IP数据包捕获与分析

学号：2010349
姓名：孟笑朵
年级：2020级
专业：计算机科学与技术
撰写时间：2022年10月28日

注：本次实验包含了大量代码，用word编写不太友好，采用markdown的形式

实验内容说明

本实验的主要要求包括以下几点：
（1）了解NPcap的架构。
（2）学习NPcap的设备列表获取方法、网卡设备打开方法，以及数据包捕获方法。
（3）通过NPcap编程，实现本机的IP数据报捕获，显示捕获数据帧的源MAC地址和目的MAC地址，以及类型/长度字段的值。
（4）捕获的数据报不要求硬盘存储，但应以简单明了的方式在屏幕上显示。必显字段包括源MAC地址、目的MAC地址和类型/长度字段的值。
（5）撰写实验报告。

前期准备

在这一部分中参考了以下文档/博文的内容

> 1. Npcap Reference Guide
> 2. C/C++ Npcap包实现数据嗅探 - lyshark - 博客园 (cnblogs.com)
> 3. (29条消息) QT基于Npcap设计的网络抓包小程序__菜 -∞的博客-CSDN博客_npcap抓包

前期准备内容主要包括Npcap中相关的函数使用语法的学习。

设备列表获取方法

```C++
int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf);
//下面的函数也是获取设备列表的方法,只不过描述的信息比上面的函数要更加全面
```

```
int pcap_findalldevs_ex(char* source,  struct pcap_rmtauth *auth,
    pcap_if_t** alldevs,   char* errbuf );
```

参数解析:

1. `char* source`: 指定是指定是本地适配器或者远程适配器, 本地适配器PCAP_SRC_IF_STRING (rpcap://), 抓包文件PCAP_SRC_FILE_STRING(file://) , 远程适配器rpcap://host:port;
2. `struct pcap_rmtauth *auth`: 需要抓取其他主机的网卡信息时，该结构体中需要有访问目的主机时的身份验证信息，访问本机时可以传入NULL;
3. `pcap_if_t **alldevsp`: 存放获取的网卡设备的结构体，使用的数据结构是链表;
4. `char *errbuf`: 存放错误信息;
5. 返回值-1代表获取失败, 0代表获取成功;

`pcap_if_t` 结构体链表的描述:

```C++
struct pcap_if {
    struct pcap_if *next;
    char *name;          /* 网卡的name */
    char *description;   /* 描述,注意可能为NULL*/
    struct pcap_addr *addresses; /* 网卡的地址 */
    bpf_u_int32 flags;   /* PCAP_IF_ interface flags */
};
```

`struct pcap_rmtauth` 的描述:

```C++
struct pcap_rmtauth
{
    int type;            // 简要身份验证所需的类型
    char *username;      // 用户名
    char *password;      // 密码
};
```

释放网卡信息:

```C++
//释放网卡信息的结构体链表，防止内存泄漏
void pcap_freealldevs(pcap_if_t *alldevs);
```

网卡设备打开方法

```C++
pcap_t* pcap_open(const char* source,int snaplen,int flags,int
read_timeout,
        struct pcap_rmtauth* auth,char* errbuf);
```

参数解析:

1. `const char* source`:需要打开源的名称, 也就是之前获取的网卡设备的名称;
2. `int snaplen`: 必须保留的包的长度。对于每个数据包，只接收前 snaplen 长度的数据, 一般设定为65536, 保证整个数据包可以被链路层捕获;
3. `int flags`:捕获数据包的方式, 有一些flag定义;
4. `int read_timeout`: 以毫秒为单位。read timeout被用来设置在遇到一个数据包的时候读操作不必立即返回，而是等待一段时间，让更多的数据包到来后从OS内核一次读多个数据包;
5. `struct pcap_rmtauth*`: 保存当一个用户登录到某个远程机器上时的必要信息。假如不是远程抓包，该指针被设置为NULL;
6. `char* errbuf`:一个指向用户申请的缓冲区的指针，存放当该函数出错时的错误信息;
7. 返回值是一个`pcap_t*`指针，它可以作为下一步调用（例如`pcap_compile()`等）的参数，并且指定了一个已经打开的`npcap会话`。在遇到问题的情况下，它返回NULL并且`errbuf`变量保存了错误信息。

捕获数据包的方式定义:

```C++
/*
 * Specifies whether promiscuous mode is to be used.
 * 混杂模式获取数据包 UDP
 */
#define PCAP_OPENFLAG_PROMISCUOUS       0x00000001
/*
 * 它定义了数据传输（假如是远程抓包）是否用UDP协议来处理
 */
#define PCAP_OPENFLAG_DATATX_UDP        0x00000002
/*
*它定义了远程探测器是否捕获它自己产生的数据包
*/
#define PCAP_OPENFLAG_NOCAPTURE_RPCAP       0x00000004
```

数据包捕获方法

```
int pcap_next_ex(pcap_t* p,struct pcap_pkthdr** pkt_header,const u_char**
pkt_data)
```

参数解析:

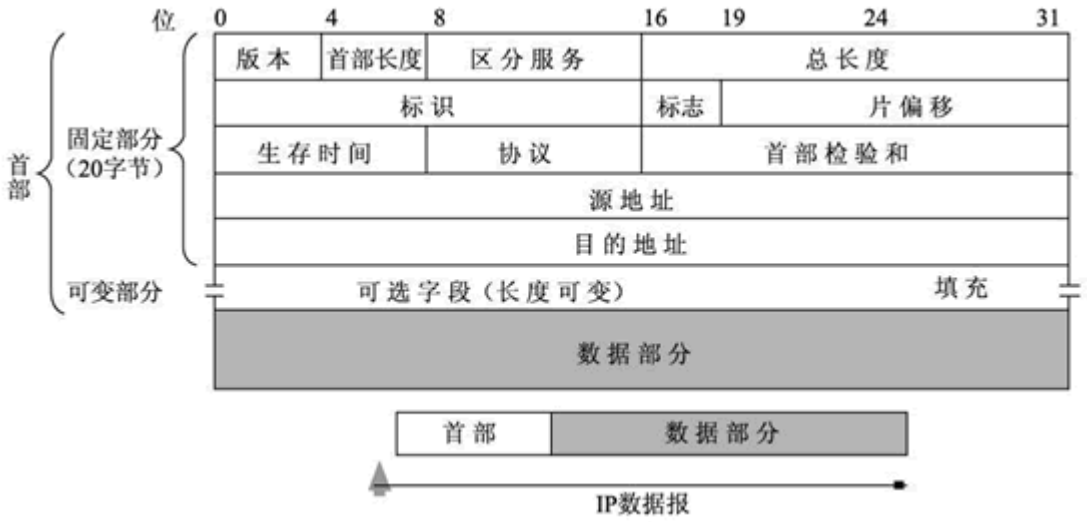
1. `pcap_t* p`: 已打开的捕捉实例的描述符，之前通过pcap_open函数的返回值
2. `struct pcap_pkthdr** pkt_header`: 报文头
3. `const u_char** pkt_data`: 报文内容
4. 返回值1: 成功;0: 获取报文超时;-1: 发生错误;-2: 获取到离线记录文件的最后一个报文

以太帧的结构

对其结构体的封装如下:

```
/* 以太网头数据帧结构体封装 */
struct ether_header {
    u_char ether_dhost[6];    // 目标地址
    u_char ether_shost[6];    // 源地址
    u_short ether_type;       // 以太网类型
};
```

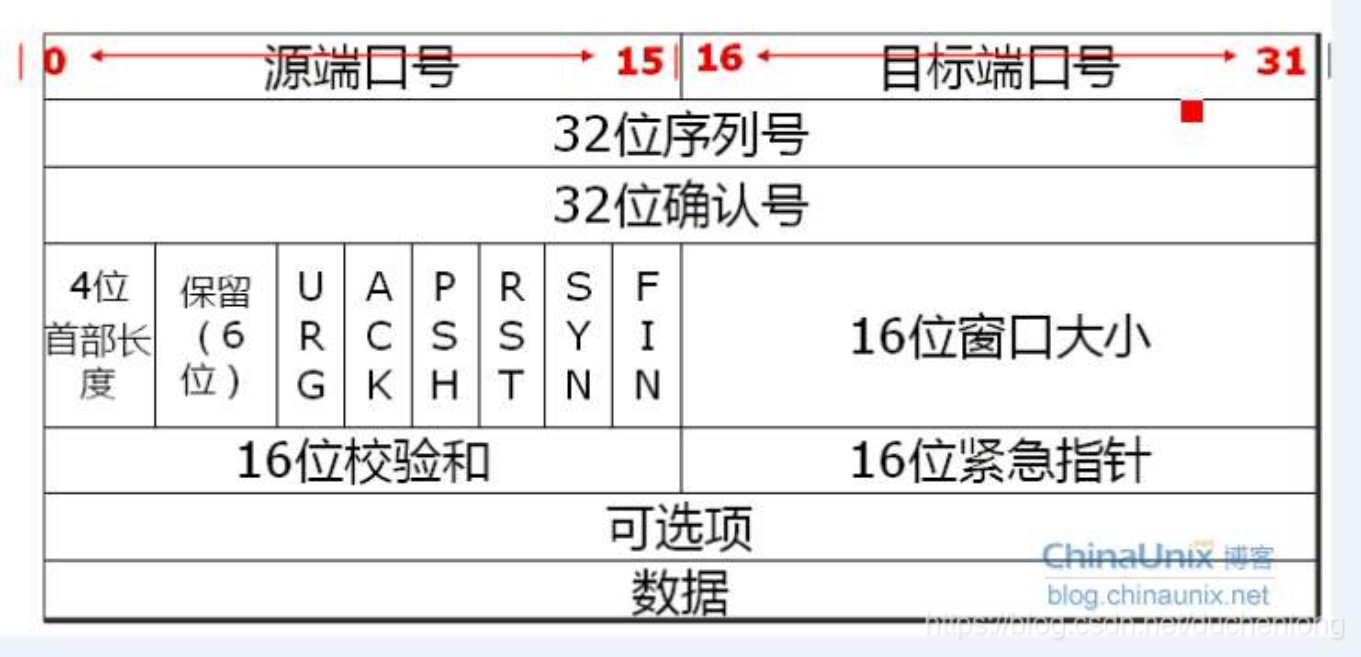
IP数据报格式



封装结构体如下

```C++
/* 网络层IP协议封装结构体封装 */
struct ip_hander {
    u_char      _version_handerLen;     // 版本 4  首部长度 4
    u_char      _diffserv;              // 服务类型
    u_short     _totalLen;              // 总长度
    u_short     _identification;        // 标识
    u_short     _flag_offset;           // 标志 3 + 片偏移 13
    u_char      _timeLive;              // 生存时间
    u_char      _protocol;              // 协议
    u_short     _checkSum;              // 首部校验和
    long        _src;                   // 源地址
    long        _desc;                  // 目的地址
};
```

传输层TCP协议结构



封装结构体如下:

```C++
/* 传输层TCP协议 */
struct TCP_hander {
    u_short      _sport;          // 源端口16bits
    u_short      _dport;          // 目的端口16bits
    u_int        _seqNum;         // 序列号32bits
    u_int        _ackNum;         // 确认号32bits
    u_short      _off_res_flag;   // 数据偏移 4  保留位 6  标志位 6
    u_short      _winSize;        // 窗口大小16bits
    u_short      _checkSum;       // 校验和16bits
    u_short      _urgentPoint;    // 紧急指针16bits
};
```

传输层UDP协议结构

封装结构体如下：

```C++
/* 传输层UDP协议 */
struct UDP_handler {
    u_short      _sport;     // 源端口
    u_short      _dport;     // 目的端口
    u_short      _len;       // 数据长度
    u_short      _checksum;  // 校验和
};
```

实验过程

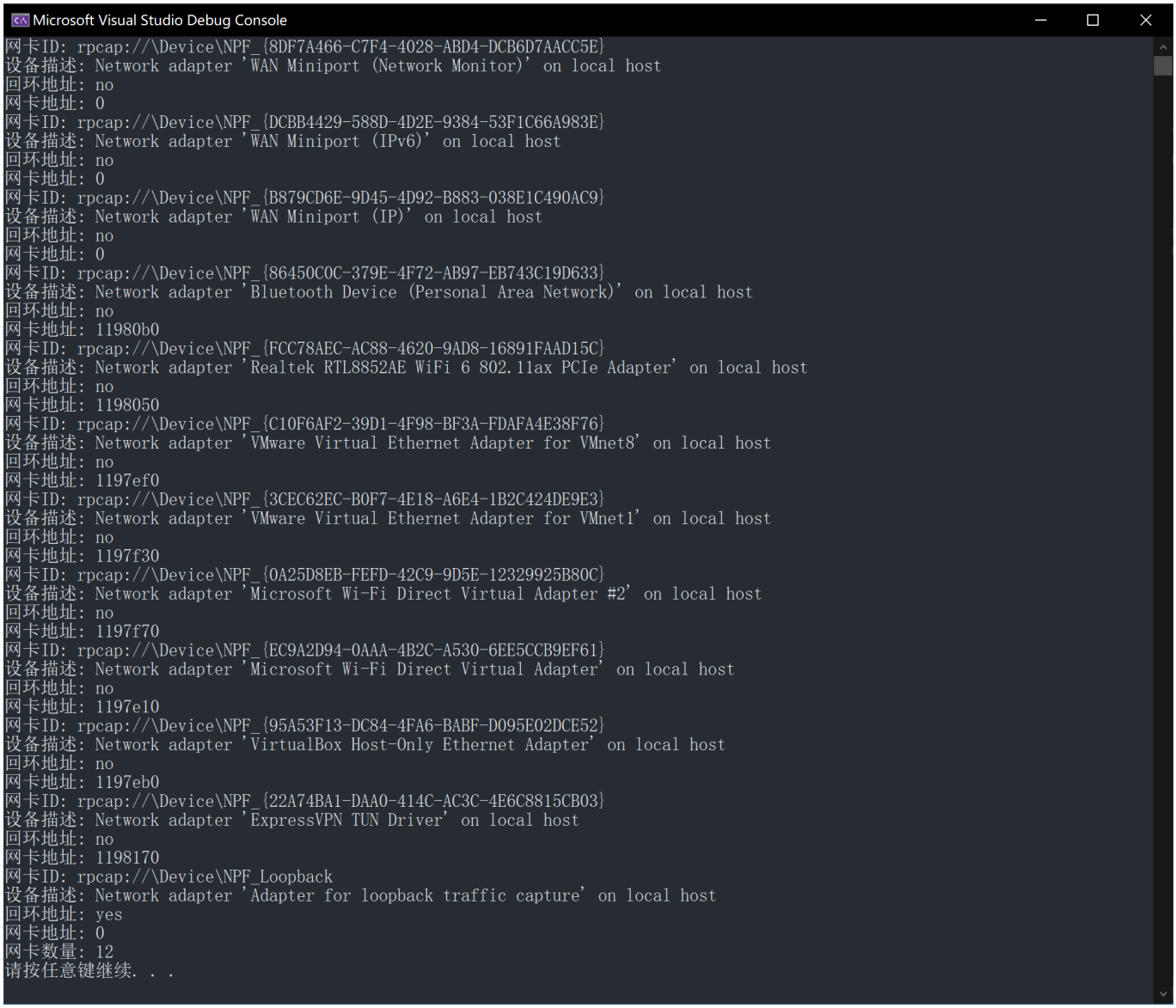
实验部分分为四大部分，第一部分实现简单的获取设备列表打印设备，第二部分实现打开设备，对数据包实现嗅探，第三部分实现对数据包的链路层解析，第四部分输出捕获数据报的所有信息。

获取设备列表打印设备信息

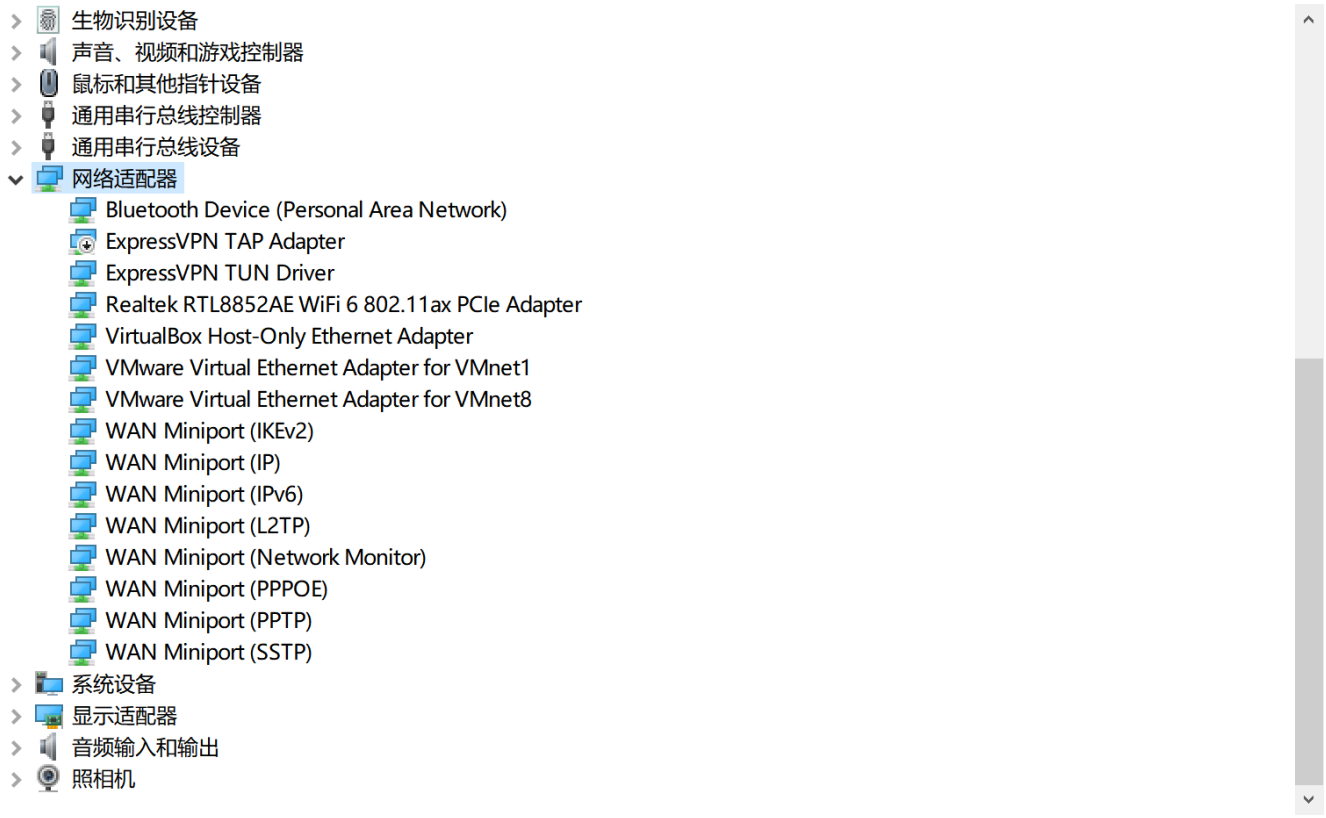
在前期准备中已经详细说明了`npcap`获取设备信息的方法，这里我们采用获取信息更加全面的函数`pcap_findalldevs_ex`来进行实现：

```C++
int enumAdapters()
{
    pcap_if_t* allAdapters;    // 所有网卡设备保存
    pcap_if_t* ptr;            // 用于遍历的指针
    int index = 0;
    char errbuf[PCAP_ERRBUF_SIZE];
```

```
    /* 获取本地机器设备列表 */
    if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &allAdapters,
errbuf) != -1)
    {
        /* 打印网卡信息列表 */
        for (ptr = allAdapters; ptr != NULL; ptr = ptr->next)
        {
            /* 获取设备数目 */
            ++index;
            /* Name */
            printf("网卡ID: %s\n", ptr->name);
            /* Description */
            if (ptr->description)
                printf("设备描述: %s\n", ptr->description);
            /* Loopback Address*/
            printf("回环地址: %s\n", (ptr->flags & PCAP_IF_LOOPBACK) ?
"yes" : "no");
            /* IP addresses */
            printf("网卡地址: %x \n", ptr->addresses);
        }
    }

    /* 不再需要设备列表了，释放它 */
    pcap_freealldevs(allAdapters);
    return index;
}
```



打开设备网卡信息，可以对照比较（有些网卡捕获不到）：



打开设备，对数据包实现嗅探

在这里我们可以选取任意上述捕获到的网卡，这里选取的是`Wifi`的网卡进行捕获数据包，具体的捕获代码如下：

1. 先找到指定网卡；
2. 打开网卡；
3. 开始侦听；

```C++
void MonitorAdapter(int nChoose)
{
    pcap_if_t* adapters;
    char errbuf[PCAP_ERRBUF_SIZE];

    if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &adapters, errbuf)
!= -1)
    {
        // 找到指定的网卡
        for (int x = 0; x < nChoose - 1; ++x) {
            // 找不到的情况
            if (adapters == NULL) {
                printf("error!");
                return;
```

```
            }
            adapters = adapters->next;
        }

        char errorBuf[PCAP_ERRBUF_SIZE];

        pcap_t* handle = pcap_open(adapters->name, //打卡的设备名称
            65536, //必须保留包的长度
            PCAP_OPENFLAG_PROMISCUOUS, // 网卡为混乱模式
            1000, // 超时1s
            NULL, // 远程连接确认
            errbuf // 错误信息
        );

        printf("开始侦听: % \n", adapters->description);
        pcap_pkthdr* Packet_Header;    // 数据包头
        const u_char* Packet_Data;    // 数据本身
        //持续接收数据包
        while (pcap_next_ex(handle, &Packet_Header, &Packet_Data) >= 0){
            printf("侦听长度: %d \n", Packet_Header->len);
        }
    }
}
```

实现数据包链路层解析

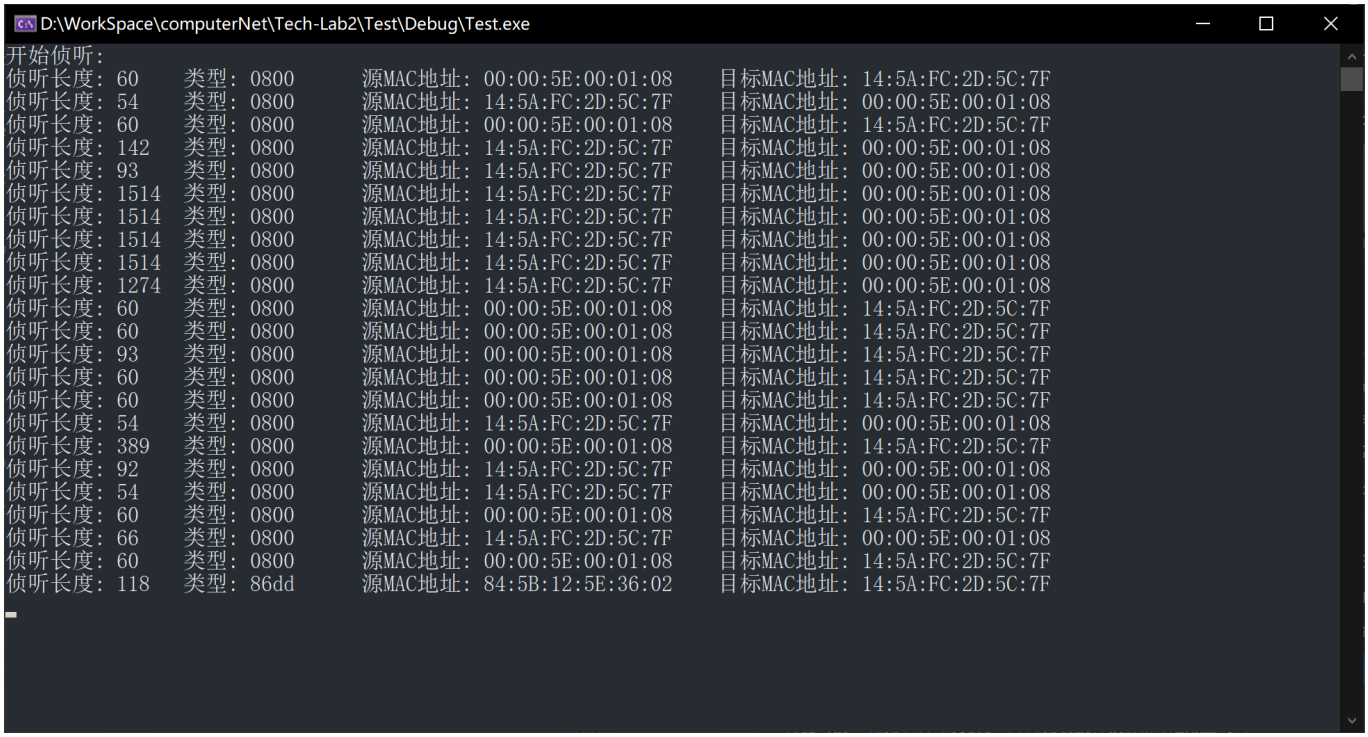
具体的链路层的解析，需要使用上面定义的`ether_header`以太帧的数据结构，注意这里的`ntohs`进行数据转换，实际上就是为了对齐结构体内部的数据排列方式。如下所示：

C++

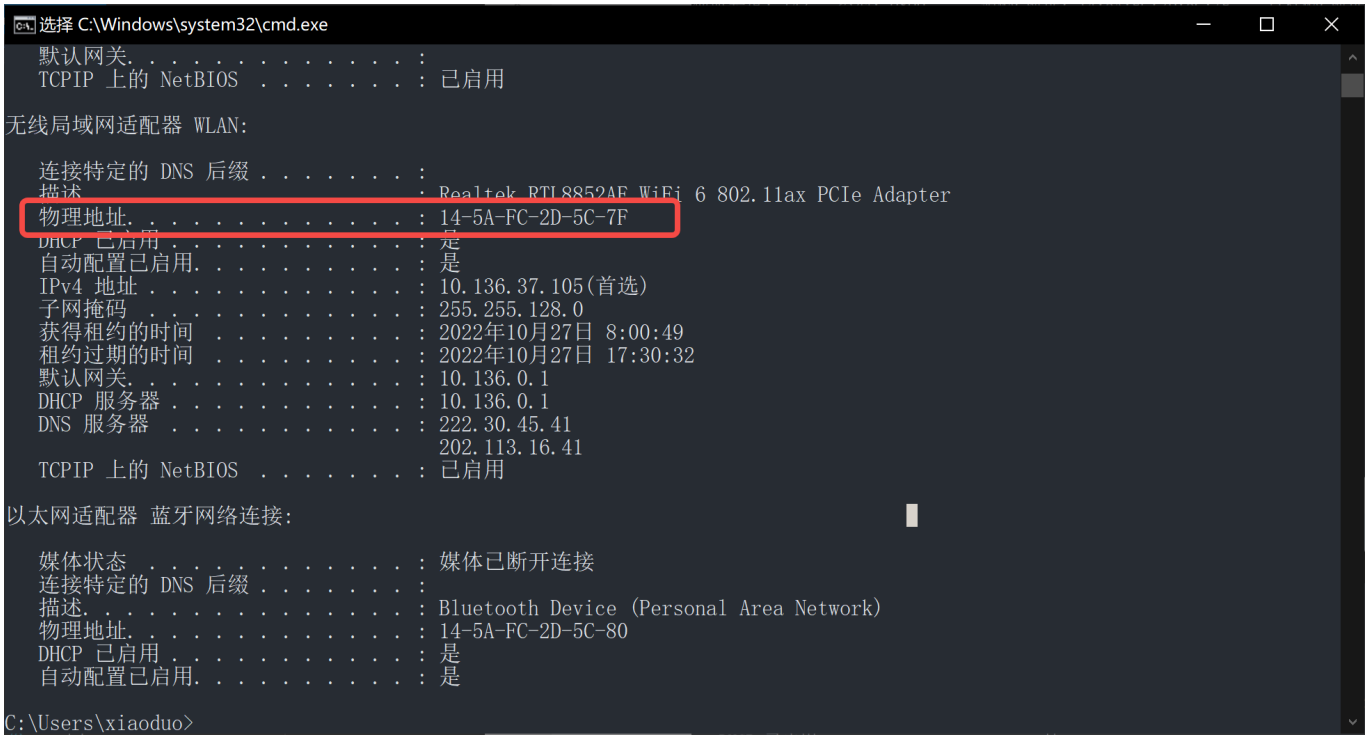
```
// 输出MAC地址等
void PrintEtherHeader(const u_char* packetData)
{
    ether_header* eth_protocol;
    eth_protocol = (ether_header*)packetData;
    u_char* ether_src = eth_protocol->ether_shost;          // 以太网原始
MAC地址
    u_char* ether_dst = eth_protocol->ether_dhost;          // 以太网目标
MAC地址
    u_short ether_type = ntohs(eth_protocol->ether_type);  // 以太网类型
```

```
    printf("类型: %04x \t", ether_type);
    printf("源MAC地址: %02X:%02X:%02X:%02X:%02X:%02X \t",
        ether_src[0], ether_src[1], ether_src[2], ether_src[3],
ether_src[4], ether_src[5]);
    printf("目标MAC地址: %02X:%02X:%02X:%02X:%02X:%02X \n",
        ether_dst[0], ether_dst[1], ether_dst[2], ether_dst[3],
ether_dst[4], ether_dst[5]);
}
```

输出如下图所示：



对照Wifi的MAC地址，对照成功：



实现数据包全信息输出

结合实验准备中的结构体变量,我们可以进一步解析对应的IP地址,目的端口和源端口的TCP和UDP信息,关键输出函数如下所示:

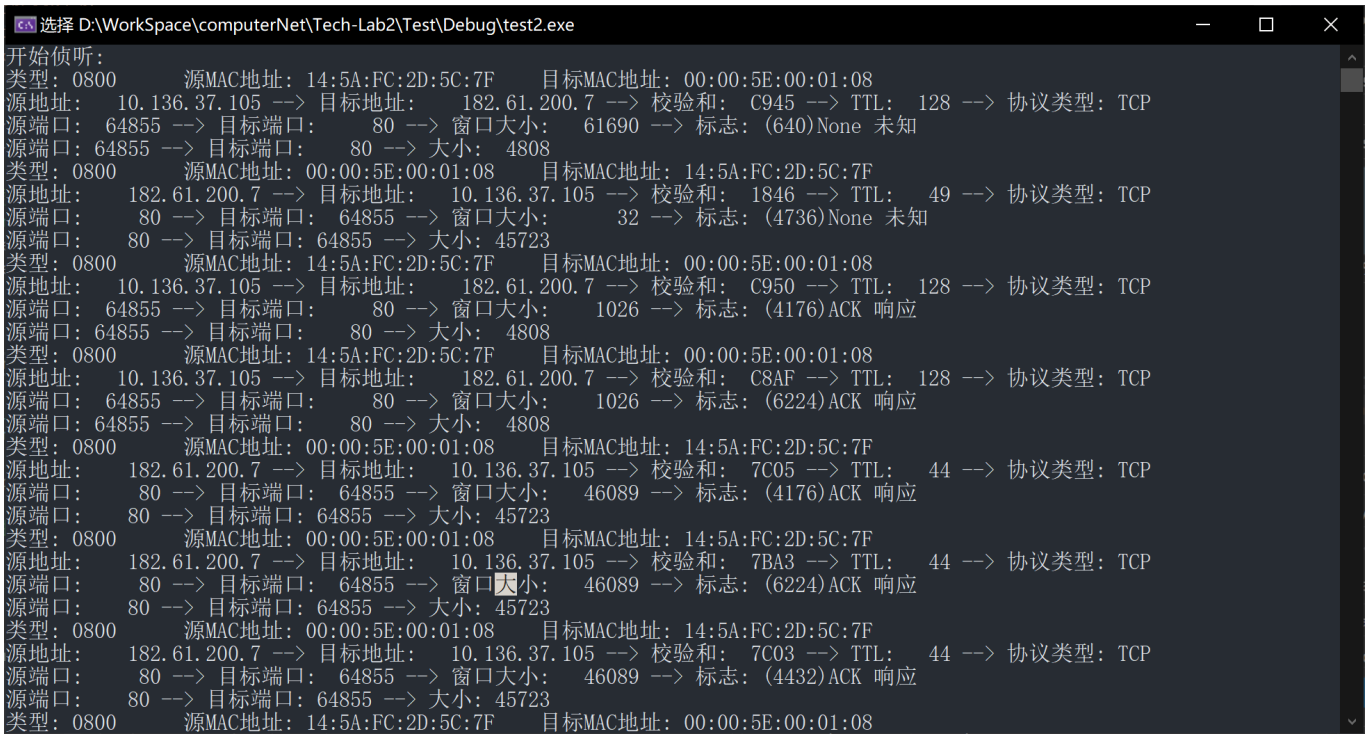
C++

```
// IP地址的捕获
void PrintIPHeader(const u_char* packetData)
{
    ip_hander* ip_protocol;

    // 为了跳过数据链路层
    ip_protocol = (ip_hander*)(packetData + 14);
    SOCKADDR_IN Src_Addr, Dst_Addr = { 0 };

    u_short check_sum = ntohs(ip_protocol->_checkSum);
    int ttl = ip_protocol->_timeLive;
    int proto = ip_protocol->_protocol;

    Src_Addr.sin_addr.s_addr = ip_protocol->_src;
    Dst_Addr.sin_addr.s_addr = ip_protocol->_desc;

    printf("源地址: %15s --> ", inet_ntoa(Src_Addr.sin_addr));
    printf("目标地址: %15s --> ", inet_ntoa(Dst_Addr.sin_addr));

    printf("校验和: %5X --> TTL: %4d --> 协议类型: ", check_sum, ttl);
```

```
    switch (ip_protocol->_protocol)
    {
        case 1: printf("ICMP \n"); break;
        case 2: printf("IGMP \n"); break;
        case 6: printf("TCP \n");  break;
        case 17: printf("UDP \n"); break;
        case 89: printf("OSPF \n"); break;
        default: printf("None \n"); break;
    }
}

// 打印TCP信息
void PrintTCPHeader(const unsigned char* packetData)
{
    TCP_hander* tcp_protocol;
    // +14 跳过数据链路层 +20 跳过IP层
    tcp_protocol = (TCP_hander*)(packetData + 14 + 20);

    u_short sport = ntohs(tcp_protocol->_sport);
    u_short dport = ntohs(tcp_protocol->_dport);
    int window = tcp_protocol->_winSize;
    int flags = tcp_protocol->_off_res_flag;

    printf("源端口: %6d --> 目标端口: %6d --> 窗口大小: %7d --> 标志: (%d)",
        sport, dport, window, flags);

    if (flags & 0x08) printf("PSH 数据传输\n");
    else if (flags & 0x10) printf("ACK 响应\n");
    else if (flags & 0x02) printf("SYN 建立连接\n");
    else if (flags & 0x20) printf("URG \n");
    else if (flags & 0x01) printf("FIN 关闭连接\n");
    else if (flags & 0x04) printf("RST 连接重置\n");
    else printf("None 未知\n");
}

// 打印UDP信息
void PrintUDPHeader(const unsigned char* packetData)
{
    UDP_handler* udp_protocol;
    // +14 跳过数据链路层 +20 跳过IP层
```

```
    udp_protocol = (UDP_handler*)(packetData + 14 + 20);

    u_short sport = ntohs(udp_protocol->_sport);
    u_short dport = ntohs(udp_protocol->_dport);
    u_short datalen = ntohs(udp_protocol->_len);

    printf("源端口: %5d --> 目标端口: %5d --> 大小: %5d \n", sport, dport,
datalen);
}
```

输出如下图所示:



还有一些其他实验可以探索，比如发送ARP数据包等，在官方文档中也给出了对应的例子，这里就不进行赘述了。

特殊现象分析

在进行实验过程中发现了一些特殊的函数`ntohs`，`inet_ntoa`等,实验发现去掉这些函数后程序会报错,这里列举了一些函数的含义进行解析。

```C++
//以太网类型捕获
u_short ether_type = ntohs(eth_protocol->ether_type);  // 以太网类型
//ip地址捕获
Src_Addr.sin_addr.s_addr = ip_protocol->_src;
```

```
Dst_Addr.sin_addr.s_addr = ip_protocol->_desc;

printf("源地址: %15s --> ", inet_ntoa(Src_Addr.sin_addr));
printf("目标地址: %15s --> ", inet_ntoa(Dst_Addr.sin_addr));
//TCP UDP端口号获取
u_short sport = ntohs(tcp_protocol->_sport);
u_short dport = ntohs(tcp_protocol->_dport);
```

`ntohs`是一个函数名，作用是将一个16位数由网络字节顺序转换为主机字节顺序

```
u_short PASCAL FAR ntohs( u_short netshort);
```
C++

参数解析:

1. `netshort`：一个以网络字节顺序表达的16位数;
2. 返回一个以主机字节顺序表达的数。

与它对应的是`htons`函数，功能是将一个无符号短整型数值转换为网络字节序，即大端模式(big-endian)

```
u_short PASCAL FAR htons( u_short hostshort);
```
C++

参数解析:

1. `hostshort`: 16位无符号整数
2. 返回TCP / IP网络字节顺序

还有`ntohl`，`htonl`函数发挥着类似的功能，总的来说:
htonl()--"Host to Network Long"
ntohl()--"Network to Host Long"
htons()--"Host to Network Short"
ntohs()--"Network to Host Short"

在本程序中的含义是网络中抓取的数据包，如果直接按结构体对齐的方式赋值给结构体指针，则结构体中长度小于字节的整形数据必须使用ntohs转换，因为抓取的包字段是按网络字节序排列的，直接通过指针的指向，小于一个字节的数据赋值将与原值不一直，因为内存排列方式不一样