

实验二十七：LWIP_NETIO 实验——以太网测速

一、实验目的与意义

- 1、了解 LWIP 协议栈和 NETIO 的结构
- 2、了解 NETIO 特征
- 3、了解 UCOSII 的使用方法
- 4、掌握 NETIO 的使用方法
- 5、掌握 KEIL MDK 集成开发环境使用方法

二、实验设备及平台

- 1、iCore4 双核心板
- 2、JLINK（或相同功能）仿真器
- 3、Micro USB 线缆
- 4、网线
- 5、Keil MDK 开发平台
- 6、装有 WIN XP（及更高版本）系统的计算机

三、实验原理

1、NETIO 简介

网络基准测量工具 NETIO 是通过 NetBIOS、UDP 和 TCP 协议测量网络净生产量的网络基准(Unix 只支持 TCP 和 UDP)使用各种各样的不同的包大小。

2、LwIP 简介

LwIP 是 Light Weight (轻型)IP 协议，有无操作系统的支持都可以运行。LwIP 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用，它只需十几 KB 的 RAM 和 40K 左右的 ROM 就可以运行，这使 LwIP 协议栈适合在低端的嵌入式系统中使用。

LwIP 协议栈主要关注的是怎么样减少内存的使用和代码的大小，这样就可以让 LwIP 适用于资源有限的小型平台例如嵌入式系统。为了简化处理过程和内存要求，LwIP 对 API 进行了裁减，可以不需要复制一些数据。

LwIP 提供三种 API: 1)RAW API 2)LwIP API 3)BSD API。

RAW API 把协议栈和应用程序放到一个进程里边, 该接口基于函数回调技术, 使用该接口的应用程序可以不用进行连续操作。不过, 这会使应用程序编写难度加大且代码不易被理解。为了接收数据, 应用程序会向协议栈注册一个回调函数。该回调函数与特定的连接相关联, 当该关联的连接到达一个信息包, 该回调函数就会被协议栈调用。这既有优点也有缺点。优点是既然应用程序和 TCP/IP 协议栈驻留在同一个进程中, 那么发送和接收数据就不再产生进程切换。主要缺点是应用程序不能使自己陷入长期的连续运算中, 这样会导致通讯性能下降, 原因是 TCP/IP 处理与连续运算是不能并行发生的。这个缺点可以通过把应用程序分为两部分来克服, 一部分处理通讯, 一部分处理运算。

LwIP API 把接收与处理放在一个线程里面。这样只要处理流程稍微被延迟, 接收就会被阻塞, 直接造成频繁丢包、响应不及时等严重问题。因此, 接收与协议处理必须分开。LwIP 的作者显然已经考虑到了这一点, 他为我们提供了 `tcpip_input()` 函数来处理这个问题, 虽然他并没有在 `rawapi` 一文中说明。讲到这里, 读者应该知道 `tcpip_input()` 函数投递的消息从哪里来的答案了吧, 没错, 它们来自于由底层网络驱动组成的接收线程。我们在编写网络驱动时, 其接收部分以任务的形式创建。数据包到达后, 去掉以太网包头得到 IP 包, 然后直接调用 `tcpip_input()` 函数将其投递到 `mbox` 邮箱。投递结束, 接收任务继续下一个数据包的接收, 而被投递得 IP 包将由 TCPIP 线程继续处理。这样, 即使某个 IP 包的处理时间过长也不会造成频繁丢包现象的发生。这就是 LwIP API。

BSD API 提供了基于 `open-read-write-close` 模型的 UNIX 标准 API, 它的最大特点是使应用程序移植到其它系统时比较容易, 但用在嵌入式系统中效率比较低, 占用资源多。这对于我们的嵌入式应用有时是不能容忍的。

其主要特性如下:

- (1) 支持多网络接口下的 IP 转发;
- (2) 支持 ICMP 协议;
- (3) 包括实验性扩展的 UDP(用户数据报协议);
- (4) 包括阻塞控制、RTT 估算、快速恢复和快速转发的 TCP(传输控制协议);
- (5) 提供专门的内部回调接口(Raw API), 用于提高应用程序性能;
- (6) 可选的 Berkeley 接口 API (在多线程情况下使用);
- (7) 在最新的版本中支持 ppp;

(8) 新版本中增加了 IP fragment 的支持;

(9) 支持 DHCP 协议,动态分配 ip 地址。

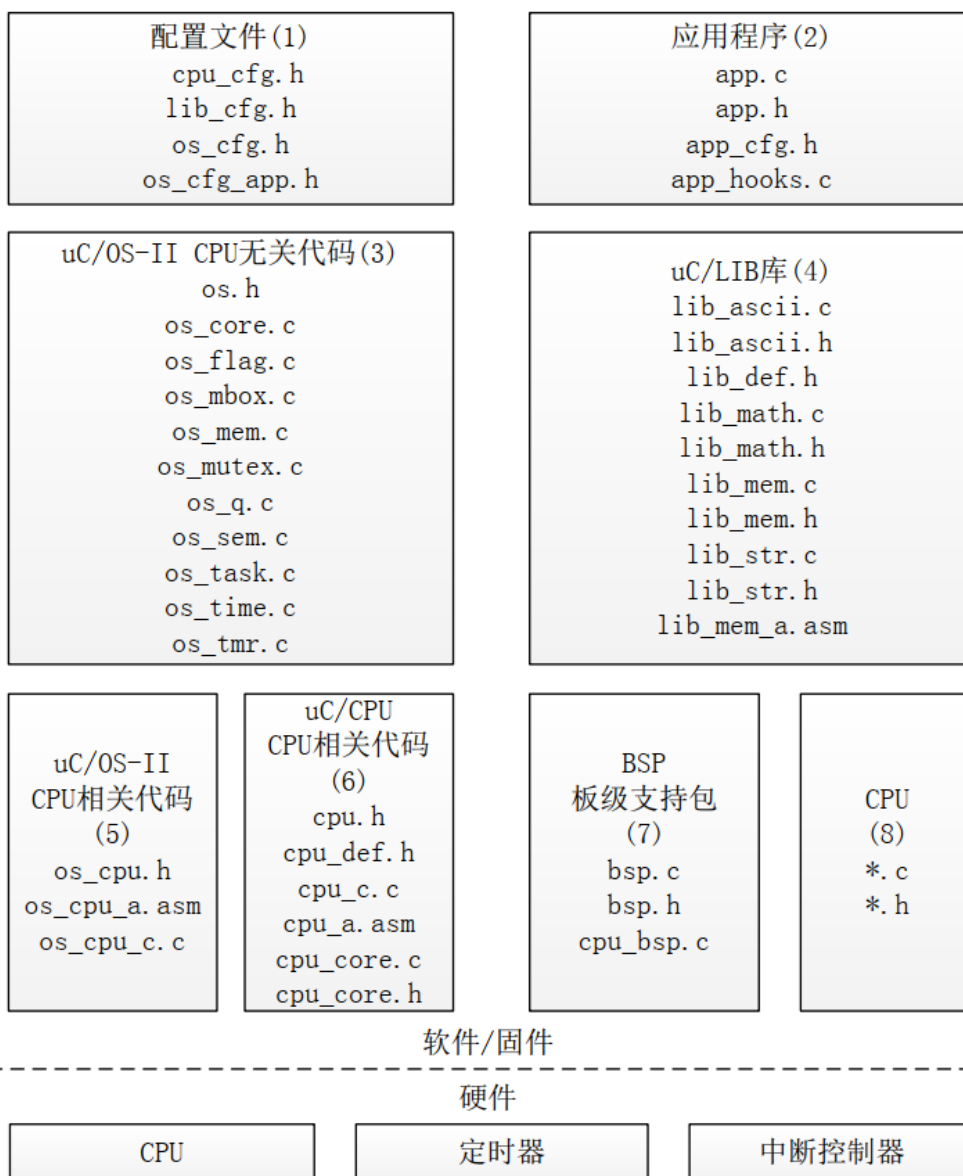
3、UCOSII 简介

UCOSII 的前身是 UCOS,最早出自于 1992 年美国嵌入式系统专家 JeanJ.Labrosse 在《嵌入式系统编程》杂志的 5 月和 6 月刊上刊登的文章连载,并把 UCOS 的源码发布在该杂志的 BBS 上。目前最新的版本: UCOSIII 已经出来,但是现在使用最为广泛的还是 UCOSII。

UCOSII 是一个可以基于 ROM 运行的、可裁减的、抢占式、实时多任务内核,具有高度可移植性,特别适合于微处理器和控制器,是和很多商业操作系统性能相当的实时操作系统(RTOS)。为了提供最好的移植性能,UCOSII 最大程度上使用 ANSI C 语言进行开发,并且已经移植到近 40 多种处理器体系上,涵盖了从 8 位到 64 位各种 CPU(包括 DSP)。

UCOSII 是专门为计算机的嵌入式应用设计的,绝大部分代码是用 C 语言编写的。CPU 硬件相关部分是用汇编语言编写的、总量约 200 行的汇编语言部分被压缩到最低限度,为的是便于移植到任何一种其它的 CPU 上。用户只要有标准的 ANSI 的 C 交叉编译器,有汇编器、连接器等软件工具,就可以将 UCOSII 嵌入到开发的产品中。UCOSII 具有执行效率高、占用空间小、实时性能优良和可扩展性强等特点,最小内核可编译至 2KB。UCOSII 已经移植到了几乎所有知名的 CPU 上。

UCOSII 构思巧妙、结构简洁精练、可读性强,同时又具备了实时操作系统的全部功能,虽然它只是一个内核,但非常适合初次接触嵌入式实时操作系统的朋友,可以说是麻雀虽小,五脏俱全。UCOSII 体系结构如图所示:



(1) 这部分是系统配置文件，用来配置所需的系统功能，比如需要用到的 UCOSII 的模块、时钟频率等等。

(2) 这部分为用户的应用程序，即使用 UCOSII 完成的应用层代码，文件不一定命名为 app.c，可以命名为其他的。注意，app_hooks.c 里面是钩子函数的应用层代码，app_cfg.h 是与 APP 配置有关的，这个是 Micrium 公司提供的模板，不使用的話就可以直接删掉。

(3) 这部分是 UCOSII 的核心源码，它们是与处理器无关的代码，都是由高度可移植的 ANSIC 编写的。

(4) Micrium 重写了 stdlib 库中的一些函数，如内存复制，字符串相关函数等。这样做的目的是为了保证在不同应用程序和编译器之间的可移植性。

(5) 这部分的文件需要根据不同的 CPU 架构去做修改，也就是移植的过程。从这里可看

出移植的真正核心就是这三个文件的修改。

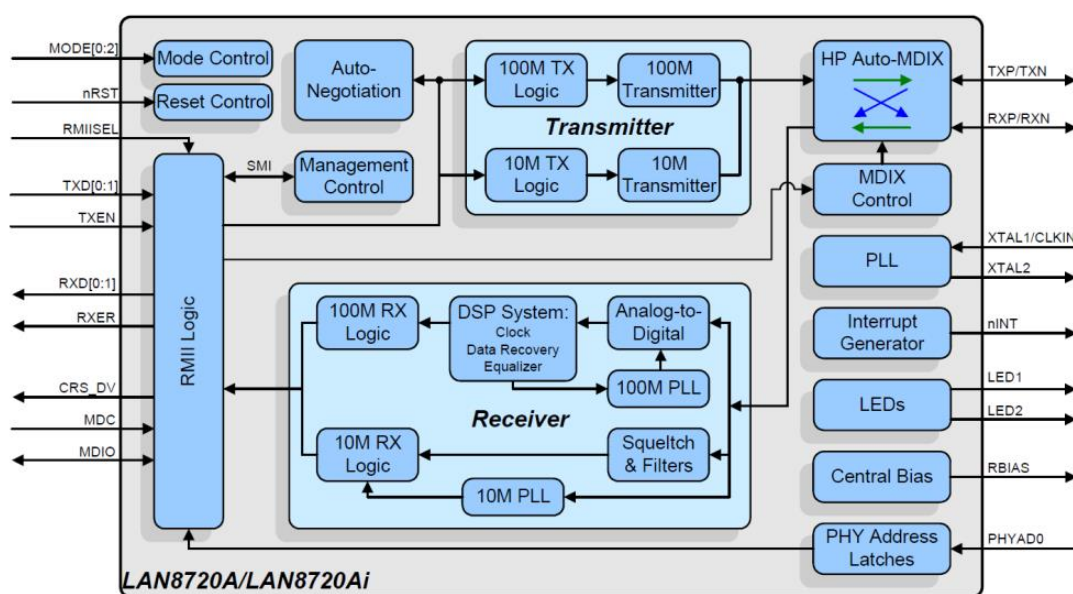
(6) 此部分是 Micrium 官方封装起来的 CPU 相关功能代码，比如打开和关闭中断等。

(7) 板级支持包(BSP)，说白了就是外设驱动代码，根据需求用户自行编写，不一定要用 bsp.c 和 bsp.h 这样的文件命名。cpu_bsp.c 是与 cpu 有关的驱动。

(8) CPU 厂商提供的针对本公司 CPU 所制作的库函数，比如 ST 针对 STM32 提供的 STD 和 HAL 这种库函数。

4、LAN8720A 简介

LAN8720A 功能框图如图所示：



LAN8720A 是低功耗的 10/100M 以太网 PHY 层芯片, I/O 引脚电压符合 IEEE802.3-2005 标准, 支持通过 RMII 接口与以太网 MAC 层通信, 内置 10-BASE-T/100BASE-TX 全双工传输模块, 支持 10Mbps 和 100Mbps。

LAN8720A 可以通过自协商的方式与目的主机最佳的连接方式(速度和双工模式), 支持 HP Auto-MDIX 自动翻转功能, 无需更换网线即可将连接更改为直连或交叉连接。LAN8720A 的主要特点如下:

- 高性能的 10/100M 以太网传输模块
- 支持 RMII 接口以减少引脚数
- 支持全双工和半双工模式
- 两个状态 LED 输出
- 可以使用 25M 晶振以降低成本



- ### 5、原理图

1、主函数

```
int main(void)
{
    system_clock.initialize();           //系统时钟初始化
    led.initialize();                   //LED 初始化
    adc.initialize();                   //ADC 初始化
    delay.initialize(216);              //延时初始化
    my_malloc.initialize(SRAMIN);       //动态内存初始化
    usart6.initialize(115200);          //串口波特率设置
    usart6.printf("\033[1;32;40m");     //设置字体终端为绿色
    usart6.printf("\r\nHello, I am iCore4!\r\n\r\n"); //串口信息输出

    OSInit();                           //UCOS 初始化

    while(lwip.initialize())            //lwip 初始化
    {
        LED_RED_ON;
        usart6.printf("\r\nETH initialize error!\r\n\r\n"); //ETH 初始化失败
    }
    netio.initialize();                 //netio 初始化
    tcp.initialize();                   //tcp 初始化

    OSTaskCreate(start_task, (void*)0, (OS_STK*)&START_TASK_STK[START_TASK_SIZE-1], START_TASK_PRIO);
    OSStart();                          //开启 UCOS
}
```

2、开始任务

```
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr;
    pdata = pdata ;

    OSStatInit();                      //初始化统计任务
    OS_ENTER_CRITICAL();               //关中断

#ifdef LWIP_DHCP
    lwip_comm_dhcp_creat();           //创建 DHCP 任务
#endif
#ifdef LWIP_DNS
    my_dns.initialize();              //创建 DNS 任务
#endif
#endif
```

```
#endif

OSTaskCreate(led_task, (void*)0, (OS_STK*)&LED_TASK_STK[LED_STK_SIZE-1], LED_TASK_PRIO); //创建 LED 任务
OSTaskCreate(display_task, (void*)0, (OS_STK*)&DISPLAY_TASK_STK[DISPLAY_STK_SIZE-1], DISPLAY_TASK_PRIO); //显示任务
OSTaskSuspend(OS_PRIO_SELF); //挂起 start_task 任务

OS_EXIT_CRITICAL(); //开中断
}
```

3、动态内存初始化

```
//内存管理初始化
void my_mem_init(u8 memx)
{
    memset(mallco_dev.memmap[memx], 0, memtblsize[memx]*4);
    //内存状态表数据清零
    mallco_dev.memrdy[memx]=1;
    //内存管理初始化 OK
}
```

4、LwIP 初始化

```
//LWIP 初始化(LWIP 启动的时候使用)
//返回值:0,成功
// 1,内存错误
// 2,LAN8720 初始化失败
// 3,网卡添加失败.
u8 lwip_comm_init(void)
{
    OS_CPU_SR cpu_sr;
    struct netif *Netif_Init_Flag; //调用 netif_add() 函数时的返回值,用于判断网络初始化是否成功

    struct ip_addr ipaddr; //ip 地址
    struct ip_addr netmask; //子网掩码
    struct ip_addr gw; //默认网关
    if(lan8720.memory_malloc())return 1; //内存申请失败
    if(lwip_comm_mem_malloc())return 1; //内存申请失败
    if(lan8720.initialize())return 2; //初始化 LAN8720 失败
    tcpip_init(NULL, NULL); //初始化 tcp ip 内核,该函数里面会创建 tcpip_thread 内核任务
    lwip_comm_default_ip_set(&lwipdev); //设置默认 IP 等信息
    #if LWIP_DHCP //使用动态 IP
        ipaddr.addr = 0;
    #endif
}
```



```
netmask.addr = 0;
gw.addr = 0;

#else //使用静态 IP
    IP4_ADDR(&ipaddr, lwipdev.ip[0], lwipdev.ip[1], lwipdev.ip[2], lwipdev.ip[3]);
    IP4_ADDR(&netmask, lwipdev.netmask[0], lwipdev.netmask[1], lwipdev.netmask[2], lwipdev.netmask[3]);
    IP4_ADDR(&gw, lwipdev.gateway[0], lwipdev.gateway[1], lwipdev.gateway[2], lwipdev.gateway[3]);

    usart6.printf("网卡 en 的 MAC 地址为:.....%d.%d.%d.%d.%d.%d\r\n", lwipdev.mac[0], lwipdev.mac[1], lwipdev.mac[2], lwipdev.mac[3], lwipdev.mac[4], lwipdev.mac[5]);

    usart6.printf("静态 IP 地址.....%d.%d.%d.%d\r\n", lwipdev.ip[0], lwipdev.ip[1], lwipdev.ip[2], lwipdev.ip[3]);

    usart6.printf("子网掩码.....%d.%d.%d.%d\r\n", lwipdev.netmask[0], lwipdev.netmask[1], lwipdev.netmask[2], lwipdev.netmask[3]);

    usart6.printf("默认网关.....%d.%d.%d.%d\r\n", lwipdev.gateway[0], lwipdev.gateway[1], lwipdev.gateway[2], lwipdev.gateway[3]);
#endif

    Netif_Init_Flag=netif_add(&lwip_netif, &ipaddr, &netmask, &gw, NULL, &ethernetif_init, &tcpip_input); //向网卡列表中添加一个网口
    #if LWIP_DNS
        dns_init();
    #endif

    if(Netif_Init_Flag==NULL) return 3; //网卡添加失败
    else //网口添加成功后, 设置 netif 为默认值, 并且打开 netif 网口
    {
        netif_set_default(&lwip_netif); //设置 netif 为默认网口
        netif_set_up(&lwip_netif); //打开 netif 网口
    }

    return 0; //操作 OK.
}
```

5、NETIO 初始化

```
static INT8U netio_server_init(void)
{
    INT8U res;
    OS_CPU_SR cpu_sr;
    OS_ENTER_CRITICAL();    //关中断
    res = OSTaskCreate(netio_init, (void*)0, (OS_STK*)&NETIO_TASK_STK[NETIO
_STK_SIZE-1], NETIO_PRIO);
    OS_EXIT_CRITICAL();    //开中断
    return res; //返回值:0
}
```

```
static void netio_init(void *arg)
{
    struct tcp_pcb *pcb;

    pcb = tcp_new();
    tcp_bind(pcb, IP_ADDR_ANY, 18767);
    pcb = tcp_listen(pcb);
    tcp_accept(pcb, netio_accept);
}
```

6、TCP 初始化

```
static INT8U tcp_server_init(void) //创建 TCP 服务器线程
{
    INT8U res;
    OS_CPU_SR cpu_sr;

    OS_ENTER_CRITICAL();    //关中断
    res = OSTaskCreate(tcp_server_thread, (void*)0, (OS_STK*)&TCPSERVER_TAS
K_STK[TCPSERVER_STK_SIZE-1], TCPSERVER_PRIO); //创建 TCP 服务器线程
    OS_EXIT_CRITICAL();    //开中断

    return res; //返回值:0 TCP 服务器创建成功
}
```

7、NETIO 关闭

```
static void netio_close(void *arg, struct tcp_pcb *pcb)
{
    err_t err;

    struct netio_state *ns = arg;

    ns->state = NETIO_STATE_DONE;    //标记 NETIO 不处于任何状态
    tcp_recv(pcb, NULL);
    err = tcp_close(pcb);            //关闭连接

    if (err != ERR_OK) {
        tcp_recv(pcb, netio_recv);    //关闭失败，稍后重试
    } else {
        //关闭成功
#ifdef NETIO_USE_STATIC_BUF != 1
        if (ns->buf_ptr != NULL) {
            mem_free(ns->buf_ptr);
        }
#endif
    }
}
```

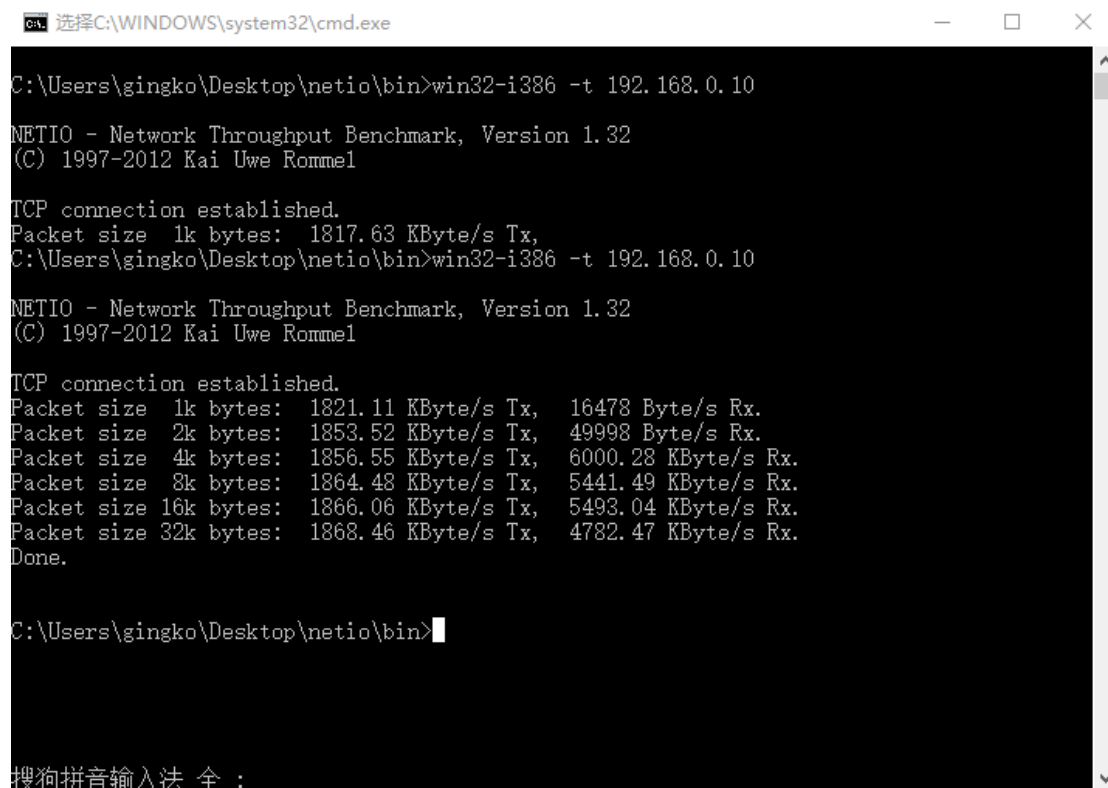
```
tcp_arg(pcb, NULL);    //注销掉参数
tcp_poll(pcb, NULL, 0);    //注销掉轮训函数
tcp_sent(pcb, NULL);    //注销掉发送函数
if (arg != NULL) {
    mem_free(arg);    //释放 arg 的内存
}
}
```

五、实验步骤

- 1、把仿真器与 iCore4 的 SWD 调试口相连（直接相连或者通过转接器相连）；
- 2、将跳线帽插在 USB UART；
- 3、把 iCore4（USB UART）通过 Micro USB 线与计算机相连，为 iCore4 供电；
- 4、把 iCore4 网口通过网线与计算机网口相连；
- 5、打开 Keil MDK 开发环境，并打开本实验工程；
- 6、烧写程序到 iCore4 上；
- 7、打开 netio 文件夹，双击运行 cmd.reg 注册表，右击 bin 文件夹，选择在此位置打开 cmd，输入命令：win32-i386 -t 192.168.0.10.
- 8、也可以进入 Debug 模式，单步运行或设置断点验证程序逻辑。

六、实验现象

实验现象如图所示：



```
C:\Users\gingko\Desktop\netio\bin>win32-i386 -t 192.168.0.10

NETIO - Network Throughput Benchmark, Version 1.32
(C) 1997-2012 Kai Uwe Rommel

TCP connection established.
Packet size 1k bytes: 1817.63 KByte/s Tx,
C:\Users\gingko\Desktop\netio\bin>win32-i386 -t 192.168.0.10

NETIO - Network Throughput Benchmark, Version 1.32
(C) 1997-2012 Kai Uwe Rommel

TCP connection established.
Packet size 1k bytes: 1821.11 KByte/s Tx, 16478 Byte/s Rx.
Packet size 2k bytes: 1853.52 KByte/s Tx, 49998 Byte/s Rx.
Packet size 4k bytes: 1856.55 KByte/s Tx, 6000.28 KByte/s Rx.
Packet size 8k bytes: 1864.48 KByte/s Tx, 5441.49 KByte/s Rx.
Packet size 16k bytes: 1866.06 KByte/s Tx, 5493.04 KByte/s Rx.
Packet size 32k bytes: 1868.46 KByte/s Tx, 4782.47 KByte/s Rx.
Done.

C:\Users\gingko\Desktop\netio\bin>
```