16_t tot_len;
    u16_t len;
    u8_t type;  // PBUF_RAM\PBUF_ROM\PBUF_PER\PBUF_POOL
    u8_t flags;
    u16_t ref;
};
```

应用层
传输层
网络层
链路层

LWIP没有在各层之间进行严格的划分，各层协议之间有交叉存取。

链路层

netif描述一个硬件网络接口

```
struct netif{
    struct netif *next;

    struct ip_addr ip_addr; // IP地址
    struct ip_addr netmask; // 子网掩码
    struct ip_addr gw;      // 网关地址

    err_t (*input)(struct pbuf *p, struct netif *inp);  // 调用这个函数从网卡取得一
个数据包
    err_t (*output)(struct netif *netif, struct pbuf* p, struct ip_addr
*ipaddr);   // IP层调用函数，向网卡发送一个数据包
    err_t (*linkoutput)(struct netif *netif, struct pbuf *p);   // ARP模块调用，向
网卡发送一个数据包
```

```
    void *state;     // 用于指向用户关心的网卡信息
    u8_t hwaddr_len;     // 硬件地址长度，对于以太网就是MAC地址长度，为6个字节
    u8_t hwaddr[NETIF_MAX_HWADDR_LEN];  //MAC地址
    u16_t mtu;  //  一次可以传输的最大字节数，以太网一般设1500
    u8_t flags; //  网卡状态信息标志位

    char name[2];    // 网络接口使用的设备驱动类型的种类
    u8_t num;        // 用来标识使用同种驱动类型的不同网络接口
};
```

看一个以太网网卡接口结构是这样被初始化，还有数据包是如何接收和发送的。先来看初始化过程，源码：

```
static struct netif enc28j60;                                   (1)
struct ip_addr ipaddr, netmask, gw;                             (2)

IP4_ADDR(&gw, 192,168,0,1);                                     (3)
IP4_ADDR(&ipaddr, 192,168,0,60);                                (4)
IP4_ADDR(&netmask, 255,255,255,0);                              (5)

netif_init();                                                   (6)
netif_add(&enc28j60, &ipaddr, &netmask, &gw, NULL, ethernetif_init, tcpip_input);   (7)
netif_set_default(&enc28j60);                                   (8)
netif_set_up(&enc28j60);                                        (9)
```

```
struct netif *
  netif_add(struct netif *netif, struct ip_addr *ipaddr, struct ip_addr *netmask,
  struct ip_addr *gw,
  void *state,
  err_t (* init)(struct netif *netif),
  err_t (* input)(struct pbuf *p, struct netif *netif))
{
  static u8_t netifnum = 0;
  netif->ip_addr.addr = 0;       //复位变量 enc28j60 中各字段的值
  netif->netmask.addr = 0;
  netif->gw.addr = 0;
  netif->flags = 0;               //该网卡不允许任何功能使能

  netif->state = state;          //指向用户关心的信息，这里为 NULL
  netif->num = netifnum++; //设置 num 字段，
  netif->input = input;          //如前所诉，input 函数被赋值

  netif_set_addr(netif, ipaddr, netmask, gw);   //设置变量 enc28j60 的三个地址

  if (init(netif) != ERR_OK) {       //用户自己的底层接口初始化函数
    return NULL;
  }

  netif->next = netif_list;         //将初始化后的节点插入链表 netif_list
  netif_list = netif;               // netif_list 指向链表头

  return netif;
}
```

上面的初始化函数调用了用户自己定义的底层接口初始化函数，这里为 ethernetif_init，再来看看它的源代码：

```
err_t    ethernetif_init(struct netif *netif)
{
    netif->name[0] = IFNAME0;       //初始化变量 enc28j60 的 name 字段
    netif->name[1] = IFNAME1;       // IFNAME 在文件外定义的，这里不必关心它的具体值
```

```
    netif->output = etharp_output;     //IP 层发送数据包函数
    netif->linkoutput = low_level_output;   // //ARP 模块发送数据包函数
    low_level_init(netif);              //底层硬件初始化函数
    return ERR_OK;
}
```

天，还有函数调用！low_level_init 函数就是与我们使用的硬件密切相关的函数了。

昨天说到 low_level_init 函数是与我们使用的与硬件密切相关初始化函数，看看：

```
static void low_level_init(struct netif *netif)
{
    netif->hwaddr_len = ETHARP_HWADDR_LEN; //设置变量 enc28j60 的 hwaddr_len 字段
    netif->hwaddr[0] = 'F';             //初始化变量 enc28j60 的 MAC 地址
    netif->hwaddr[1] = 'O';             //设什么地址用户自由发挥吧，但是不要与其他
    netif->hwaddr[2] = 'R';             //网络设备的 MAC 地址重复。
    netif->hwaddr[3] = 'E';
    netif->hwaddr[4] = 'S';
    netif->hwaddr[5] = 'T';

    netif->mtu = 1500;          //最大允许传输单元
                                //允许该网卡广播和 ARP 功能，并且该网卡允许有硬件链路连接
    netif->flags    = NETIF_FLAG_BROADCAST | \
                      NETIF_FLAG_ETHARP | NETIF_FLAG_LINK_UP;

    enc28j60_init(netif->hwaddr);     //与底层驱动硬件驱动程序密切相关的硬件初始化函数
    }
```

至此，终于变量 enc28j60 被初始化好了，而且它描述的网卡芯片 enc28j60 也被初始化好了，而且变量 enc28j60 也被链入链表 netif_list。

LWIP数据包收发函数框架

low_level_input low_level_output

LWIP应用系统包括三个进程：

- 上层应用程序进程
- LWIP协议栈进程
- 底层硬件数据包接收进程

目标MAC地址	源MAC地址	类型/长度	数据	校验
6字节	6字节	2字节(IP:0x0800/ARP:0x0806)	46-1500字节	4字节

ps: 最大帧长1518字节，最小64字节

eth_hdr描述以太网数据包包头14个字节

```
PACK_STRUCT_BEGIN    // 与编译器字对其相关的宏定义
    struct eth_hdr{
        PACK_STRUCT_FIELD(struct eth_addr dest);     // 目标MAC地址
        PACK_STRUCT_FIELD(struct eth_addr src);      // 源MAC地址
        PACK_STRUCT_FIELD(u16_t type);              // 类型
    }PACK_STRUCT_STRUCT;
PACK_STRUCT_END
```

大端模式：某个半字或字数据的高位字节存在内存的低地址端，低位字节存放在内存的高地址端
小端模式：某个半字或字数据的 高位字节存在内存的高地址端，低位字节存放在内存的低地址端

ARM处理器使用的是小端模式;；网络字节数据用的大端模式

最后需要注意的地方，netif->input 在结构 enc28j60 初始化时已经被设置为指向 tcpip_input 函数，所以实际上上面是调用 tcpip_input 函数往上层递交数据包。tcpip_input 属于 IP 层函数，从这里我们可以看出 LWIP 的一个很大的特点，即各层之间没有明显的界限划分。像前面所讲的那样，LWIP 协议栈进程完成初始化相关工作后，会阻塞在一个邮箱上等待数据包的输入，这就对了，tcpip_input 函数就是向这个邮箱发送一条消息，且该消息中包含了收到的数据包存储的地址。LWIP 协议栈进程从邮箱中取到该地址后就可以对数据包进行处理了。

ARP(地址解析协议)表

动态映射：将32位的IP地址转换为对应48位的MAC地址

核心：ARP缓存表；对缓存表的建立、更新、查询等操作

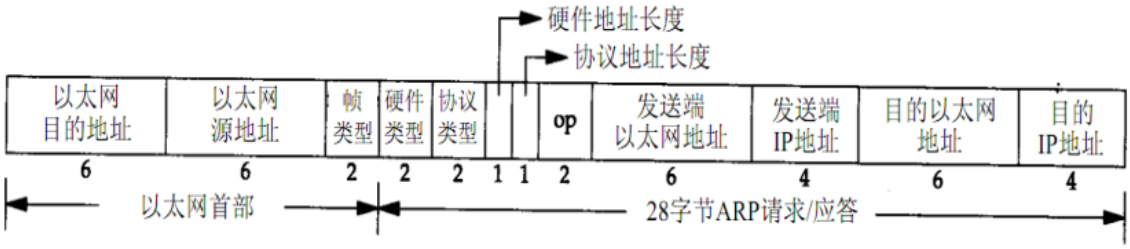
```
struct etharp_entry{    // 缓存表项
    #if ARP_QUEUEING
        struct etharp_q_entry *q;    //数据包缓冲队列指针
    #endif
```

```
        struct ip_addr ipaddr;      // 目标IP地址
        struct eth_addr ethaddr;    // MAC地址
        enum etharp_state state;    // 描述该entry的状态
        u8_t ctime;                 // 描述entry的时间信息。当大于规定的值时，该表项被内核删除
        struct netif *netif;        // 相应网络接口信息
};

enum etharp_state{
    ETHARP_STATE_EMPTY = 0,
    ETHARP_STATE_PENDING,    // 处于不稳定状态，会发送一个广播ARP请求到数据链路上，让对应IP地址的主机回应其MAC地址
    ETHARP_STATE_STABLE
};

static struct etharp_entry arp_table[ARP_TABLE_SIZE];        //数组方式创建ARP缓存表
```

ARP 数据包可以分为 ARP 请求数据包和 ARP 应答数据包，ARP 数据包到达底层链路时会被加上以太网数据包头发送出去，最终呈现在链路上的数据报头格式如下图，



以太网包头中的前两个字段是以太网的目的 MAC 地址和源 MAC 地址，在前面一章已经有讲解。目的地址为全 1 的特殊地址是广播地址。在 ARP 表项建立前，源主机只知道目的主机的 IP 地址，并不知道其 MAC 地址，所以在数据链路上，源主机只有通过广播的方式将 ARP 请求数据包发送出去。电缆上的所有以太网接口都要接收广播的数据包，并检测数据包是否是发给自己的，这点通过对照目的 IP 地址来实现，如果是发给自己的，目的主机需要回复一个 ARP 应答数据包给源主机，以告诉源主机自己的 MAC 地址。

两个字节长的以太网帧类型表示后面数据的类型。对于 ARP 请求或应答数据包来说，该字段的值为 0x0806，对于 IP 数据包来说，该字段的值为 0x0800。

以太网数据报头说完，来说 ARP 数据报头。

硬件类型字段表示硬件地址的类型，它的值为 1 即表示以太网 MAC 地址，长度为 6 个字节。协议类型字段表示要映射的协议地址类型。它的值为 0x0800 即表示要映射为 IP 地址。它的值与包含 IP 数据报的以太网数据帧头中的类型字段的值相同。

接下来的两个 1 字节的字段，硬件地址长度和协议地址长度分别指出硬件地址和协议地址的长度，以字节为单位。对于以太网上 ARP 请求或应答来说，它们的值分别为 6 和 4。

操作字段 op 指出四种操作类型，它们是 ARP 请求（值为 1）、ARP 应答（值为 2）、RARP 请求（值为 3）和 RARP 应答（值为 4），这里我们只关心前两个类型。这个字段

必需的，因为 ARP 请求和 ARP 应答的帧类型字段值是相同的。

接下来的四个字段是发送端的以太网 MAC 地址、发送端的 IP 地址、目的端的以太网 MAC 地址和目的端的 IP 地址。

注意，这里有一些重复信息：在以太网的数据帧报头中和 ARP 请求数据帧中都有发送端的以太网 MAC 地址。对于一个 ARP 请求来说，除目的端 MAC 地址外的所有其他的字段都有填充值。当目的主机收到一份给自己的 ARP 请求报文后，它就把自己的硬件地址填进去，然后将该请求数据包的源主机信息和目的主机信息交换位置，并把操作字段 op 置为 2，最后把该新构建的数据包发送回去，这就是 ARP 响应。

```
// 数据报头
struct etharp_hdr{
    PACK_STRUCT_FIELD(struct eth_hdr ethhdr); // 14 字节的以太网数据报头
    PACK_STRUCT_FIELD(u16_t hwtype); // 2 字节的硬件类型
    PACK_STRUCT_FIELD(u16_t proto); // 2 字节的协议类型
    PACK_STRUCT_FIELD(u16_t _hwlen_protolen); // 两个 1 字节的长度字段
    PACK_STRUCT_FIELD(u16_t opcode); // 2 字节的操作字段 op
    PACK_STRUCT_FIELD(struct eth_addr shwaddr); // 6 字节源 MAC 地址
    PACK_STRUCT_FIELD(struct ip_addr2 sipaddr); // 4 字节源 IP 地址
    PACK_STRUCT_FIELD(struct eth_addr dhwaddr); // 6 字节目的 MAC 地址
    PACK_STRUCT_FIELD(struct ip_addr2 dipaddr); // 4 字节目的 IP 地址
}PACK_STRUCT_STRUCT;
// PACK_STRUCT_FIELD()是防止编译器字对齐的宏定义
```

ARP表查询

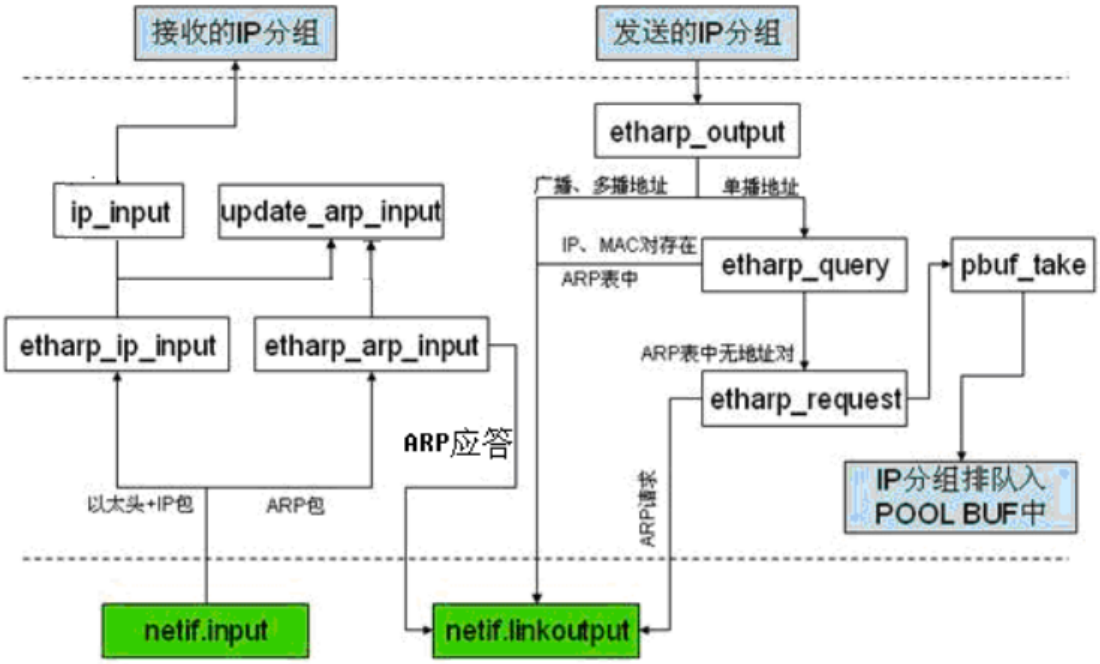
ARP攻击，针对是针对以太网地址解析协议（ARP）的一种攻击技术。在局域网中， ARP病毒收到广播的 ARP 请求包，能够解析出其它节点的 (IP, MAC) 地址, 然后病毒伪装为目的主机，告诉源主机一个假 MAC 地址，这样就使得源主机发送给目的主机的所有数据包都被病毒软件截取，而源主机和目的主机却浑然不知。 ARP 攻击通过伪造 IP 地址和 MAC 地址实现 ARP 欺骗，能够在网络中产生大量的 ARP 通信量使网络阻塞，攻击者只要持续不断的发出伪造的 ARP 响应包就能更改目标主机 ARP 缓存中的 IP-MAC 条目。 ARP 协议在设计时未考虑网络安全方面的特性，这就注定了其很容易遭受 ARP 攻击。黑客只要在局域网内阅读送上门来的广播 ARP 请求数据包，就能偷听到网内所有的 (IP, MAC)地址。而源节点收到 ARP 响应时，它也不会质疑，这样黑客很容易冒充他人。

```
 // 寻找一个匹配的ARP项或者创建一个新的ARP表项，并返回该表项的索引号
static s8_t find_entry(struct ip_addr *ipaddr, u8_t flags)
```

lwip 有一个比较巧妙的地方 ,LWIP 中有个全局的变量 etharp_cached_entry，它始终保存着上次用到的索引号，如
果这个索引恰好就是我们要找的内容，且索引的表项已经处于 stable 状态，那就直接返回这个索引号就完成了 。

```
// 向给定的 IP 地址发送一个数据包或者发送一个ARP请求，
err_t etharp_query(struct netif *netif, struct ip_addr *ipaddr, struct pbuf *q)
```

```
// 该函数用于更新 ARP 缓存表中的表项或者在缓存表中插入一个新的表项。
// 该函数会在收到一个 IP 数据包或 ARP 数据包后被调用。
static err_t update_arp_entry(struct netif *netif, struct ip_addr *ipaddr,
struct eth_addr *ethaddr, u8_t flags)
```



IP层

对于 IP 层主要讨论信息包的接收、分片数据包重装、信息包的发送和转发三个内容。IP 数据报头结构如下所示，其中，选项字段是可以没有的，所以通常的 IP 数据报头长度为 20 个字节。

IP 数据报头

```
struct ip_hdr {
    PACK_STRUCT_FIELD(u8_t _v_hl);       // 前三个字段：版本号、首部长度
      /* type of service */
    PACK_STRUCT_FIELD(u8_t _tos);        // 服务类型，描述该IP数据包急需的服务类型，如最小延时、最大吞吐量等
    PACK_STRUCT_FIELD(u16_t _len);       // 总长度，整个IP数据报，包括IP数据报头的总字节数
    PACK_STRUCT_FIELD(u16_t _id);        // 标识字段
    PACK_STRUCT_FIELD(u16_t _offset);    // 3 位标志和 13 位片偏移字段;用于IP数据包分片时使用
    #define IP_RF 0x8000 //
    #define IP_DF 0x4000 // 不分组标识位掩码
    #define IP_MF 0x2000 // 后续有分组到来标识位掩码
    #define IP_OFFMASK 0x1fff // 获取 13 位片偏移字段的掩码
    PACK_STRUCT_FIELD(u8_t _ttl); // TTL 字段,描述IP数据包最多能被转发的次数，为0时，一个ICMP报文会被返回至源主机
    PACK_STRUCT_FIELD(u8_t _proto);// 协议字段， 1 ICMP, 2 ICMP, 6 TCP, 17 UDP
    PACK_STRUCT_FIELD(u16_t _chksum); // 首部校验和字段,只针对IP首部做校验;它并不关心其内部数据在传输过程中出错与否对于数据的校验是上层协议负责的，如 ICMP、 IGMP、 TCP、 UDP 协议都会计算它们头部以及整个数据区的长度。
    PACK_STRUCT_FIELD(struct ip_addr src); // 源 IP 地址
    PACK_STRUCT_FIELD(struct ip_addr dest); // 目的 IP 地址
} PACK_STRUCT_STRUCT;
```

ICMP处理(Internet 控制报文协议)

控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。

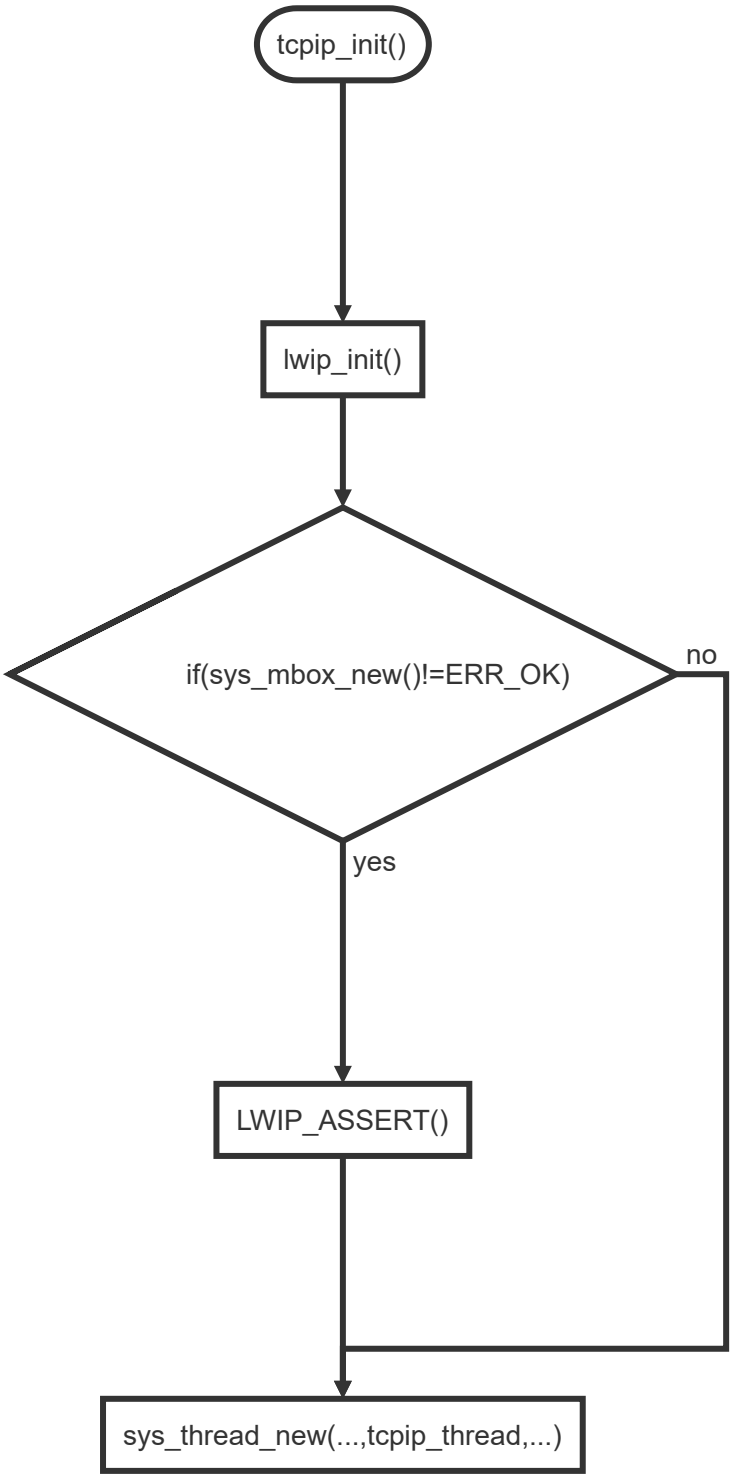
在以前讲解 IP 层 ip_input 函数时，已经三次涉及到了 ICMP 的东西，第一次在数据包转发过程中，需要将数据包的 TTL 值减 1，若此时 TTL 值变为 0 则用 icmp_time_exceeded 函数向源主机返回一份超时 ICMP 信息；还有两次是 ip_input 函数通过 IP 报文头部的协议字段值判断该数据包是交给哪个上层协议的，若是 ICMP 协议，则调用 icmp_input 函数；若没有一个协议能接受这个数据包，则调用 icmp_dest_unreach 函数向源主机返回一个协议不可达 ICMP 差错控制包。这里先讲解 icmp_time_exceeded 和 icmp_dest_unreach 函数是怎样发送 ICMP 信息包的。

void icmp_dest_unreach(struct pbuf *p, enum icmp_dur_type t)

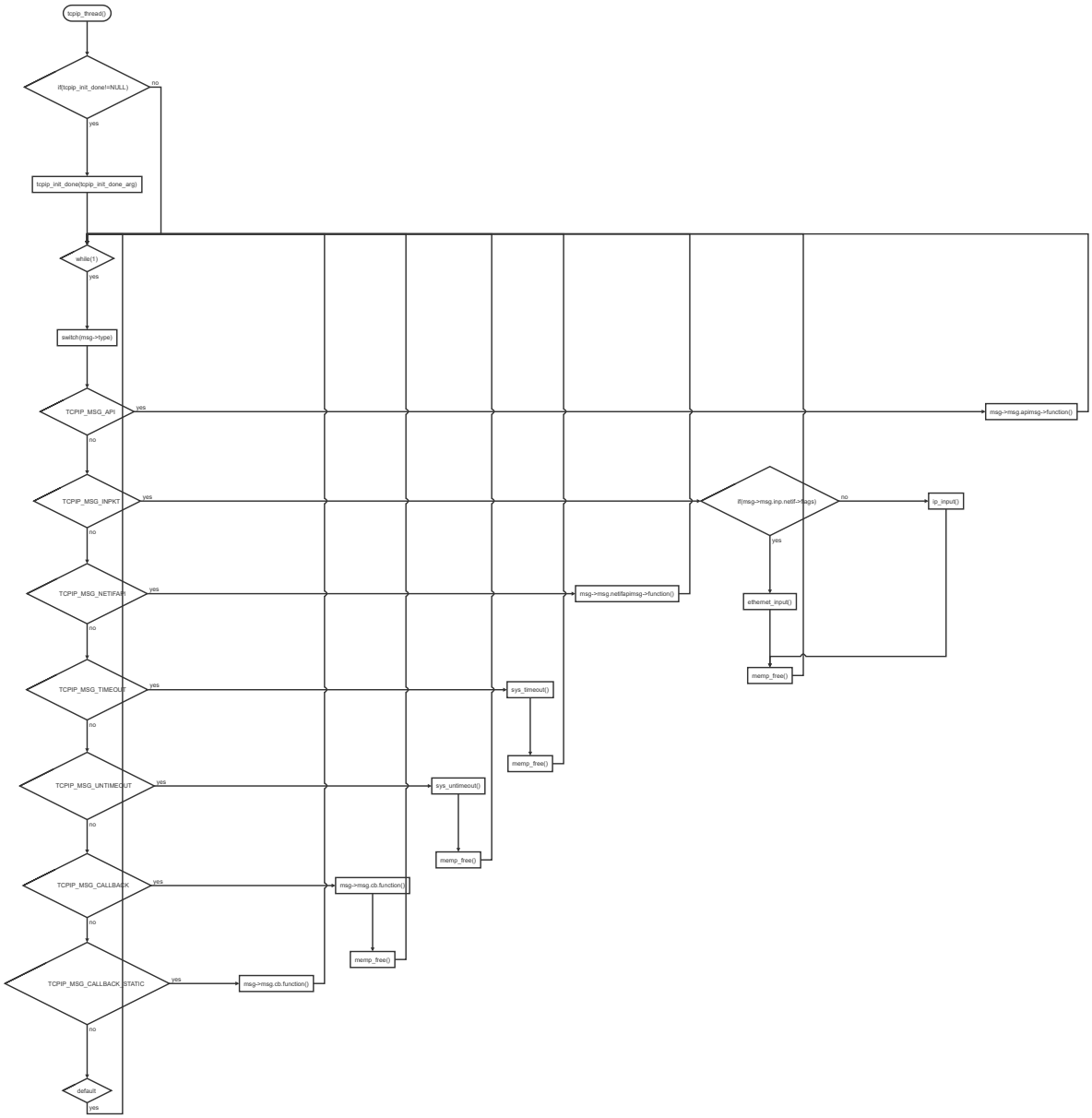
现在来看看这个函数做了哪些工作，首先为要发送数去的 ICMP 数据包申请一个 pbuf 缓冲区，这个缓冲区的长度为上图所述结构的长度与一个 IP 数据报头大小之和，之所以要多申请 IP 数据报头大小的空间是为了当该 ICMP 数据包被递交给 IP 层发送时，IP 层不需要再去申请一个数据报头来封装该 ICMP 数据包，而是直接在已经申请好的报头中填入 IP 头部数据，注意这里申请好的 pbuf 的 payload 指针是指向 ICMP 数据报头处的。接下来，函数填写 ICMP 数据包的相关字段，将类型段填充为 3，代码段为输入参数 t，同时将不可达的 IP 数据包的 IP 报头和数据前 8 个字节拷贝到 ICMP 数据包相应字段，最后计算校验和字段的值，然后调用 ip_output 函数将组装好的 ICMP 包发送出去。

TCPIP_Thread线程启动流程

源码文件tcpip.c tcpip.h



tcpip_thread主线程处理


tcpip_thread()
if(tcpip_init_done!=NULL)
no
yes
tcpip_init_done(tcpip_init_done_arg)
while(1)
yes
switch(msg->type)
TCPIP_MSG_API
yes
msg->msg.apimsg->function()
no
TCPIP_MSG_INPKT
yes
if(msg->msg.inp.netif->flags)
no
ip_input()
yes
ethernet_input()
no
TCPIP_MSG_NETIFAPI
yes
msg->msg.netifapimsg->function()
no
TCPIP_MSG_TIMEOUT
yes
sys_timeout()
memp_free()
no
TCPIP_MSG_UNTIMEOUT
yes
sys_untimeout()
memp_free()
no
TCPIP_MSG_CALLBACK
yes
msg->msg.cb.function()
memp_free()
no
TCPIP_MSG_CALLBACK_STATIC
yes
msg->msg.cb.function()
no
default
yes