# 一、 实验详细设计

*注意不要完全照搬实验指导书上的内容，请根据你自己的设计方案来填写*

## 1. Eth 协议详细设计

**【整体设计思路】**

Eth 协议位于数据链路层，其主要内容是控制主机在**发送**来自上层协议的数据包的时候，为该数据包**添加以太网包头**（包括长度/类型、目的 MAC 地址、源 MAC 地址等信息）；另一方面，还要控制主机在接收到来自驱动层的数据包时，**解析其中的以太网包头**，获得相关信息（包括长度/类型、源 MAC 地址等），再**去除以太网包头**。值得注意的是，由于网络字节序是大端字节序，而 X86 平台上是以小端字节序存储，因此**在上述发送、解析的过程中，还需要进行一定的大小端转换**。

以太网包头的帧格式如下所示：

| MAC 目标地址 | MAC 源地址 | 长度/类型 |
|---|---|---|
| 6 字节 | 6 字节 | 2 字节 |

故本次实验的主要任务就是在发送数据包时添加上述以太网包头，在接收数据包时解析并去除上述包头。

（1） 以太网数据帧发送处理流程（即 ethernet_out 函数）的设计与实现

流程设计如下：

- 【第一步】首先判断数据长度，如果不足 46 的话则在后面显式地填充 0。
- 【第二步】添加以太网包头。具体来说调用已经提供的 buf_add_header 函数在 buf 头部添加一段长度，并用指针 hdr 指向这段长度的起始地址，便于后续处理。
- 【第三步】根据传入的 mac 参数，使用 memcpy 函数为以太网头部填写目的 MAC 地址。
- 【第四步】使用 memcpy 函数为以太网头部填写源 MAC 地址。值得注意的是，由于上述目的 MAC 地址、源 MAC 地址的都是 uint8_t 数组，数组单个元素的大小都为一个字节，因此不需要进行大小端序的转换。
- 【第五步】为以太网头部填写协议类型。由于协议类型的大小为两个字节，因此需要先调用 swap16 函数进行大小端序的转换。
- 【第六步】将上述添加完以太网包的数据帧发送到驱动层。

代码实现如下：

```
/**
 * @brief 处理一个要发送的数据包
 *
 * @param buf 要处理的数据包
 * @param mac 目标 MAC 地址
 * @param protocol 上层协议
 */
void ethernet_out(buf_t *buf, const uint8_t *mac, net_protocol_t protocol)
{
    // TO-DO
    // Step1: 判断数据长度，如果不足 46 则显示填充 0
```

```
    if (buf->len < 46)
    {
        size_t len_padding = 46 - buf->len;
        buf_add_padding(buf, len_padding);
    }

    // Step2: 添加以太网包头
    buf_add_header(buf, sizeof(ether_hdr_t));
    ether_hdr_t *hdr = (ether_hdr_t *) buf->data;

    // Step3: 填写目的 MAC 地址
    memcpy(hdr->dst, mac, NET_MAC_LEN);

    // Step4: 填写源 MAC 地址
    memcpy(hdr->src, net_if_mac, NET_MAC_LEN);

    // Step5: 填写协议类型 protocol
    hdr->protocol16 = swap16(protocol);

    // Step6: 将添加了以太网包头的数据帧发送到驱动层
    driver_send(buf);
}
```

（2） 以太网数据帧接收处理流程（即 ethernet_in 函数）的设计与实现

流程设计如下：

- 【第一步】首先判断数据帧的长度，若数据帧长度小于以太网头部的长度，则认为数据包不完整，直接丢弃该数据包、不处理。
- 【第二步】解析以太网包头，获得协议类型、源 MAC 地址等信息。具体来说，获得数据包 buf 中的数据起始地址，将其赋值给以太网头部指针 hdr。然后利用指针 hdr 访问以太网包头的 protocol16、src 等成员变量，获取相应信息。与发送数据帧的过程同理，成员 protocol16 的大小为两个字节，需要转换大小端序，而 src 是 uint8_t 数组，数组单个元素的大小都为一个字节，因此不需要进行大小端序的转换。获取完包头信息后，将以太网包头去除。
- 【第三步】使用 net_in 函数向上层协议传递这个**已去除以太网包头的数据包**。

代码实现如下：

```
/**
 * @brief 处理一个收到的数据包
 *
 * @param buf 要处理的数据包
 */
void ethernet_in(buf_t *buf)
{
    // TO-DO
```

```
    // Step1: 判断数据长度，若数据小于以太网头部长度，则认为数据包不完整，丢弃不处理
    if (buf->len < sizeof(ether_hdr_t))
    {
        return;
    }

    // Step2: 解析以太网包头，获得协议类型、源 MAC 地址，再移除以太网包头
    ether_hdr_t *hdr = (ether_hdr_t *) buf->data;
    net_protocol_t protocol = swap16(hdr->protocol16);
    uint8_t *src = hdr->src;
    buf_remove_header(buf, sizeof(ether_hdr_t));

    // Step3: 向上层传递数据包
    net_in(buf, protocol, src);
}
```

## 2. ARP 协议详细设计

**【整体设计思路】**

在 **TCP/IP** 的网络构造和网络通信中无需事先知道 MAC 地址是什么，只要确定了 IP 地址就可以向这个目标地址发送 IP 数据报。但是**数据链路层**使用硬件地址（即 MAC 地址）进行报文传输，IP 地址不能被物理网络识别，因此必须建立 IP 地址和 MAC 地址的映射关系。**建立这一映射的过程就需要使用 ARP 地址解析协议。**

ARP 协议借助 **ARP 请求**与 **ARP 响应**两种类型的包来确定 MAC 地址。在每台使用 ARP 协议的主机中，都保留了一个专用的内存区存放最近的 **IP 地址-MAC 地址**映射对应关系。一旦该主机收到 ARP 响应，就会将获得的 **IP 地址-MAC 地址**映射关系存到缓存中。当发送报文时，主机首先去缓存中查找相应的项，如果能找到相应项（即获得对应的 MAC 地址），就将报文直接发送过去；如果找不到，再利用 ARP 进行解析，**发送 ARP 请求**，并将当前由于未获得 MAC 地址而无法发送的报文缓存到另一片内存区域中。此外，主机检测到发来的某个 ARP 请求报文所请求的 IP 地址正是本机地址，也会**发送 ARP 响应**。

故本次实验的主要任务包括实现 ARP 请求函数 arp_req、ARP 发送处理函数 arp_out、ARP 接受处理函数 arp_in、ARP 响应函数 arp_resp。

（1）ARP 请求函数 arp_req 的设计与实现

流程设计如下：

- 【第一步】使用 buf_init 函数对 tx_buf 进行初始化，传入的数据初始长度为 ARP 报文的长度（即 arp_pkt_t 的大小）。
- 【第二步】填写 ARP 报头，用预先定义好的初始 ARP 包 arp_init_pkt 进行填充。
- 【第三步】修改 ARP 的操作类型为 ARP_REQUEST，并修改目的 IP 地址为 target_ip。值得注意的是，pkt->opcode16 的大小为两个字节，因此需要进行大小端转换；而 pkt->target_ip 是单个元素大小为一个字节的 uint8_t 数组，因此不需要进行大小端转换（理由在前面已经介绍，不再赘述）。
- 【第四步】使用 ethernet_out 函数将 ARP 报文发送出去。

代码实现如下：

```
/**
 * @brief 发送一个 arp 请求
 *
 * @param target_ip 想要知道的目标的 ip 地址
 */
void arp_req(uint8_t *target_ip)
{
    // TO-DO
    // Step1: 对 txbuf 进行初始化
    buf_init(&txbuf, sizeof(arp_pkt_t));

    // Step2: 填写 ARP 报头
    arp_pkt_t *pkt = (arp_pkt_t *) txbuf.data;
    memcpy(pkt, &arp_init_pkt, sizeof(arp_pkt_t));

    // Step3: 修改 ARP 报头的操作类型为 ARP_REQUEST，并修改目的 IP 地址为
target_ip
    pkt->opcode16 = swap16(ARP_REQUEST);
    memcpy(pkt->target_ip, target_ip, NET_IP_LEN);

    // Step4: 调用 ethernet_out 函数将 ARP 报文发送出去
    ethernet_out(&txbuf, ether_broadcast_mac, NET_PROTOCOL_ARP);
}
```

（2）ARP 发送处理函数 arp_out 的设计与实现

流程设计如下：

- 【第一步】根据 IP 地址来查找 ARP 表。
- 【第二步】如果能找到该 IP 地址对应的 MAC 地址，直接调用 ethernet_out 函数将数据包发送给以太网层。
- 【第三步】如果没有找到该 IP 地址对应的 MAC 地址，先判断 arp_buf 是否已经有包了。如果有，说明正在等待该 IP 回应 ARP 请求，此时不能再发送 ARP 请求；如果没有，将来自 IP 层的数据包缓存到 arp_buf，发送一个请求与目标 IP 地址对应的 MAC 地址的 ARP 请求报文。

代码实现如下：

```
/**
 * @brief 处理一个要发送的数据包
 *
 * @param buf 要处理的数据包
 * @param ip 目标 ip 地址
 * @param protocol 上层协议
 */
void arp_out(buf_t *buf, uint8_t *ip)
```

```
{
    // TO-DO
    // Step1: 根据 IP 地址来查找 ARP 表
    uint8_t *mac = (uint8_t *) map_get(&arp_table, ip);

    if (mac != NULL)
    {
        // Step2: 如果能找到该 IP 地址对应的 MAC 地址，直接调用 ethernet_out
函数将数据包发送给以太网层
        ethernet_out(buf, mac, NET_PROTOCOL_IP);
    }
    else
    {
        // Step3: 如果没有找到对应的 MAC 地址，先判断 arp_buf 是否已经有包了
        buf_t *buf2 = (buf_t *) map_get(&arp_buf, ip);
        if (buf2 != NULL)
        {
            // 如果有，说明正在等待该 IP 回应 ARP 请求，此时不能再发送 ARP 请
求
            return;
        }
        else
        {
            // 如果没有，则将来自 IP 层的数据包缓存到 arp_buf，发送一个请求与
目标 IP 地址对应的 MAC 地址的 ARP 请求报文
            map_set(&arp_buf, ip, buf);
            arp_req(ip);
        }
    }
}
```

（3）ARP 接受处理函数 arp_in 的设计与实现

流程设计如下：

- 【第一步】首先判断数据长度，如果数据长度小于 ARP 头部长度，认为数据包不完整，丢弃不处理。
- 【第二步】做报头检查，检测 ARP 报头的硬件类型、上层协议类型、MAC 地址长度、IP 地址长度、操作类型等是否符合协议规定（与前面同理，同样需要主要大小端转换的问题，不再赘述）。
- 【第三步】更新 ARP 表项，使用 map_set 函数更新之前缓存的 **IP 地址-MAC 地址**映射。
- 【第四步】查看该接收报文的 IP 地址是否有对应的 arp_buf 缓存。如果有，说明上一次调用 arp_out 函数发送数据包时，由于没有找到对应的 MAC 地址先发送了 ARP 请求报文，但此时收到了该请求的响应报文，因此需要将缓存的数据包发送到以太网层，再将这个缓存的数据包删除掉；如果没有，还需要判断接收到的报文是否为 ARP 请求报文且所请求的主机正是本机，如果是的话，回应一个 ARP 响应报文。

代码实现如下：

```
/**
 * @brief 处理一个收到的数据包
 *
 * @param buf 要处理的数据包
 * @param src_mac 源 mac 地址
 */
void arp_in(buf_t *buf, uint8_t *src_mac)
{
    // TO-DO
    // Step1: 首先判断数据长度，如果数据长度小于 ARP 头部长度，认为数据包不完整，丢弃不处理
    if (buf->len < sizeof(arp_pkt_t))
    {
        return;
    }

    // Step2: 做报头检查，检测 ARP 报头的硬件类型、上层协议类型、MAC 硬件地址长度、IP 协议地址长度、操作类型等是否符合协议规定
    arp_pkt_t *arp_pkt = (arp_pkt_t *) buf->data;
    if (arp_pkt->hw_type16 != swap16(ARP_HW_ETHER)) return;
    if (arp_pkt->pro_type16 != swap16(NET_PROTOCOL_IP)) return;
    if (arp_pkt->hw_len != NET_MAC_LEN) return;
    if (arp_pkt->pro_len != NET_IP_LEN) return;
    if (arp_pkt->opcode16 != swap16(ARP_REQUEST) && arp_pkt->opcode16 != swap16(ARP_REPLY)) return;

    // Step3: 更新 ARP 表项
    map_set(&arp_table, arp_pkt->sender_ip, arp_pkt->sender_mac);

    // Step4: 查看该接收报文的 IP 地址是否有对应的 arp_buf 缓存
    buf_t *buf2 = (buf_t *) map_get(&arp_buf, arp_pkt->sender_ip);
    if (buf2 != NULL)
    {
        // 如果有，说明上一次调用 arp_out 函数发送数据包时，由于没有找到对应的 MAC 地址故先发送了 ARP request 报文
        // 此时收到了该 request 的应答报文，因此需要将缓存的数据包发送给以太网层，再将这个缓存的数据包删除掉
        ethernet_out(buf2, arp_pkt->sender_mac, NET_PROTOCOL_IP);
        map_delete(&arp_buf, arp_pkt->sender_ip);
    }
    else
    {
```

```
        // 如果没有，还需要判断接收到的报文是否为 ARP_REQUEST 请求报文，并且该
请求报文的 target_ip 是本机的 IP
        if (arp_pkt->opcode16 == swap16(ARP_REQUEST) &&
memcmp(arp_pkt->target_ip, net_if_ip, NET_IP_LEN) == 0)
        {
            // 如果是，回应一个 ARP 响应报文
            arp_resp(arp_pkt->sender_ip, arp_pkt->sender_mac);
        }
    }
}
```

（4）ARP 响应函数 arp_resp 的设计与实现

设计流程：

- 【第一步】使用 buf_init 函数对 tx_buf 进行初始化，传入的数据初始长度为 ARP 报文的长度（即 arp_pkt_t 的大小）。
- 【第二步】填写 ARP 报头，用预先定义好的初始 ARP 包 arp_init_pkt 进行填充（其中已经包含了本机的 MAC 地址）。
- 【第三步】修改 ARP 的操作类型为 ARP_REPLY，并修改目的 IP 地址为 target_ip、目的 MAC 地址为 target_mac。值得注意的是，pkt->opcode16 的大小为两个字节，因此需要进行大小端转换；而 pkt->target_ip、pkt->target_mac 是单个元素大小为一个字节的 uint8_t 数组，因此不需要进行大小端转换（理由在前面已经介绍，不再赘述）。
- 【第四步】使用 ethernet_out 函数将 ARP 报文发送出去。

代码实现如下：

```
/**
 * @brief 发送一个 arp 响应
 *
 * @param target_ip 目标 ip 地址
 * @param target_mac 目标 mac 地址
 */
void arp_resp(uint8_t *target_ip, uint8_t *target_mac)
{
    // TO-DO
    // Step1: 初始化 txbuf
    buf_init(&txbuf, sizeof(arp_pkt_t));

    // Step2: 填写 ARP 报头
    arp_pkt_t *pkt = (arp_pkt_t *) txbuf.data;
    memcpy(pkt, &arp_init_pkt, sizeof(arp_pkt_t));

    // Step3: 修改 ARP 报头的操作类型为 ARP_REPLY，并修改目的 IP 地址为
target_ip、目的 MAC 地址为 target_mac
    pkt->opcode16 = swap16(ARP_REPLY);
    memcpy(pkt->target_ip, target_ip, NET_IP_LEN);
```

```
    memcpy(pkt->target_mac, target_mac, NET_MAC_LEN);

    // Step4: 调用 ethernet_out 函数将 ARP 报文发送出去
    ethernet_out(&txbuf, target_mac, NET_PROTOCOL_ARP);
}
```

# 二、实验结果截图及分析

## 1. Eth 协议实验结果及分析

（1）实验结果

与 Eth 协议相关的测试均已通过，如下所示。

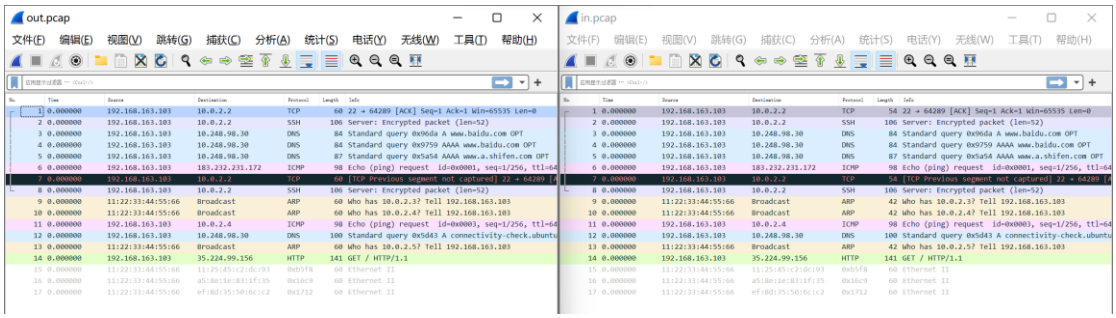
```
PS D:\Users\ywbisthebest\Documents\GitHub\2023_HITSZ_protocol-stack-labs\build> ctest -R eth_out
Test project D:/Users/ywbisthebest/Documents/GitHub/2023_HITSZ_protocol-stack-labs/build
    Start 2: eth_out
1/1 Test #2: eth_out ..........................   Passed    1.42 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   1.44 sec
PS D:\Users\ywbisthebest\Documents\GitHub\2023_HITSZ_protocol-stack-labs\build> ctest -R eth_in
Test project D:/Users/ywbisthebest/Documents/GitHub/2023_HITSZ_protocol-stack-labs/build
    Start 1: eth_in
1/1 Test #1: eth_in ...........................   Passed    0.14 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.15 sec
```

（2）结果分析

**对于 eth_out 测试：**

测试的方法为剥离 in.pcap 中的数据包的以太网头部，然后检查输出到 out.pcap 中的数据包是否被正确地恢复了以太网头部。

经过测试可以发现 out.pcap 中的输出与 demo_out.pcap 中的输出相一致，同时也可以发现 out.pcap 也恢复了 in.pcap 中的被去除了以太网头部的数据包。结果对比截图如下所示：




存在少部分数据帧的长度不一样，其中的原因在于，ethernet_out 函数会对**被去除了以太网包头的、长度不足 46 的数据包**显示填充 0。因此，这并不影响所实现的 ethernet_out 函数的正确性。

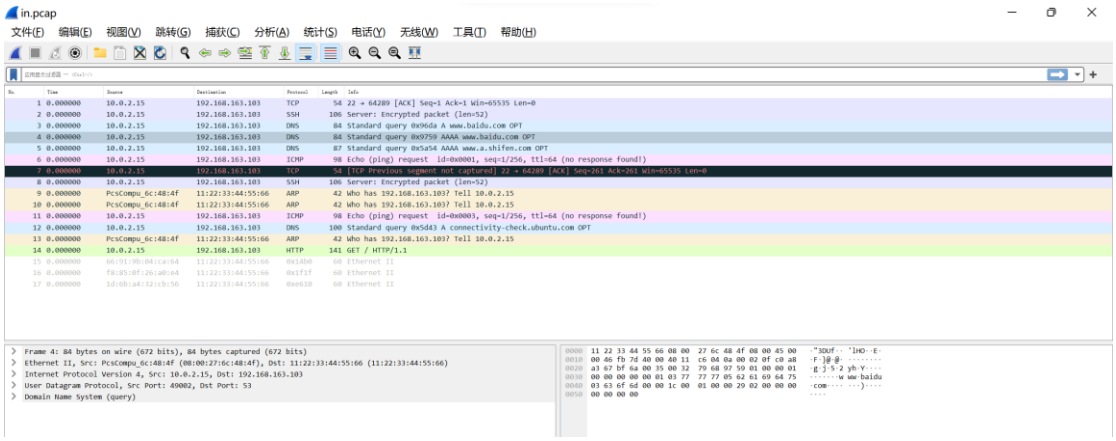
综上，eth_out 的功能实现正确。

**对于 eth_in 测试：**

测试的方法为查看输出到 log 文件中的 **IP 报文或 ARP 报文**是否被正确地解析、剥离了以太网头部。

经过测试可以发现 log 文件中的输出与 demo_log 文件中的输出相一致，同时 log 中的输出也确实对应于 in.pcap 中的数据包被去除了以太网头部的结果。

以第 4 个数据包为例，截图如下所示：




```
Round 04 ----------------------------
ip_in:
    mac:08:00:27:6c:48:4f
    buf: 45 00 00 46 fb 7d 40 00 40 11 c6 04 0a 00 02 0f c0 a8 a3 67 bf 6a 00 35 00 32 79 68 97 59 01 00 00 01 00 00 00 00 00 01 03 77
```

观察不难发现，in.pcap 中的第 4 个数据包确实被正确地**解析、去除**以太网头部，并被输出到 log 文件中，作为 Round 04 的 ip_in。Round 04 的 ip_in 的 mac、buf，也与 in.pcap 中的第 4 个数据包的内容相一致。

对于 in.pcap 文件以及 log 文件的其它数据包，可同理获得类似的分析，不再赘述。

综上，eth_in 的功能实现正确。

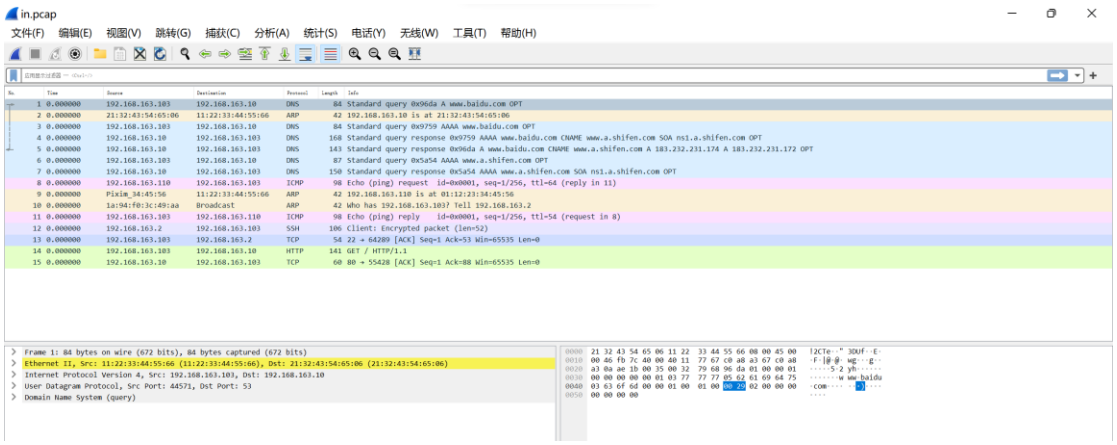
## 2. ARP 协议实验结果及分析

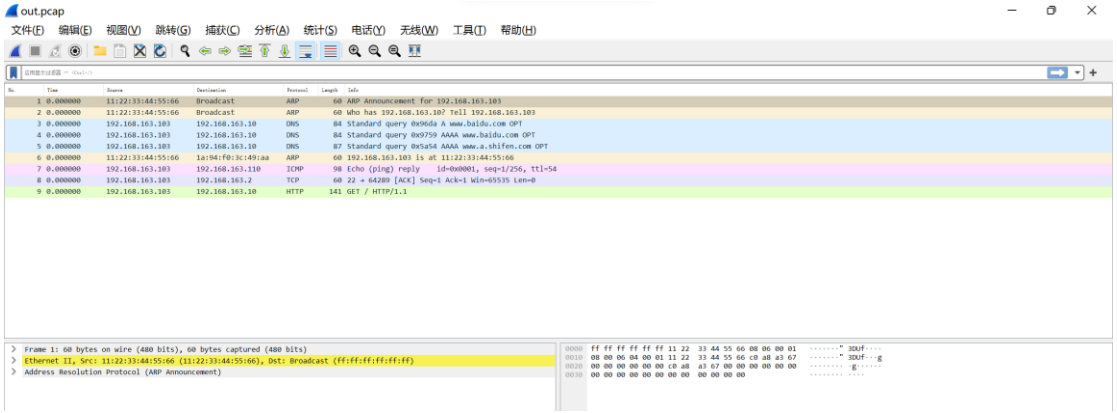
（1）实验结果

已通过 arp_test 的测试，如下图所示。

```
PS D:\Users\ywbisthebest\Documents\GitHub\2023_HITSZ_protocol-stack-labs\build> ctest -R arp_test
Test project D:/Users/ywbisthebest/Documents/GitHub/2023_HITSZ_protocol-stack-labs/build
    Start 3: arp_test
1/1 Test #3: arp_test .........................   Passed    0.14 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.15 sec
```

（2）结果分析




In[1]先试图发送域名解析请求，但是由于没有获得 MAC 地址，因此不能发送成功，只能暂时被缓存起来。Out[1]则是发送了无回报 ARP 包，表明本机要使用 IP 地址 192.168.163.103。Out[2]则是发送 ARP 请求，希望获得 IP 地址 192.168.163.10 的 MAC 地址，随后可以发现 In[2]获得了 ARP 回应，即本机已经成功获得 IP 地址 192.168.163.10 的 MAC 地址。之后可以发现 Out[3]成功发出了域名解析请求（不难发现 Out[3]与 In[1]是一致的，这也进一步印证：之前想发送域名解析请求，但是没有 MAC 地址无法发送，现在获得了 MAC 地址后可以发送了）。

之后的流程同理，不再赘述。由此可说明 arp_test 功能的正确性。