实验19：以太网传输程序编写实验@

一、实验目的

1.通过实验了解以太网通讯原理和驱动程序开发方法。

2.通过实验了解 TCP 和 UDP 协议的功能和作用。

3.通过实验了解基于 TCP/UDP 的 socket 编程。

二、实验内容

1. 学习 KSZ8001L 网卡驱动程序。

2. 测试网卡功能，编写基于 TFP/UDP 协议网络聊天程序，并且能够接受键盘输入和彼 此之间相互发数据。

三、实验设备

1.硬件：PC 机，基于 ARM9 系统教学实验系统实验箱 1 台；网线；串口线，电压表。

2.软件：PC 机操作系统；Putty；服务器 Linux 操作系统；arm-v5t_le-gcc 交叉编译环境。

3.环境：ubuntu12.04.4；文件系统版本为 filesys_test；烧写的内核版本为 uImage_mceb。源码及参考文档见附件 I2C 驱动实验文件夹。

四、预备知识

1. C 语言的基础知识。

2. 软件调试的基础知识和方法。

3. Linux 基本操作。

4. Linux 应用程序的编写。

五、实验说明

1.1 以太网的工作原理

以太网采用带冲突检测的载波帧听多路访问（CSMA/CD）机制。以太网中节点都可以看到在网络中发送的所有信息，因此，我们说以太网是一种广播网络。

以太网的工作过程如下：

当以太网中的一台主机要传输数据时，它将按如下步骤进行：

1、帧听信道上收否有信号在传输。如果有的话，表明信道处于忙状态，就继续帧听，直到信道空闲为止。

2、若没有帧听到任何信号，就传输数据

3、传输的时候继续帧听，如发现冲突则执行退避算法，随机等待一段时间后，重新执行步骤 1（当冲突发生时，涉及冲突的计算机会发送会返回到帧听信道状态。

注意：每台计算机一次只允许发送一个包，一个拥塞序列，以警告所有的节点）

4、若未发现冲突则发送成功，计算机所有计算机在试图再一次发送数据之前，必须在最近一次发送后等待 9.6 微秒（以 10Mbps 运行）。

1.2 以太网帧传输原理

使用 KSZ8001L 网卡做为以太网的物理层接口。它的基本工作原理是：在收到主机发送的数据后（如图 1 帧格式所示，从目的地址域到数据域），先侦听并判断网络线路。若网络线路繁忙，等到网络线路空闲为止，否则，立即发送数据帧。在其过程中，先填加以太网帧头(如图 1 帧格式所示，包括前导码和帧开始标志)，接着生成 CRC 校验码，最后将以数据帧形式将其发送到以太网上去。

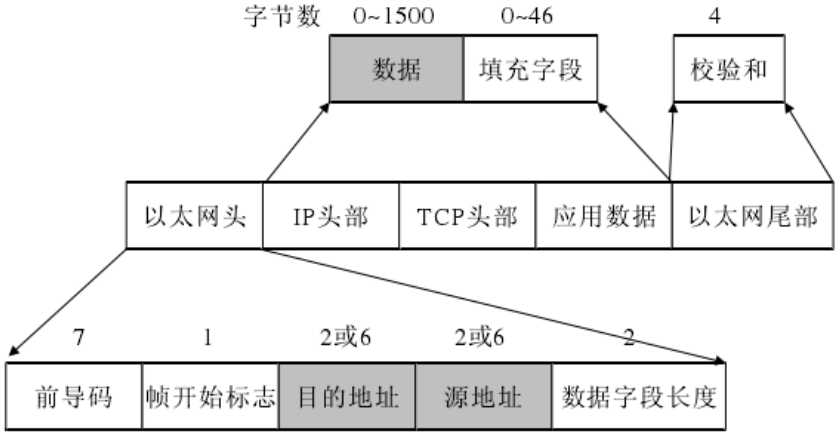


图 1　802.3 帧格式

以太网的数据帧规定，以太网包头为 14byte ，IP 包头为 20byte ，TCP 包头为 20byte ，有些以太网的最大帧长为 1514 字节。

在网卡接收数据过程中，将先把从以太网收到的数据帧通过解码、去帧头和地址检验等相关步骤后缓存在片内；再经过 CRC 校验后，通知网卡 KSZ8001L 接收已经到了数据帧；最后，用某种传输模式传送 ARM 的存储器中。大多数嵌入式系统内嵌一个以太网控制器，用来支持媒体独立接口（MII）和带缓冲的 DMA 接口（BDI）。可在半双工或全双工模式下提供 10M/100Mbps 的以太网接入。在半双工模式下，控制器支持 CSMA/CD 协议，在全双工模式下控制器能够支持 IEEE 802.3 MAC 的控制层协议。

以太网内部控制器内部结构图如下所示

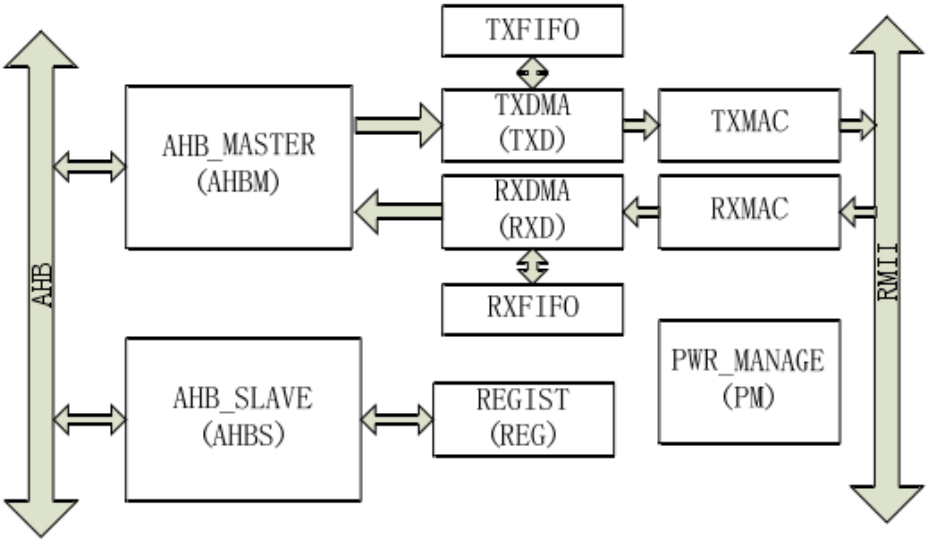


图 2　MAC 控制器的结构框图

由图可以看出模块的组成以及模块中各组成部分的连接方式。MAC 控制器各组成部分的特征或功能有：

1.AHB_MASTER 同时支持大端和小端模式，可以根据需要自行设定。一旦决定其模式，AHB_SLAVE 也将按这种模式工作。

2.TXDMA 主要有读取发送状态描述符或将发送状态写入描述符、将要发送的数据包从发送缓存区中传送到 TXFIFO 中、控制 TXFIFO 的读写状态等三个功能。

3.RXDMA 也有三个作用，分别是：读取接收状态描述符或将接收状态写入描述符、将要发送的数据包从 TXFIFO 中传送到接收缓存区中、控制 RXFIFO 的读写状态。

4. TXMAC 的作用是将数据包从 TXFIFO 中传送到网卡，RXMAC 的作用是将数据包从网卡接收到 RXFIFO 中。不论是接收到的还是要发送出去的数据都包括前缀、校验、发送状态等。

1.2.1 以太网发送和接收的流程

当 TXMAC 去发送一个数据包时，它首先会检测网卡的状态，挂起发送装置直到网卡空闲。然后，TXMAC 给数据包加上前缀和校验信息，将数据包发送到网卡。如果在发送过程中 TXMAC 探测到冲突，就向网卡发送阻塞信号，然后检测冲突源是否是外来的，如果不是，在等待休止时间后，继续发送该数据包。发送的流程如图 3 所示。

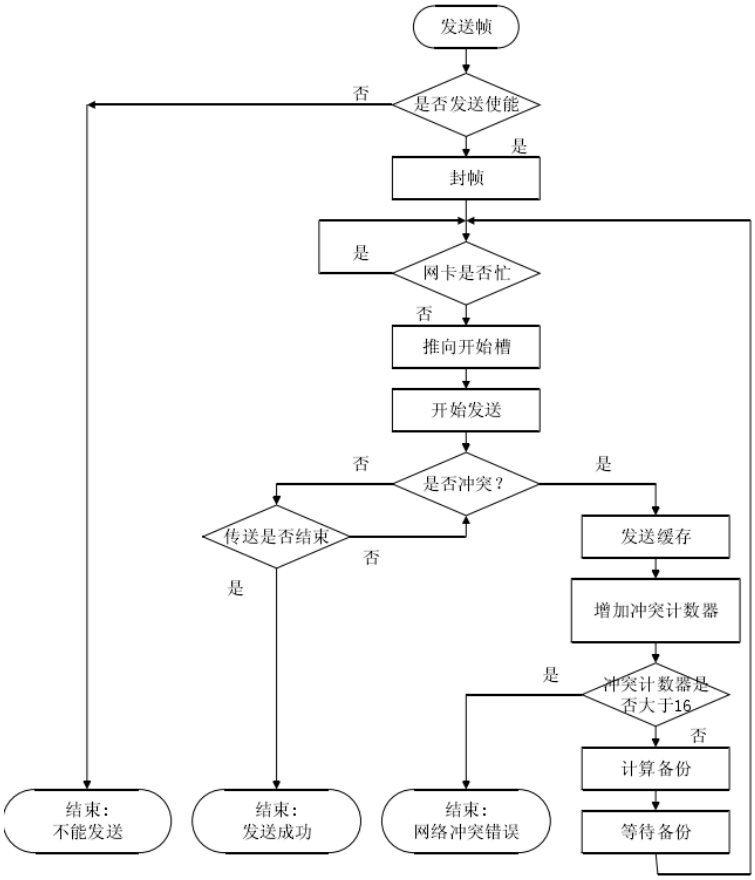


图 3　MAC 发送流程

当网络上有数据包到达，RXDMA 会将数据包从 RXMAC 中拿出来，并传送到 RX FIFO 中。当校验信息和数据包的地址都是正确时，RXDMA 就会将收到的数据包保存在 RXFIFIO 中，否则，忽略该数据包。图 4 描述了 MAC 接收数据包的基本流程。

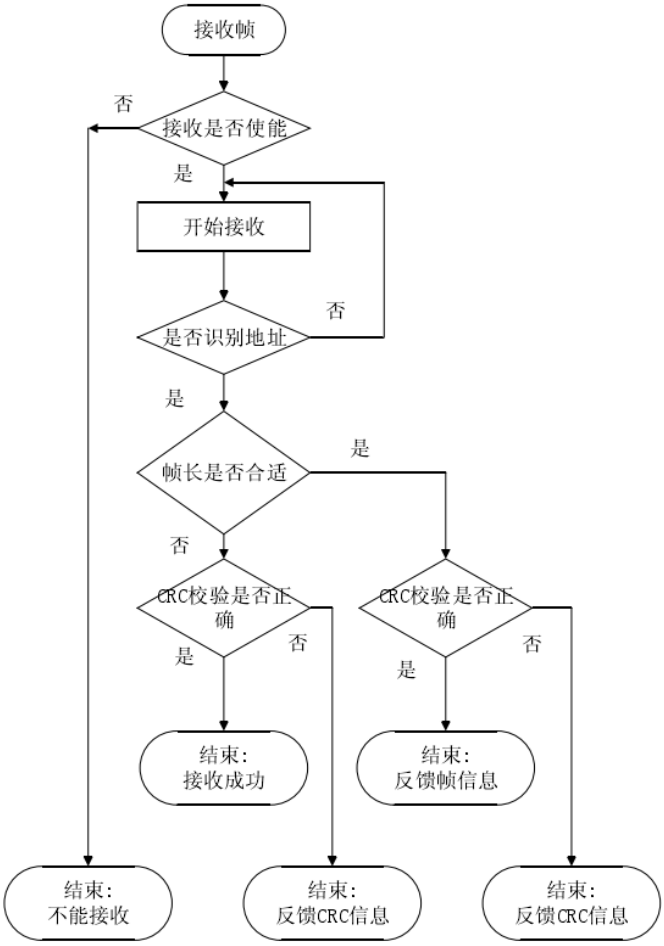


图 4　MAC 接收流程图

1.2 芯片介绍

KSZ8001L 是一款性能优良的支持自动协商和手动选择 10 / 100Mbps 的速度和全/半双工模式以太网控制器 ,完全适用于 IEEE802.3u 协议。除了具备其他以太网控制芯片所具有的一些基本功能外 ,还有他的独特优点: 工业级温度范围(0 ～ + 80 ℃);1.8V 工作电压 ,功耗低; 高度集成的设计 ,使用 KSZ8001L 可以将一个完整的以太网电路设计电路最小化 ,适合作为智能嵌入设备网络接口; 独特的 Packet Page 结构 ,可自动适应网络通信模式的改变 ,占用系统资源少 ,从而增加系统效率。

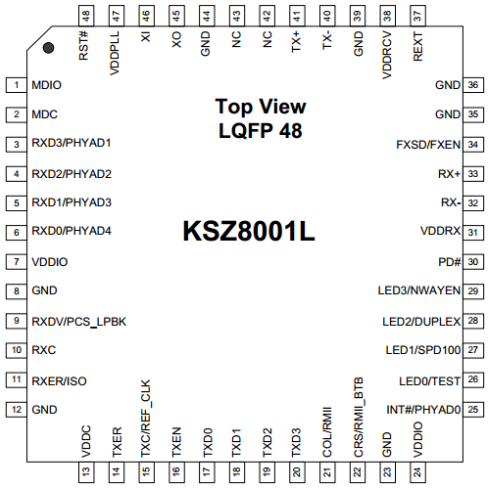


图 2　KSZ8001L

3 相关寄存器介绍

Register 0h – Basic Control ： 主要功能是做一些基本的控制，包括设备的重启，重置，传输速率选择等。

Register 1h – Basic Status ： 基本标志位，主要是传输状态标志，传输速率标志等。

Register 4h – Auto-Negotiation Advertisement ： 自动协商广播，包括暂停，传输速率协商等。

Register 5h – Auto-Negotiation Link Partner Ability ： 自动协商链接对方寄存器

Register 6h – Auto-Negotiation Expansion ： 自动协商扩展寄存器

Register 15h – RXER Counter ： 接收计数寄存器

Register 1bh – Interrupt Control/Status Register ： 中断控制/标志寄存器

Register 1dh – LinkMD Control/Status Register ： 链接控制/标志寄存器

Register 1eh – PHY Control ： 物理层控制寄存器

实现的功能

1.实现网络的连通性。

2.实现网络传输数据。

基本原理

Linux 系统网络栈架构图

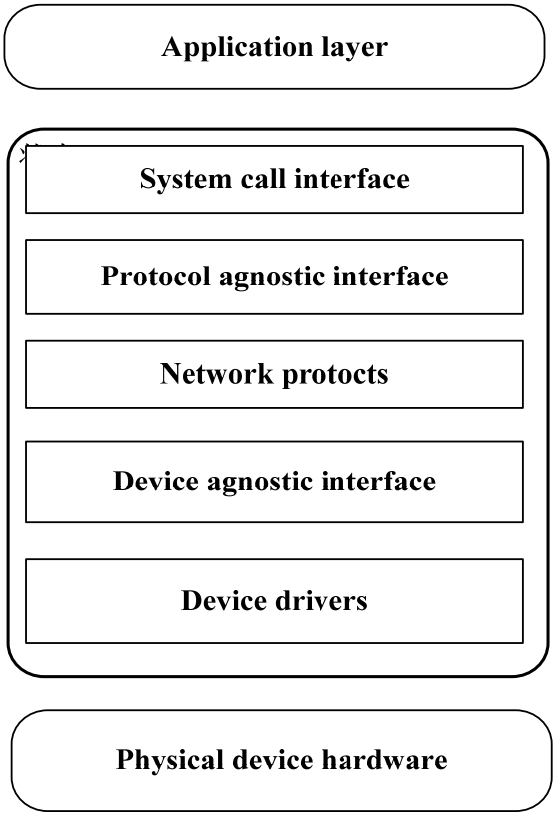


图 3　Linux 系统网络栈架构图

最上面是用户空间层，或称为应用层，其中定义了网络栈的用户。底部是物理设备，提供了对网络的连接能力（串口或诸如以太网之类的高速网络）。中间是内核空间，即网络子系统。顶部是系统调用接口，它简单地为用户空间的应用程序提供了一种访问内核网络子系统的方法。位于其下面的是一个协议无关层，它提供了一种通用方法来使用底层 传输层协议。然后是实际协议，在 Linux 中包括内嵌的协议 TCP、UDP，当然还有 IP。然后是另外一个协议无关层，提供了与各个设备驱动程序通信的通用接口，最下面是设备驱动程序本身。

3.1 IP 网络协议原理

TCP/IP 协议是一组包括 TCP 协议和 IP 协议，UDP 协议、ICMP 协议和其他一些协议的协议组。

TCP/IP 协议采用分层结构共分为四层，每一层独立完成指定功能，如图：

应用层（第四层）
传输层（第三层）
互联层（第二层）
网络接口层（第一层）

图 4　TCP/IP 协议层

网络接口层：负责接收和发送物理帧。

互联层：负责相邻接点之间的通信。

传输层：负责起点到终点的通信。

应用层：定义了应用程序使用互联网的规程。

3.2 TCP 协议

TCP 是面向连接的通信协议，通过三次握手建立连接，通讯完成时要拆除连接，由于 TCP 是面向连接的所以只能用于端到端的通讯。

TCP 提供的是一种可靠的数据流服务，采用“带重传的肯定确认”技术来实现传输的可靠性。TCP 还采用一种称为“滑动窗口”的方式进行流量控制，所谓窗口实际表示接收能力，用以限制发送方的发送速度。

如果 IP 数据包中有已经封好的 TCP 数据包，那么 IP 将把它们向‘上’传送到 TCP 层。TCP 将包排序并进行错误检查，同时实现虚电路间的连接。TCP 数据包中包括序号和确认，所以未按照顺序收到的包可以被排序，而损坏的包可以被重传。

TCP 将它的信息送到更高层的应用程序，例如 Telnet 的服务程序和客户程序。应用程序轮流将信息送回 TCP 层，TCP 层便将它们向下传送到 IP 层，设备驱动程序和物理介质，最后到接收方。

面向连接的服务（例如 Telnet、FTP、rlogin、X Windows 和 SMTP）需要高度的可靠性，所以它们使用了 TCP。DNS 在某些情况下使用 TCP（发送和接收域名数据库），但使用 UDP 传送有关单个主机的信息。

TCP 是因特网中的传输层协议，使用三次握手协议建立连接。当主动方发出 SYN 连接请求后，等待对方回答 SYN+ACK，并最终对对方的 SYN 执行 ACK 确认。这种建立连接的方法可以防止产生错误的连接，TCP 使用的流量控制协议是可变大小的滑动窗口协议。

TCP 三次握手的过程如下：

（1）客户端发送 SYN（SEQ=x）报文给服务器端，进入 SYN_SEND 状态。

（2）服务器端收到 SYN 报文，回应一个 SYN （SEQ=y）ACK(ACK=x+1）报文，进入 SYN_RECV 状态。

（3）客户端收到服务器端的 SYN 报文，回应一个 ACK(ACK=y+1）报文，进入 Established 状态。

三次握手完成，TCP 客户端和服务器端成功地建立连接，可以开始传输数据了。

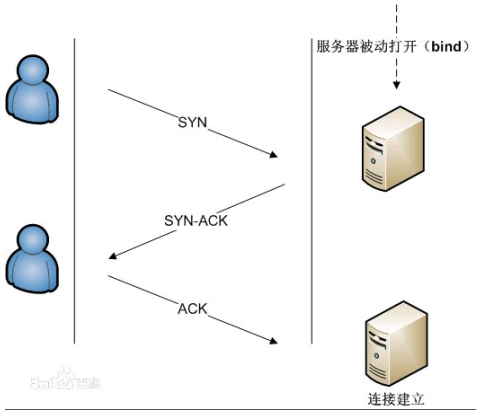


图 5　TCP 的三次握手

3.3 基于 TCP 的 socket 编程

在 server 端，server 首先启动，调用 socket( )创建套接字；然后调用 bind( ) 绑定 server 的地址（IP+port）；再调用 listen( )让 server 做好侦听准备，并规定好请求队列长度，然后 server 进入阻塞状态，等待 client 的连接请求；最后通过 accept( )来接收连接请求，并获得 client 的地址。当 accpet 接收到一个 client 发来的 connet 请求时，将生成一个新的 socket，用于传输数据。

在 client 端，client 在创建套接字并指定 client 的 socket 地址，然后就调用 connect( )和 server 建立连接。一旦连接建立成功，client 和 server 之间就可以通过调用 recv 和 send 来接收和发送数据。一旦数据传输结束，server 和 client 通过调用 close( )来关闭套接字。原理图如下：

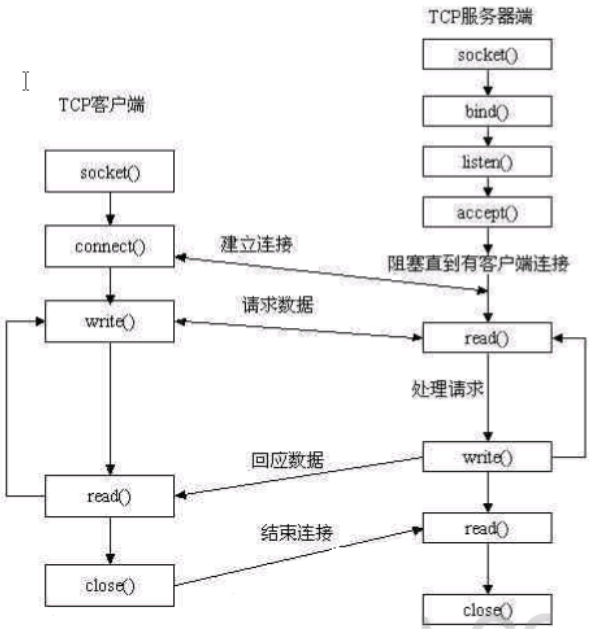


图 6　TCP 通信原理图

3.4 UDP 协议

UDP 是面向无连接的通讯协议，UDP 数据包括目的端口号和源端口号信息，由于通讯不需要连接，所以可以实现广播发送。

UDP 通讯时不需要接收方确认，属于不可靠的传输，可能会出现丢包现象，实际应用中要求程序员编程验证。

UDP 与 TCP 位于同一层，但它不管数据包的顺序、错误或重发。因此，UDP 不被应用于那些使用虚电路的面向连接的服务，UDP 主要用于那些面向查询---应答的服务，例如 NFS。相对于 FTP 或 Telnet，这些服务需要交换的信息量较小。使用 UDP 的服务包括 NTP（网络时间协议）和 DNS（DNS 也使用 TCP）。

UDP 是 OSI 参考模型中一种无连接的传输层协议，它主要用于不要求分组顺序到达的传输中，分组传输顺序的检查与排序由应用层完成，提供面向事务的简单不可靠信息传送服务。UDP 协议基本上是 IP 协议与上层协议的接口。UDP 协议适用端口分别运行在同一

台设备上的多个应用程序。

UDP 提供了无连接通信，且不对传送数据包进行可靠性保证，适合于一次传输少量数据，UDP 传输的可靠性由应用层负责。常用的 UDP 端口号有：

应用协议　端口号

DNS　　53

TFTP　　69

SNMP　　161

UDP 在 IP 报文中的位置如图所示：

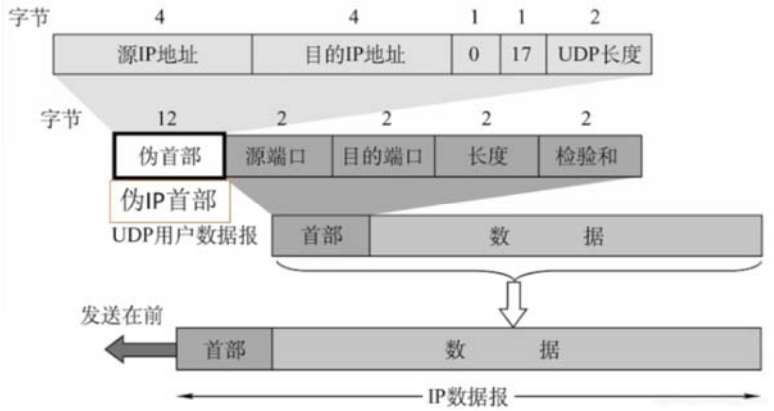


图 7　UDP 在 IP 报文位置

3.4 基于 UDP 的 socket 编程

基于 UDP 协议的无连接 C/S 的工作流程 在 server 端，server 首先启动，调用 socket( )创建套接字，然后调用 bind( )绑定 server 的地址（IP+port），调用 recvfrom( )等待接收数据。

在 client 端，先调用 socket()创建套接字，调用 sendto( )向 server 发送数据。

server 接收到 client 发来数据后，调用 sendto( )向 client 发送应答数据，client 调用 recv 接收 server 发来的应答数据。数据传输结束，server 和 client 通过调用 close( )关闭套接字。原理图如下图

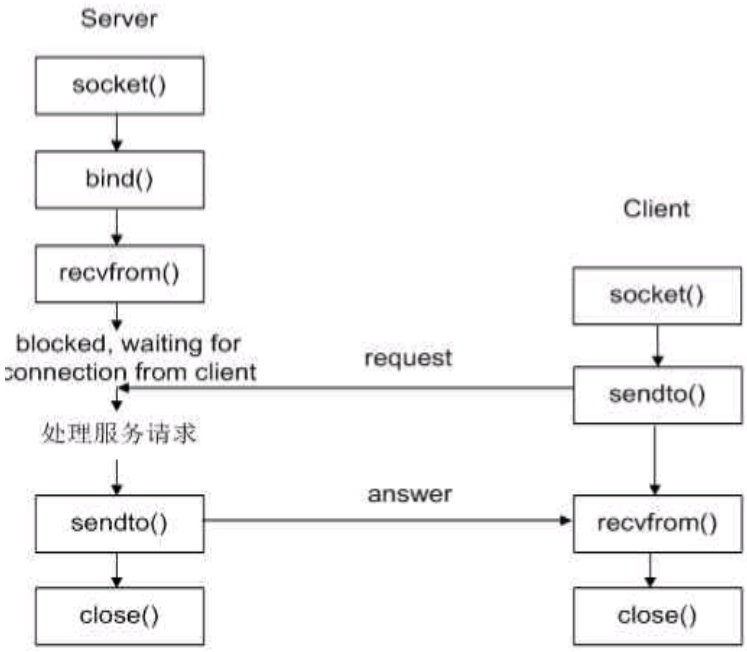


图 8　UDP 通信原理图

4.驱动开发的基本流程

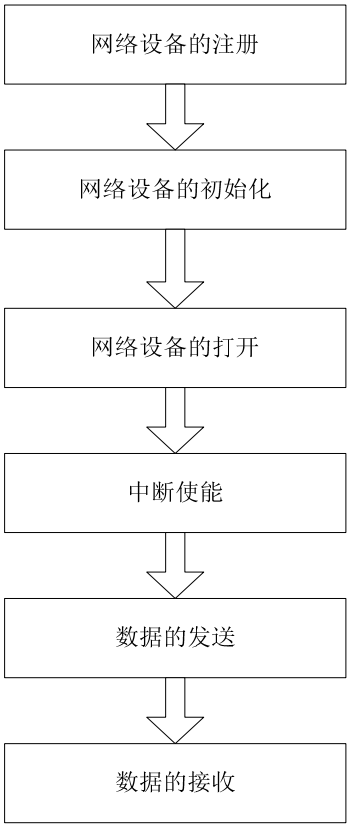


图 9　驱动开发基本流程图

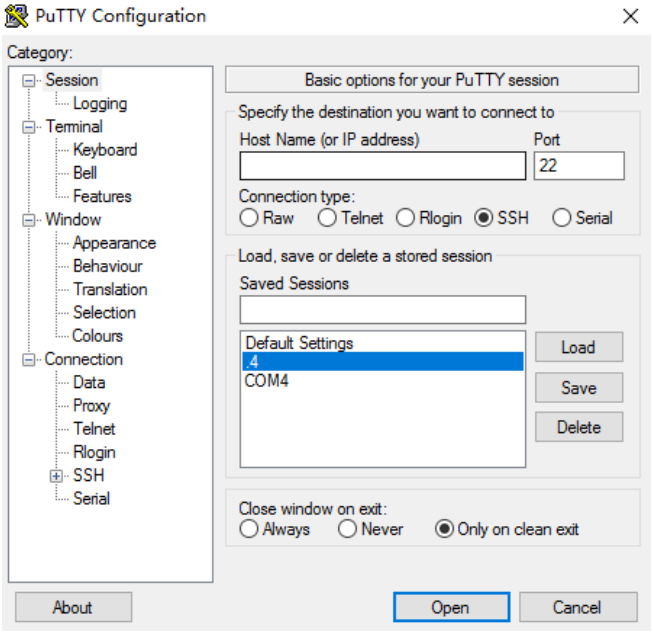
4.1 驱动程序中重要函数

static int __init davinci_emac_init(void)
//网络设备驱动的注册
static void __exit davinci_emac_exit(void)
//网络设备的注销
static int __init davinci_emac_probe(struct platform_device *pdev)
//网络设备的检测，包括以太网的分配，时钟使能等
static void emac_enable_interrupt(struct emac_dev *dev, int ack_eoi)
//使能中断
static int emac_open(struct emac_dev *dev, void *param)
//设备打开，包括使能设备的硬件资源，申请中断、DMA，激活发送列队等
static int emac_send(struct emac_dev *dev, net_pkt_obj *pkt, int channel, bool send_args)
//数据包的发送
static int emac_net_rx_cb(struct emac_dev *dev, net_pkt_obj *net_pkt_list,void *rx_args)
//数据包的接收

六. 实验步骤

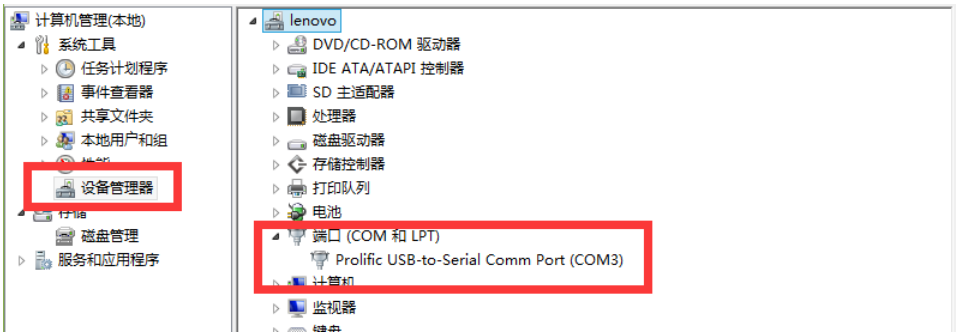
步骤 1：硬件连接

首先通过 putty 软件使用 ssh 通信方式登录到服务器，如下图一所示（在 Hostname 栏输入服务器的 ip 地址）：



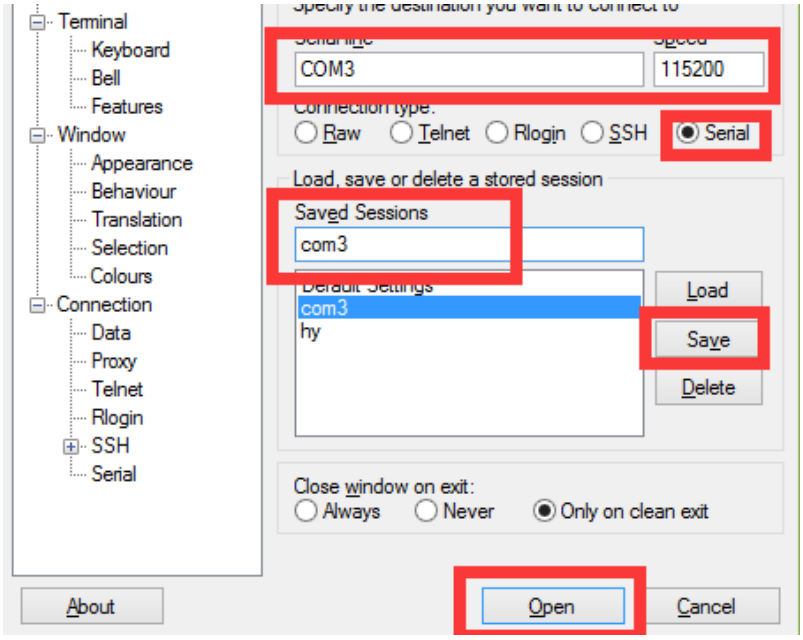
图一 打开 putty 连接

接着查看串口号，通过 putty 软件使用串口通信方式连接实验箱，如下图二所示：



图二 端口号查询

选择 putty 串口连接实验箱如三所：s



图三 putty 串口连接配置

输入启动参数，接着启动内核，如下图四所示：

```
DM365 EVM :>setenv bootargs mem=70M console=ttyS0,115200n8 root=/dev/nfs rw nfsr
oot=192.168.0.135:/home/st1/zh/filesys_test/ ip=192.168.0.109:192.168.0.135:192.
168.0.1:255.255.255.0::eth0:off eth=00:40:01:C1:56:19 video=davincifb:vid0=OFF:v
id1=OFF:osd0=640x480x16,600K:osd1=0x0x0,0K dm365_imp.oper_mode=0 davinci_capture
.device_type=1 davinci_enc_mngr.ch0_output=LCD
DM365 EVM :>boot

Loading from NAND 1GiB 3,3V 8-bit, offset 0x400000
   Image Name:   Linux-2.6.18-plc_pro500-davinci_
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    1995748 Bytes =  1.9 MB
   Load Address: 80008000
   Entry Point:  80008000
## Booting kernel from Legacy Image at 80700000 ...
   Image Name:   Linux-2.6.18-plc_pro500-davinci_
   Image Type:   ARM Linux Kernel Image (uncompressed)
```

图四 输入启动参数启动

输入用户名 root 登录实验箱如下图五所示：

```
zjut login: root


Welcome to MontaVista(R) Linux(R) Professional Edition 5.0.0 (0801921).

login[754]: root login on 'console'
/******Set QT environment********/
[root@zjut ~]#
```

图五 登陆实验箱

步骤 2：测试网络连通性

利用 ping 命令测试 ,输入命令：（在 pc 机上 ping 板子的 ip）

$ ping 192.168.0.109 （192.168.0.109 为板子设定的 ip,根据实际情况来）。

并观察响应时间和丢包率，判断连接是否正常，如果正常，说明 ARP ,IP , ICMP 协议正常。如下图 1 所示：（注意是通过 pc 输入 ping 命令）

```
st1@ubuntu:~$ ping 192.168.0.109
PING 192.168.0.109 (192.168.0.109) 56(84) bytes of data.
64 bytes from 192.168.0.109: icmp_req=1 ttl=64 time=0.669 ms
64 bytes from 192.168.0.109: icmp_req=2 ttl=64 time=0.812 ms
64 bytes from 192.168.0.109: icmp_req=3 ttl=64 time=1.02 ms
64 bytes from 192.168.0.109: icmp_req=4 ttl=64 time=0.883 ms
64 bytes from 192.168.0.109: icmp_req=5 ttl=64 time=0.851 ms
64 bytes from 192.168.0.109: icmp_req=6 ttl=64 time=0.563 ms
64 bytes from 192.168.0.109: icmp_req=7 ttl=64 time=0.515 ms
^C
--- 192.168.0.109 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6002ms
rtt min/avg/max/mdev = 0.515/0.760/1.028/0.171 ms
```

图 1 ping 命令

步骤 3：基于 TCP 的 socket 编程

编写好 TCP 代码，包括服务器端和客户端代码。在用户家目录中创建 ethenet 目录，进入该目录创建 tcpclient.c 和 tcpserver.c 文件，参考代码如下：

TCP-server 代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>

#define MAXLINE 7777 //服务器端口号

int main(int argc, char** argv){
    int    listenfd, connfd;
    struct sockaddr_in    servaddr;
    char    recv_buff[1024]={0};
```

```
char    send_msg[1024]={0};

int    n;

if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
       printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
         fflush(stdout);
       return 0;
}

memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(6666);

if( bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1){
       printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
         fflush(stdout);

       return 0;
}

if( listen(listenfd, 10) == -1){
       printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
       return 0;
}

printf("======waiting for client's request======\n");
 fflush(stdout);

 memset(send_msg,0,sizeof(send_msg));
       if( (connfd = accept(listenfd, (struct sockaddr*)NULL, NULL)) == -1){
             printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
           // continue;
       }
         while(1){
                              n = recv(connfd, recv_buff, MAXLINE, 0);
                              recv_buff[n] = '\0';
                              printf("recv msg from client: %s",recv_buff);
                                fflush(stdout);

                              if(strncmp(recv_buff,"QUIT",4)==0)//被动收到 quit 消息
                                {
                                   printf("client send : QUIT\n");
                                   fflush(stdout);
                                   close(listenfd);
                                  return 0;
```

```
                        }

                        printf("please input msg:\n");
                        fflush(stdout);

                        fgets(send_msg,sizeof(send_msg),stdin);
                        if(strncmp(send_msg,"QUIT",4)==0)//主动发起 quit 消息
                            {
                                printf("server send : QUIT\n");
                                fflush(stdout);

                                   send(connfd,"QUIT",4,0);
                                close(listenfd);
                                return 0;
                            }

                        send(connfd,send_msg,strlen(send_msg),0);
                }
        close(listenfd);
        return 0;
}
```

TCP-client 代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#define MAXLINE 4096

int main(int argc, char** argv){
        int     sockfd, n;
        char    recvline[4096], sendline[4096];
        struct sockaddr_in    servaddr;

        if( argc != 2){
                printf("usage: ./client <ipaddress>\n");
                  fflush(stdout);
                return 0;
        }

        if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
                printf("create socket error: %s(errno: %d)\n", strerror(errno),errno);
                  fflush(stdout);
```

```
        return 0;
}

memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(6666);
if( inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0){
        printf("inet_pton error for %s\n",argv[1]);
          fflush(stdout);
        return 0;
}

if( connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0){
        printf("connect error: %s(errno: %d)\n",strerror(errno),errno);
        fflush(stdout);
        return 0;
}


 while(1){
                 memset( recvline,0,sizeof( recvline));
                 memset( sendline,0,sizeof( sendline));

                printf("send msg to server: \n");
                 fflush(stdout);
                if(fgets(sendline, 4096, stdin)==NULL)    //主动结束
                {
                        printf("fgets is err\n");
                        fflush(stdout);
                 }
                 if(strstr(sendline,"QUIT")!=NULL)//send QUIT to server.
                 {
                        send(sockfd, sendline, strlen(sendline), 0);
                        printf("client send QUIT , client quit ok\n");
                        fflush(stdout);
                          close(sockfd);
                        return 0;
                 }

                if( send(sockfd, sendline, strlen(sendline), 0) < 0){
                      printf("send msg error: %s(errno: %d)\n", strerror(errno), errno);
                        fflush(stdout);
                      return 0;
                }
                if(recv(sockfd,    recvline, sizeof( recvline), 0)<0) //被动结束
                 {
                        printf("recv is fail\n");
```

```
                fflush(stdout);
            }
            else{
                printf("tcpsever send msg:%s\n",recvline);
                fflush(stdout);
                if(strstr(recvline,"QUIT")!=NULL)
                {
                printf("QUIT ok\n");
                fflush(stdout);
                break;
                }
            }
        }
    close(sockfd);
    return 0;
}
```

编辑完毕客户端和服务器端程序后分别保存输入命令编译：

```
$ arm_v5t_le-gcc tcpclient.c -o tcpclient_arm
$ gcc tcpsever.c -o tcpsever-gcc
```

第一行命令将编译生成可以在实验箱上运行的 tcp 客户端代码，第二行编译生成可以在 pc（虚拟机）或 linux 服务器上运行的代码，如果第二行改为如下所示，那么服务器代码也可以在实验向上运行。

```
$ arm_v5t_le- gcc tcpsever.c -o tcpsever _arm
```

下图实验过程以第一种情况为例说明。

然后将生成的可执行文件拷贝到文件系统的/home/stx/filesys_test/opt/dm365 下：

```
$ cp tcpclient_arm /home/stx/filesys_test/opt/dm365
```

在服务器端（pc 端）直接运行 tcpsever-gcc 代码等待客户端连接如下图 2 所示：

```
st1@ubuntu:~/ethenet$ ./tcpserver-gcc
======waiting for client's request======
```

图 2 等待客户端连接

接着在实验箱上运行客户端代码 tcpsever _arm 如下图 3 所示：

```
[root@zjut dm365]# ./tcpclient_arm 192.168.0.135
send msg to server:
```

图 3 运行客户端代码

（注意 192.168.0.135 是运行服务端程序的 pc 或者服务器的 ip 地址）

接着可以通过客户端向服务器发送消息：如下图 4 所示：

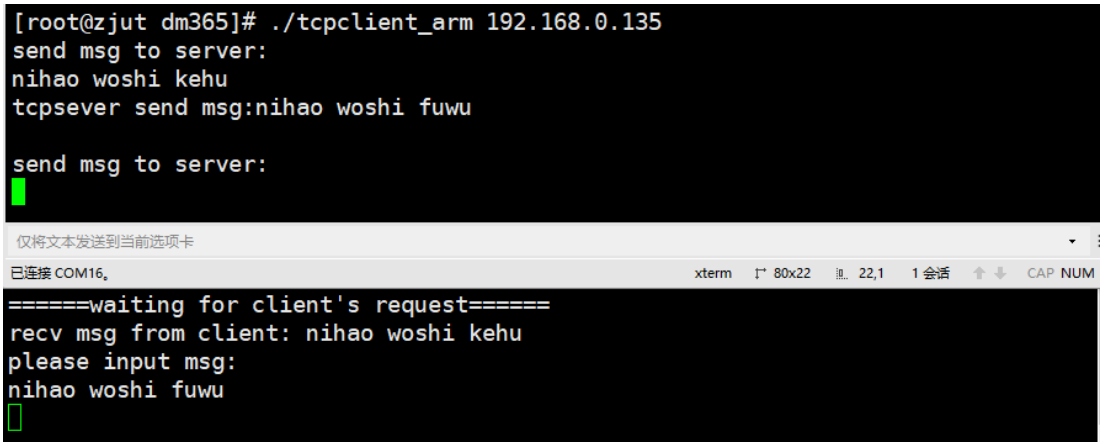


图 4 客户端向服务器发送消息

可以看到服务端收到客户端的消息并且可以回复消息。

步骤 4：基于 UDP 的 socket 编程

编写好 TCP 代码，包括服务器端和客户端代码。在用户家目录中创建 ethenet 目录，进入该目录创建 udpclient.c 和 udpserver.c 文件，参考代码如下：

UDP-server 代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<string.h>

#define MYPORT 8887
#define ERR_EXIT(m) \
        do { \
        perror(m); \
        exit(EXIT_FAILURE); \
        } while (0)

void echo_ser(int sock)
{
        char recvbuf[1024] = {0};
          char sendbuf[1024] = {0};
        struct sockaddr_in peeraddr;
        socklen_t peerlen;
        int n;

        while (1)
        {

                peerlen = sizeof(peeraddr);
```

```
        memset(recvbuf, 0, sizeof(recvbuf));
        n = recvfrom(sock, recvbuf, sizeof(recvbuf), 0,
                        (struct sockaddr *)&peeraddr, &peerlen);
        if (n <= 0)
        {
            if (errno == EINTR)
                continue;

            ERR_EXIT("recvfrom error");
        }else{
                if(strncmp(recvbuf,"QUIT",4)==0) //被动 QUIT
                {

                        printf("   从客户端接收：%s              UDPserver QUIT\n",recvbuf);
                        fflush(stdout);
                        break;

                }else{
                printf("接收到的数据：%s\n",recvbuf);
                fflush(stdout);
                }
        }
          memset(recvbuf,0,sizeof(recvbuf));

          printf("please input msg you want to send：\n");
          fflush(stdout);
          if(fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)
    {

                // printf("向客户端发送：%s\n",sendbuf);
                  //fflush(stdout);
                  if(strncmp(sendbuf,"QUIT",4)==0)
                  {
                        sendto(sock, sendbuf, strlen(sendbuf), 0, (struct sockaddr *)&peeraddr,
sizeof(peeraddr));
                        printf("   向客户端发送：%s              UDPserver QUIT \n",sendbuf);
                        fflush(stdout);

                        break;

                  }else{
                      sendto(sock, sendbuf, strlen(sendbuf), 0, (struct sockaddr *)&peeraddr,
sizeof(peeraddr));
                  }
          memset(recvbuf,0,sizeof(recvbuf));

          }
```

```
        }
        close(sock);
    }

    int main(void)
    {
        int sock;
        if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
            ERR_EXIT("socket error");

        struct sockaddr_in servaddr;
        memset(&servaddr, 0, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(MYPORT);
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

        printf("监听%d 端口\n",MYPORT);
          printf("服务器等待接收客户端的消息>>>\n");
          fflush(stdout);
        if (bind(sock, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
            ERR_EXIT("bind error");

        echo_ser(sock);

        return 0;
    }
```

UDP-client 代码：

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

#define MYPORT 8887
char* SERVERIP ;

#define ERR_EXIT(m) \
    do \
{ \
    perror(m); \
    exit(EXIT_FAILURE); \
```

```
    } while(0)

void echo_cli(int sock)
{
    struct sockaddr_in servaddr;
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(MYPORT);
    servaddr.sin_addr.s_addr = inet_addr(SERVERIP);

    int ret;
    char sendbuf[1024] = {0};
    char recvbuf[1024] = {0};
    while(1)
     {
          printf("please input msg you want to send：\n");
              fflush(stdout);
         if(fgets(sendbuf, sizeof(sendbuf), stdin) != NULL)
         {

             //printf("向服务器发送：%s\n",sendbuf);
               //fflush(stdout);
               if(strncmp(sendbuf,"QUIT",4)==0)
               {
                    sendto(sock, sendbuf, strlen(sendbuf), 0, (struct sockaddr *)&servaddr,
sizeof(servaddr));
                    printf("向服务器发送：%s . client QUIT \n",sendbuf);
                    fflush(stdout);

                    break;

               }else{
             sendto(sock, sendbuf, strlen(sendbuf), 0, (struct sockaddr *)&servaddr, sizeof(servaddr));
               }

         }
               memset(sendbuf, 0, sizeof(sendbuf));
             ret = recvfrom(sock, recvbuf, sizeof(recvbuf), 0, NULL, NULL);
             if (ret == -1)
             {
                  if (errno == EINTR)
                       continue;
                  ERR_EXIT("recvfrom");
             }
               if(strncmp(recvbuf,"QUIT",4)==0) //被动 QUIT
               {
```

```
                    printf("从服务器接收：%s，  client QUIT\n",recvbuf);
                    fflush(stdout);

                    break;

                }
                printf("从服务器接收：%s\n",recvbuf);
                fflush(stdout);

                memset(recvbuf, 0, sizeof(recvbuf));

        }
    close(sock);
}

int main( int argc,char **argv)
{
    int sock;
    if(argc !=2){
            printf("usage: ./udpxxx 192.168.x.x\n");
            return -1;
     }
    SERVERIP =argv[1];//argv[1] is ip input
    if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
        ERR_EXIT("socket");

    echo_cli(sock);

    return 0;
}
```

编辑完毕客户端和服务器端程序后分别保存输入命令编译：

```
$ arm_v5t_le-gcc udpclient.c -o udpclient_arm
$ gcc tcpsever.c -o udpsever-gcc
```

第一行命令将编译生成可以在实验箱上运行的 tcp 客户端代码，第二行编译生成可以在 pc（虚拟机）或 linux 服务器上运行的代码，如果第二行改为如下所示，那么服务器代码也可以在实验向上运行。

```
$ arm_v5t_le- gcc udpsever.c -o udpsever _arm
```

下图实验过程以第一种情况为例说明。

然后将生成的可执行文件拷贝到文件系统的/home/stx/filesys_test/opt/dm365 下：

```
$ cp udpclient_arm /home/stx/filesys_test/opt/dm365
```

在服务器端（pc 端）直接运行 tcpsever-gcc 代码等待客户端连接如下图 5 所示：

```
st1@ubuntu:~/ethenet$ ./udpserver-gcc
监听8887端口
服务器等待接收客户端的消息>>>
```

图 5 等待客户端连接

在实验箱执行 udp 客户端命令并且输入发送数据如下图 6 所示：

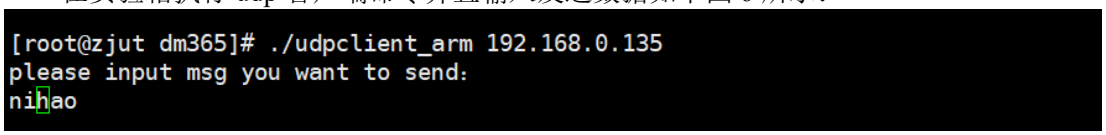


图 6 输入发送数据

通信效果如下图 7 所示：

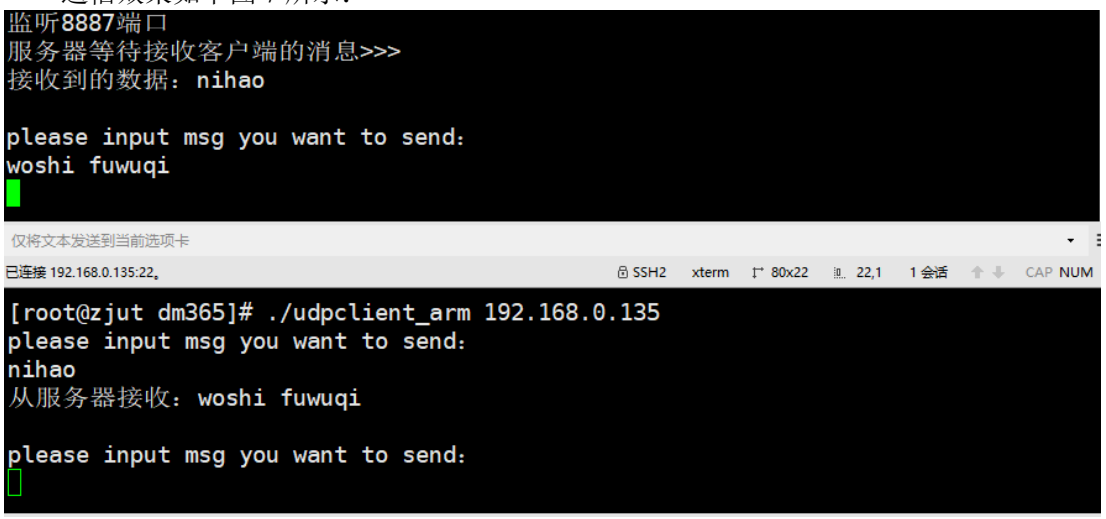


图 7 通信效果

实验结束。