第五十七章 ENC28J60 网络实验

本章，我们将向大家介绍 ALIENTEK ENC28J60 网络模块及其使用。本章，我们将使用 ALIENTEK ENC28J60 网络模块和 uIP 1.0 实现：TCP 服务器、TCP 客服端以及 WEB 服务器等三个功能。本章分为如下几个部分：

57.1 ENC28J60 以及 uIP 简介
57.2 硬件设计
57.3 软件设计
57.4 下载验证

57.1 ENC28J60 以及 uIP 简介

本章我们需要用到 ENC28J60 以太网控制器和 uIP 1.0 以太网协议栈。接下来分别介绍这两个部分。

57.1.1 ENC28J60 简介

ENC28J60 是带有行业标准串行外设接口（Serial Peripheral Interface，SPI）的独立以太网控制器。它可作为任何配备有 SPI 的控制器的以太网接口。ENC28J60 符合 IEEE 802.3 的全部规范，采用了一系列包过滤机制以对传入数据包进行限制。 它还提供了一个内部 DMA 模块，以实现快速数据吞吐和硬件支持的 IP 校验和计算。 与主控制器的通信通过两个中断引脚和 SPI 实现，数据传输速率高达 10 Mb/s。两个专用的引脚用于连接 LED，进行网络活动状态指示。

ENC28J60 的主要特点如下：

- 兼容 IEEE802.3 协议的以太网控制器
- 集成 MAC 和 10 BASE-T 物理层
- 支持全双工和半双工模式
- 数据冲突时可编程自动重发
- SPI 接口速度可达 10Mbps
- 8K 数据接收和发送双端口 RAM
- 提供快速数据移动的内部 DMA 控制器
- 可配置的接收和发送缓冲区大小
- 两个可编程 LED 输出
- 带 7 个中断源的两个中断引脚
- TTL 电平输入
- 提供多种封装：SOIC/SSOP/SPDIP/QFN 等

ENC28J60 的典型应用电路如图 57.1.1.1 所示：

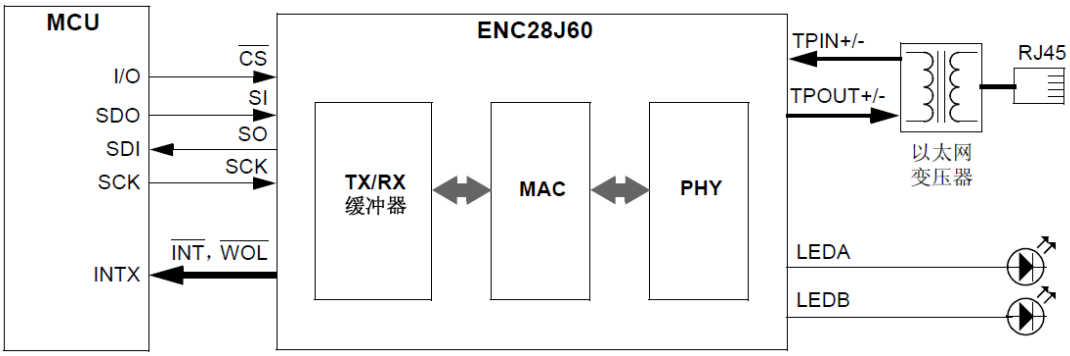

图 57.1.1.1 ENC28J60 典型应用电路

ENC28J60 由七个主要功能模块组成：

1） SPI 接口，充当主控制器和 ENC28J60 之间通信通道。

2） 控制寄存器，用于控制和监视 ENC28J60。

3） 双端口 RAM 缓冲器，用于接收和发送数据包。

4） 判优器，当 DMA、发送和接收模块发出请求时对 RAM 缓冲器的访问进行控制。

5） 总线接口，对通过 SPI 接收的数据和命令进行解析。

6） MAC(Medium Access Control)模块，实现符合 IEEE 802.3 标准的 MAC 逻辑。

7） PHY(物理层)模块，对双绞线上的模拟数据进行编码和译码。

ENC28J60 还包括其他支持模块，诸如振荡器、片内稳压器、电平变换器（提供可以接受 5V 电压的 I/O 引脚）和系统控制逻辑。

ENC28J60 的功能框图如图 57.1.1.2 所示：

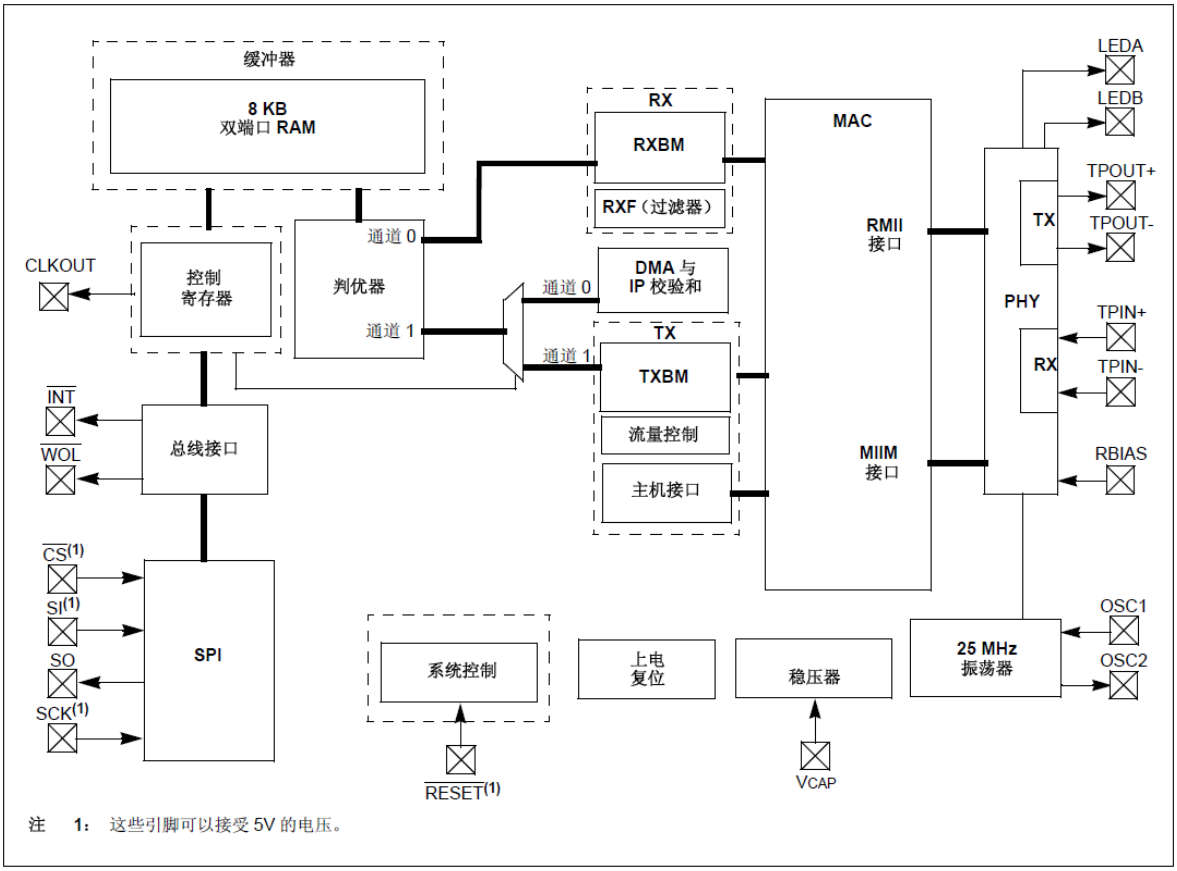

图 57.1.1.2 ENC28J60 功能框图

ALIENTEK ENC28J60 网络模块采用 ENC28J60 作为主芯片，单芯片即可实现以太网接入，利用该模块，基本上只要是个单片机，就可以实现以太网连接。ALIENTEK ENC28J60 网络模块原理图如图 57.1.1.3 所示：

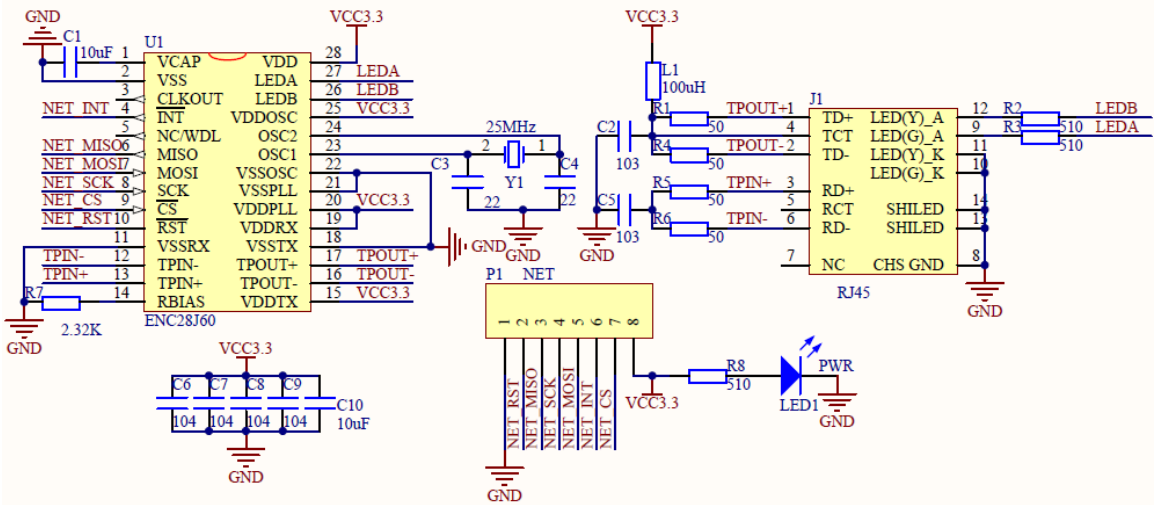

图 57.1.1.3 ALIENTEK ENC28J60 网络模块原理图

ALIENTEK ENC28J60 网络模块外观图如图 57.1.1.4 所示：

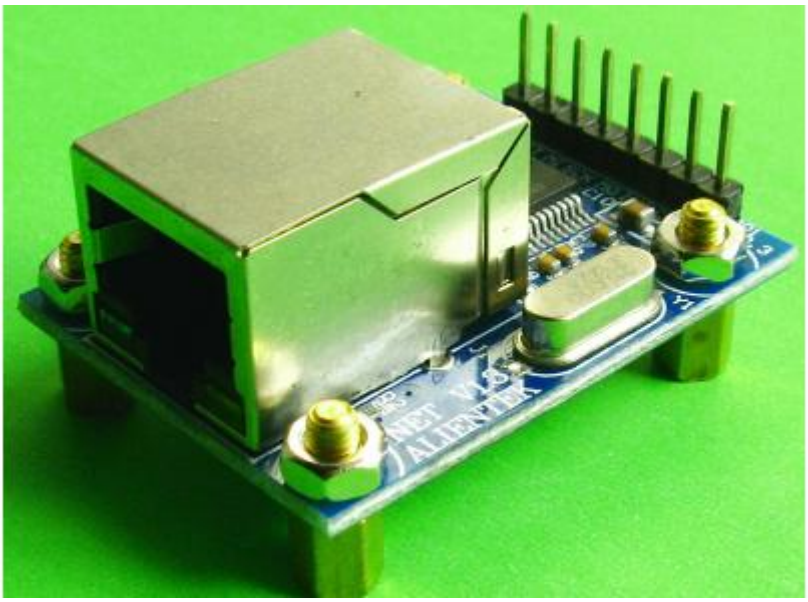

图 57.1.1.4 ALIENTEK ENC28J60 网络模块外观图

该模块通过一个 8 个引脚的排针与外部电路连接，这 8 个引脚分别是：GND、RST、MISO、SCK、MOSI、INT、CS 和 V3.3。其中 GND 和 V3.3 用于给模块供电，MISO/MOSI/SCK 用于 SPI 通信，CS 是片选信号，INT 为中断输出引脚，RST 为模块复位信号。

57.1.2 uIP 简介

uIP 由瑞典计算机科学学院(网络嵌入式系统小组)的 Adam Dunkels 开发。其源代码由 C 语言编写，并完全公开，uIP 的最新版本是 1.0 版本，本指南移植和使用的版本正是此版本。

uIP 协议栈去掉了完整的 TCP/IP 中不常用的功能，简化了通讯流程，但保留了网络通信必须使用的协议，设计重点放在了 IP/TCP/ICMP/UDP/ARP 这些网络层和传输层协议上，保证了其代码的通用性和结构的稳定性。

由于 uIP 协议栈专门为嵌入式系统而设计，因此还具有如下优越功能：

1） 代码非常少，其协议栈代码不到 6K，很方便阅读和移植。

2） 占用的内存数非常少，RAM 占用仅几百字节。

3） 其硬件处理层、协议栈层和应用层共用一个全局缓存区，不存在数据的拷贝，且发送和接收都是依靠这个缓存区，极大的节省空间和时间。

4） 支持多个主动连接和被动连接并发。

5） 其源代码中提供一套实例程序：web 服务器，web 客户端，电子邮件发送程序(SMTP 客户端)，Telnet 服务器， DNS 主机名解析程序等。通用性强，移植起来基本不用修改就可以通过。

6） 对数据的处理采用轮循机制，不需要操作系统的支持。

由于 uIP 对资源的需求少和移植容易，大部分的 8 位微控制器都使用过 uIP 协议栈，而且很多的著名的嵌入式产品和项目(如卫星，Cisco 路由器，无线传感器网络)中都在使用 uIP 协议栈。

uIP 相当于一个代码库，通过一系列的函数实现与底层硬件和高层应用程序的通讯，对于整个系统来说它内部的协议组是透明的，从而增加了协议的通用性。uIP 协议栈与系统底层和

高层应用之间的关系如图 57.1.2.1 所示：

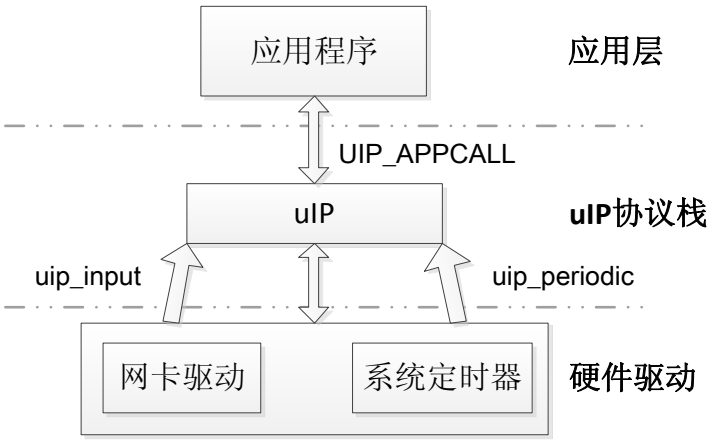

图 57.1.2.1 uIP 在系统中的位置

从上图可以看出，uIP 协议栈主要提供 2 个函数供系统底层调用：uip_input 和 uip_periodic。另外和应用程序联系主要是通过 UIP_APPCALL 函数。

当网卡驱动收到一个输入包时，将放入全局缓冲区 uip_buf 中，包的大小由全局变量 uip_len 约束。同时将调用 uip_input()函数，这个函数将会根据包首部的协议处理这个包和需要时调用应用程序。当 uip_input()返回时，一个输出包同样放在全局缓冲区 uip_buf 里，大小赋给 uip_len。如果 uip_len 是 0，则说明没有包要发送。否则调用底层系统的发包函数将包发送到网络上。

uIP 周期计时是用于驱动所有的 uIP 内部时钟事件。当周期计时激发，每一个 TCP 连接都会调用 uIP 函数 uip_periodic()。类似于 uip_input()函数。uip_periodic()函数返回时，输出的 IP 包要放到 uip_buf 中，供底层系统查询 uip_len 的大小发送。

由于使用 TCP/IP 的应用场景很多，因此应用程序作为单独的模块由用户实现。uIP 协议栈提供一系列接口函数供用户程序调用，其中大部分函数是作为 C 的宏命令实现的，主要是为了速度、代码大小、效率和堆栈的使用。用户需要将应用层入口程序作为接口提供给 uIP 协议栈， 并将这个函数定义为宏 UIP_APPCALL()。这样，uIP 在接受到底层传来的数据包后，在需要送到上层应用程序处理的地方，调用 UIP_APPCALL( )。在不用修改协议栈的情况下可以适配不同的应用程序。

uIP 协议栈提供了我们很多接口函数，这些函数在 uip.h 中定义，为了减少函数调用造成的额外支出，大部分接口函数以宏命令实现的，uIP 提供的接口函数有：

1，初始化 uIP 协议栈：uip_init()
2．处理输入包：uip_input()
3．处理周期计时事件：uip_periodic()
4．开始监听端口：uip_listen()
5．连接到远程主机：uip_connect()
6．接收到连接请求：uip_connected()
7．主动关闭连接：uip_close()
8．连接被关闭：uip_closed()
9．发出去的数据被应答：uip_acked()
10．在当前连接发送数据：uip_send()
11．在当前连接上收到新的数据：uip_newdata()
12．告诉对方要停止连接：uip_stop()
13．连接被意外终止：uip_aborted()

接下来，我们看看 uIP 的移植过程。首先，uIP1.0 的源码包里面有如下内容，如图 57.1.2.2 所示：

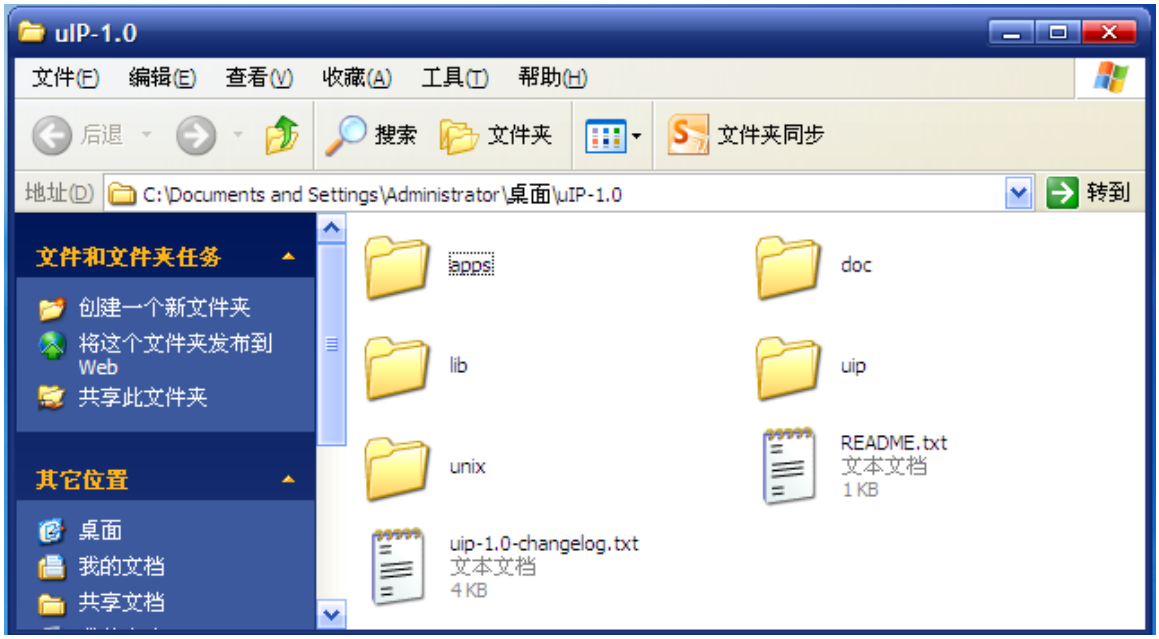

图 57.1.2.2 uIP 1.0 源码包内容

其中 apps 文件夹里面是 uip 提供的各种参考代码，本章我们主要有用到里面的 webserver 部分。doc 文件夹里面是一些 uip 的使用及说明文件，是学习 uip 的官方资料。lib 文件夹里面是用于内存管理的一个代码，本章我们没有用到。uip 里面就是 uip 1.0 的源码了，我们全盘照收。unix 里面提供的是具体的应用实例，我们移植参考主要是依照这个里面的代码。

移植第一步：实现在 unix/tapdev.c 里面的三个函数。首先是 tapdev_init 函数，该函数用于初始化网卡（也就是我们的 ENC28J60），通过这个函数实现网卡初始化。其次，是 tapdev_read 函数，该函数用于从网卡读取一包数据，将读到的数据存放在 uip_buf 里面，数据长度返回给 uip_len。最后，是 tapdev_send 函数，该函数用于向网卡发送一包数据，将全局缓存区 uip_buf 里面的数据发送出去（长度为 uip_len）。其实这三个函数就是实现最底层的网卡操作。

第二步，因为 uIP 协议栈需要使用时钟，为 TCP 和 ARP 的定时器服务，因此我们需要 STM32 提供一个定时器做时钟，提供 10ms 计时（假设 clock-arch.h 里面的 CLOCK_CONF_SECOND 为 100），通过 clock-arch.c 里面的 clock_time 函数返回给 uIP 使用。

第三步，配置 uip-conf.h 里面的宏定义选项。主要用于设置 TCP 最大连接数、TCP 监听端口数、CPU 大小端模式等，这个大家根据自己需要配置即可。

通过以上 3 步的修改，我们基本上就完成了 uIP 的移植。在使用 uIP 的时候，一般通过如下顺序：

1） 实现接口函数（回调函数）UIP_APPCALL。

该函数是我们使用 uIP 最关键的部分，它是 uIP 和应用程序的接口，我们必须根据自己的需要，在该函数做各种处理，而做这些处理的触发条件，就是前面提到的 uIP 提供的那些接口函数，如 uip_newdata、uip_acked、uip_closed 等等。另外，如果是 UDP，那么还需要实现 UIP_UDP_APPCALL 回调函数。

2） 调用 tapdev_init 函数，先初始化网卡。

此步先初始化网卡，配置 MAC 地址，为 uIP 和网络通信做好准备。

3） 调用 uip_init 函数，初始化 uIP 协议栈。

此步主要用于 uip 自身的初始化，我们直接调用就是。

4） 设置 IP 地址、网关以及掩码

这个和电脑上网差不多，只不过我们这里是通过 uip_ipaddr、uip_sethostaddr、uip_setdraddr 和 uip_setnetmask 等函数实现。

5） 设置监听端口

uIP 根据你设定的不同监听端口，实现不同的服务，比如我们实现 Web Server 就监听 80 端口(浏览器默认的端口是 80 端口)，凡是发现 80 端口的数据，都通过 Web Server 的 APPCALL 函数处理。根据自己的需要设置不同的监听端口。不过 uIP 有本地端口（lport）和远程端口（rport）之分，如果是做服务端，我们通过监听本地端口（lport）实现；如果是做客户端，则需要去连接远程端口（rport）。

6） 处理 uIP 事件

最后，uIP 通过 uip_polling 函数轮询处理 uIP 事件。该函数必须插入到用户的主循环里面（也就是必须每隔一定时间调用一次）。

57.2 硬件设计

本节实验功能简介：开机检测 ENC28J60，如果检测不成功，则提示报错。在成功检测到 ENC28J60 之后，初始化 uIP，并设置 IP 地址（192.168.1.16）等，然后监听 80 端口和 1200 端口，并尝试连接远程 1400 端口，80 端口用于实现 WEB Server 功能，1200 端口用于实现 TCP Server 功能，连接 1400 端口实现 TCP Client 功能。此时，我们在电脑浏览器输入 http://192.168.1.16 ，就可以登录到一个界面，该界面可以控制开发板上两个 LED 灯的亮灭，还会显示开发板的当前时间以及开发板 STM32 芯片的温度（每 10 秒自动刷新一次）。另外，我们通过网络调试软件（做 TCP Server 时，设置 IP 地址为：192.168.1.103，端口为 1400；做 TCP Client 时，设置 IP 地址为：192.168.1.16，端口为 1200）同开发板连接，即可实现开发板与网络调试软件之间的数据互发。按 KEY0，由开发板的 TCP Server 端发送数据到电脑的 TCP Client 端。按 KEY2，则由开发板的 TCP Client 端发送数据到电脑的 TCP Server 端。LCD 显示当前连接状态。

所要用到的硬件资源如下：

1） 指示灯 DS0 、DS1

2） KEY0/KEY2 两个按键

3） 串口

4） TFTLCD 模块

5） ENC28J60 网络模块

前面 4 部分都已经详细介绍过，本章，我们重点看看 ALIENTEK ENC28J60 网络模块同 ALIENTEK 战舰 STM32 开发板的连接，前面我们介绍了 ALIENTEK ENC28J60 网络模块的接口，我们通过杜邦线（或排线）连接网络模块和开发板的 P12 端口，连接关系如表 56.2.1 所示：

编号	1	2	3	4	5	6	7	8
网络模块(P1端子)	GND	NET_RST	NET_MISO	NET_SCK	NET_MOSI	NET_INT	NET_CS	VCC3.3
开发板（P12端子）	GND	PG6	PB14	PB13	PB15	PD2	PG8	VCC3.3

表 56.2.1 ENC28J60 网络模块同战舰 STM32 开发板连接关系表

上表可以看出，其实网络模块同战舰 STM32 开发板的线序是一一对应的，所以如果你有一个 1*8 的排线，就可以直接对插即可。这里需要注意，本来开发板的 P12 端口是用来连接 SD

卡，实现 SPI 读写 SD 卡的，如果要连接网络模块，我们需要把跳线帽连接到 P10 和 P11，这样还是可以通过 SDIO 访问 SD 卡。

在开发板连接网络模块以后，我们还需要一根网线（自备），连接网络模块和路由器，这样我们才能实现和电脑的连接。

57.3 软件设计

本章，我们在第二十八章实验 （实验 23 ）的基础上修改，在该工程源码下面加入 uIP-1.0 文件夹，存放 uIP1.0 源码，再新建 uIP-APP 文件夹，存放应用部分代码，因为 uIP 自己有一个 timer.c 和 timer.h 的文件，所以我们还需要修改 HARDWARE 里面的 timer.c 和 timer.h 为不同的名字，本章我们改为 timerx.c 和 timerx.h，我们还需要实现 ENC28J60 的驱动代码，存放在 HARDWARE 文件夹下的 ENC28J60 文件夹里面。详细的步骤我们就不一一阐述了，全部改好之后，工程如图 57.3.1 所示：

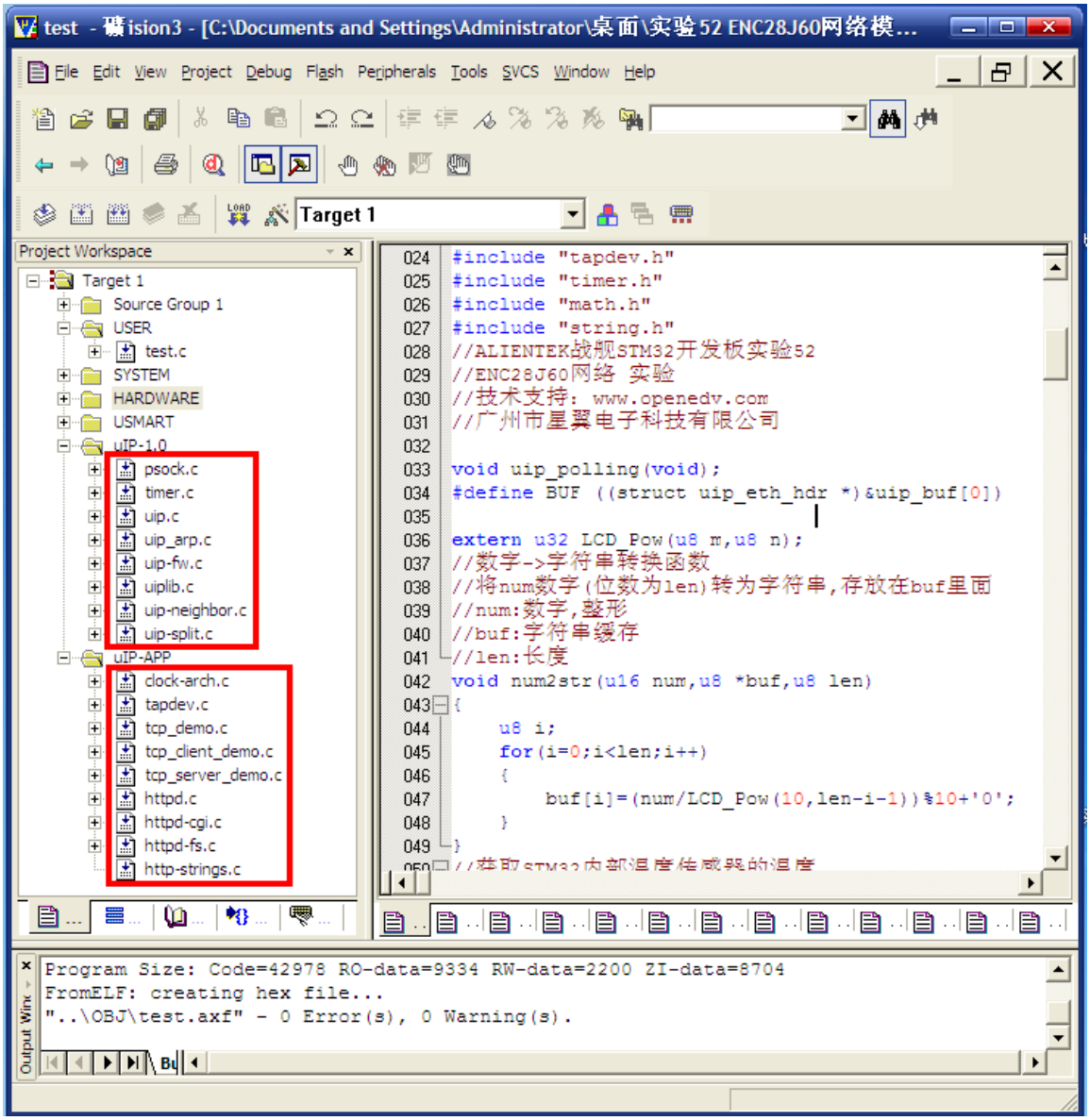

图 57.3.1 移植完后，MDK 工程图

图中 uIP-1.0 文件夹里面的代码全部是 uIP 提供的协议栈源码，而 uIP-APP 里面的代码则部分是我们自己实现的，部分是 uIP 提供的，其中：

clock-arch.c，属于 uIP 协议栈，uIP 通过该代码里面的 clock_time 函数获取时钟节拍。

tapdev.c，同样是 uIP 提供，用来实现 uIP 与网卡的接口，该文件实现 tapdev_init、tapdev_read 和 tapdev_send 三个重要函数。

tcp_demo.c，完成 UIP_APPCALL 函数的实现，即 tcp_demo_appcall 函数。该函数根据端口的不同，分别调用不同的 appcall 函数，实现不同功能。同时该文件还实现了 uip_log 函数，用于打印日志。

tcp_client_demo.c，完成一个简单的 TCP 客户端应用，实现与电脑 TCP 服务端的数据收发。

tcp_server_demo.c，完成一个简单的 TCP 服务端应用，实现与电脑 TCP 客户端的数据收发。

httpd.c、httpd-cgi.c、httpd-fs.c 和 httpd-strings.h，属于 uIP 提供的 WEB 服务器参考代码，我们通过修改部分代码，实现一个简单的 WEB 服务器。

本章代码很多，我们仅挑一些重点和大家介绍。

首先是 tapdev.c 里面的三个函数，代码如下：

```
//MAC 地址,必须唯一
//如果你有两个战舰开发板,想连入路由器,则需要修改 MAC 地址不一样!
const u8 mymac[6]={0x04,0x02,0x35,0x00,0x00,0x01};  //MAC 地址
//配置网卡硬件，并设置 MAC 地址
//返回值：0，正常；1，失败；
u8 tapdev_init(void)
{
    u8 i,res=0;
    res=ENC28J60_Init((u8*)mymac);      //初始化 ENC28J60
    //把 IP 地址和 MAC 地址写入缓存区
    for (i = 0; i < 6; i++)uip_ethaddr.addr[i]=mymac[i];
    //指示灯状态:0x476 is PHLCON LEDA(绿)=links status, LEDB(红)=receive/transmit
    //PHLCON：PHY 模块 LED 控制寄存器
    ENC28J60_PHY_Write(PHLCON,0x0476);
    return res;
}
//读取一包数据
uint16_t tapdev_read(void)
{
    return  ENC28J60_Packet_Receive(MAX_FRAMELEN,uip_buf);
}
//发送一包数据
void tapdev_send(void)
{
    ENC28J60_Packet_Send(uip_len,uip_buf);
}
```

tapdev_init 函数，该函数用于初始化网卡，即初始化我们的 ENC28J60，初始化工作主要通过调用 ENC28J60_Init 函数实现，该函数在 enc28j60.c 里面实现，同时该函数还用于设置 MAC 地址，这里请确保 MAC 地址的唯一性。在初始化 enc28j60 以后，我们设置 enc28j60 的 LED

控制器工作方式，即完成对 ENC28J60 的全部初始化工作。该函数的返回值用于判断网卡初始化是否成功。

tapdev_read 函数，该函数调用 ENC28J60_Packet_Receive 函数，实现从网卡（ENC28J60）读取一包数据，数据被存放在 uip_buf 里面，同时返回读到的包长度（包长度一般是存放在 uip_len 里面的）。

tapdev_send 函数，该函数调用 ENC28J60_Packet_Send 函数，实现从网卡（ENC28J60）发送一包数据到网络，数据内容存放在 uip_buf，数据长度为 uip_len。

再来看看 tcp_demo.c 里面的 tcp_demo_appcall 函数，该函数代码如下：

```
//TCP 应用接口函数(UIP_APPCALL)
//完成 TCP 服务(包括 server 和 client)和 HTTP 服务
void tcp_demo_appcall(void)
{
    switch(uip_conn->lport)//本地监听端口 80 和 1200
    {
        case HTONS(80):
            httpd_appcall();
            break;
        case HTONS(1200):
            tcp_server_demo_appcall();
            break;
        default: break;
    }
    switch(uip_conn->rport) //远程连接 1400 端口
    {
        case HTONS(1400):
            tcp_client_demo_appcall();
            break;
        default: break;
    }
}
```

该函数即 UIP_APPCALL 函数，是 uIP 同应用程序的接口函数，该函数通过端口号选择不同的 appcall 函数，实现不同的服务。其中 80 端口用于实现 WEB 服务，通过调用 httpd_appcall 实现；1200 端口用于实现 TCP 服务器，通过调用 tcp_server_demo_appcall 函数实现；1400 是远程端口，用于实现 TCP 客户端，调用 tcp_client_demo_appcall 函数实现。

接着，我们来看看这 3 个 appcall 函数，首先是 WEB 服务器的 appcall 函数：httpd_appcall，该函数在 httpd.c 里面实现，源码如下：

```
//http 服务（WEB）处理
void httpd_appcall(void)
{
    struct httpd_state *s = (struct httpd_state *)&(uip_conn->appstate);//读取连接状态
    if(uip_closed() || uip_aborted() || uip_timedout())//异常处理（这里无任何处理）
    else if(uip_connected())//连接成功
    {
```

```
            PSOCK_INIT(&s->sin, s->inputbuf, sizeof(s->inputbuf) - 1);
            PSOCK_INIT(&s->sout, s->inputbuf, sizeof(s->inputbuf) - 1);
            PT_INIT(&s->outputpt);
            s->state = STATE_WAITING;
            /*      timer_set(&s->timer, CLOCK_SECOND * 100);*/
            s->timer = 0;
            handle_connection(s);//处理
        }else if(s!=NULL)
        {
            if(uip_poll())
            {
                ++s->timer;
                if(s->timer >= 20)uip_abort();
                else s->timer = 0;
            }
            handle_connection(s);
        }else uip_abort();//
    }
```

该函数在连接建立的时候，通 handle_connection 函数处理 http 数据，handle_connection 函数代码如下：

```
    //分析 http 数据
    static void handle_connection(struct httpd_state *s)
    {
        handle_input(s);   //处理 http 输入数据
        if(s->state==STATE_OUTPUT)handle_output(s);//输出状态，处理输出数据
    }
```

该函数调用 handle_input 处理 http 输入数据，通过调用 handle_output 实现 http 网页输出。对我们来说最重要的是 handle_input 函数，handle_input 函数代码如下：

```
    extern unsigned char data_index_html[];//在 httpd-fsdata.c 里面定义,用于存放 html 网页源代码
    extern void get_temperature(u8 *temp);//在 main 函数实现,用于获取温度字符串
    extern void get_time(u8 *time);          //在 main 函数实现,用于获取时间字符串
    const u8 * LED0_ON_PIC_ADDR="http://www.openedv.com/upload/2012/9/27/ad65ee9f478ca
    11241933beed5b5dbcc_971.gif";          //LED0 亮,图标地址
    const u8 * LED1_ON_PIC_ADDR="http://www.openedv.com/upload/2012/9/27/bab5bef0379dc
    50129202157c2739c57_775.gif";          //LED1 亮,图标地址
    const u8 * LED_OFF_PIC_ADDR="http://www.openedv.com/upload/2012/9/27/ccecf4ebeb84b
    095545b8feb0cecc671_254.gif";          //LED 灭,图标地址
    //处理 HTTP 输入数据
    static PT_THREAD(handle_input(struct httpd_state *s))
    {
        char *strx;
        u8 dbuf[17];
        PSOCK_BEGIN(&s->sin);
```

```
PSOCK_READTO(&s->sin, ISO_space);
if(strncmp(s->inputbuf, http_get, 4)!=0)PSOCK_CLOSE_EXIT(&s->sin);//比较客户端
//浏览器输入的指令是否是申请 WEB 指令 “GET ”
PSOCK_READTO(&s->sin, ISO_space);                                    //" "
if(s->inputbuf[0] != ISO_slash)PSOCK_CLOSE_EXIT(&s->sin);     //判断第一个数据
                                                          //(去掉 IP 地址之后),是否是"/"
if(s->inputbuf[1] == ISO_space||s->inputbuf[1] == '?')       //第二个数据是空格/问号
{
    if(s->inputbuf[1]=='?'&&s->inputbuf[6]==0x31)//LED1
    {
        LED0=!LED0;
        strx=strstr((const char*)(data_index_html+13),"LED0 状态");
        if(strx)//存在"LED0 状态"这个字符串
        {
            strx=strstr((const char*)strx,"color:#");//找到"color:#"字符串
            if(LED0)//LED0 灭
            {
                strncpy(strx+7,"5B5B5B",6); //灰色
                strncpy(strx+24,"灭",2);        //灭
                strx=strstr((const char*)strx,"http:");//找到"http:"字符串
                strncpy(strx,(const char*)LED_OFF_PIC_ADDR,strlen((const char*)
                LED_OFF_PIC_ADDR));//LED0 灭图片
            }else
            {
                strncpy(strx+7,"FF0000",6);  //红色
                strncpy(strx+24,"亮",2);        //"亮"
                strx=strstr((const char*)strx,"http:");//找到"http:"字符串
                strncpy(strx,(const char*)LED0_ON_PIC_ADDR,strlen((const char*)
                LED0_ON_PIC_ADDR));//LED0 亮图片
            }
        }
    }else if(s->inputbuf[1]=='?'&&s->inputbuf[6]==0x32)//LED2
    {
        LED1=!LED1;
        strx=strstr((const char*)(data_index_html+13),"LED1 状态");
        if(strx)//存在"LED1 状态"这个字符串
        {
            strx=strstr((const char*)strx,"color:#");//找到"color:#"字符串
            if(LED1)//LED1 灭
            {
                strncpy(strx+7,"5B5B5B",6); //灰色
                strncpy(strx+24,"灭",2); //灭
                strx=strstr((const char*)strx,"http:");//找到"http:"字符串
```

```
                    strncpy(strx,(const char*)LED_OFF_PIC_ADDR,strlen((const char*)
                    LED_OFF_PIC_ADDR));//LED1 灭图片
                }else
                {
                    strncpy(strx+7,"00FF00",6);  //绿色
                    strncpy(strx+24,"亮",2); //"亮"
                    strx=strstr((const char*)strx,"http:");//找到"http:"字符串
                    strncpy(strx,(const char*)LED1_ON_PIC_ADDR,strlen((const char*)
                    LED1_ON_PIC_ADDR));//LED1 亮图片
                }
            }
        }
        strx=strstr((const char*)(data_index_html+13),"℃");//找到"℃"字符
        if(strx)
        {
            get_temperature(dbuf);                  //得到温度
            strncpy(strx-4,(const char*)dbuf,4);    //更新温度
        }
        strx=strstr((const char*)strx,"RTC 时间:");  //找到"RTC 时间:"字符
        if(strx)
        {
            get_time(dbuf);                         //得到时间
            strncpy(strx+33,(const char*)dbuf,16); //更新时间
        }
        strncpy(s->filename, http_index_html, sizeof(s->filename));
    }else //如果不是'/'?'
    {
        s->inputbuf[PSOCK_DATALEN(&s->sin)-1] = 0;
        strncpy(s->filename,&s->inputbuf[0],sizeof(s->filename));
    }
    s->state = STATE_OUTPUT;
    while(1)
    {
        PSOCK_READTO(&s->sin, ISO_nl);
        if(strncmp(s->inputbuf, http_referer, 8) == 0)
        {
            s->inputbuf[PSOCK_DATALEN(&s->sin) - 2] = 0;
        }
    }
    PSOCK_END(&s->sin);
}
```

这里，我们需要了解 uIP 是把网页数据（源文件）存放在 data_index_html，通过将这里面的数据发送给电脑浏览器，浏览器就会显示出我们所设计的界面了。当用户在网页上面操作的

时候，浏览器就会发送消息给 WEB 服务器，服务器根据收到的消息内容，判断用户所执行的操作，然后发送新的页面到浏览器，这样用户就可以看到操作结果了。本章，我们实现的 WEB 服界面如图 57.3.2 所示：

图 57.3.2 WEB 服务器界面

图中两个按键分别控制 DS0 和 DS1 的亮灭，然后还显示了 STM32 芯片的温度和 RTC 时间等信息。

控制 DS0，DS1 亮灭我们是通过发送不同的页面请求来实现的，这里我们采用的是 Get 方法（科普找百度），将请求参数放到 URL 里面，然后 WEB 服务器根据 URL 的参数来相应内容，这样实际上 STM32 就是从 URL 获取控制参数，以控制 DS0 和 DS1 的亮灭。uIP 在得到 Get 请求后判断 URL 内容，然后做出相应控制，最后修改 data_index_html 里面的部分内容（比如指示灯图标的变化，以及提示文字的变化等），再将 data_index_html 发送给浏览器，显示新的界面。

显示 STM32 温度和 RTC 时间是通过刷新实现的，uIP 每次得到来自浏览器的请求就会更新 data_index_html 里面的温度和时间等信息，然后将 data_index_html 发送给浏览器，这样达到更新温度和时间的目的。但是这样我们需要手动刷新，比较笨，所以我们在网页源码里面加入了自动刷新的控制代码，每 10 秒钟刷新一次，这样就不需要手动刷新了。

handle_input 函数实现了我们所说的这一切功能，另外请注意 data_index_html 是存放在 httpd-fsdata.c（该文件通过 include 的方式包含进工程里面）里面的一个数组，并且由于该数组的内容需要不停的刷新，所以我们定义它为 sram 数据，data_index_html 里面的数据，则是通过一个工具软件：amo 的编程小工具集合 V1.2.6.exe，将网页源码转换而来，该软件在光盘有提供，如果想自己做网页的朋友，可以通过该软件转换。

WEB 服务器就为大家介绍这么多。

接下来看看 TCP 服务器 appcall 函数：tcp_server_demo_appcall，该函数在 tcp_server_demo.c 里面实现，该函数代码如下：

```
    u8 tcp_server_databuf[200];        //发送数据缓存
    u8 tcp_server_sta;                 //服务端状态
    //[7]:0,无连接;1,已经连接;
    //[6]:0,无数据;1,收到客户端数据
    //[5]:0,无数据;1,有数据需要发送
    //这是一个 TCP 服务器应用回调函数。
    //该函数通过 UIP_APPCALL(tcp_demo_appcall)调用,实现 Web Server 的功能.
//当 uip 事件发生时，UIP_APPCALL 函数会被调用,根据所属端口(1200),确定是否执行该函数。
//例如 ：当一个 TCP 连接被创建时、有新的数据到达、数据已经被应答、数据需要重发等事件
    void tcp_server_demo_appcall(void)
    {
        struct tcp_demo_appstate *s = (struct tcp_demo_appstate *)&uip_conn->appstate;
        if(uip_aborted())tcp_server_aborted();        //连接终止
        if(uip_timedout())tcp_server_timedout();      //连接超时
        if(uip_closed())tcp_server_closed();          //连接关闭
        if(uip_connected())tcp_server_connected();    //连接成功
        if(uip_acked())tcp_server_acked();            //发送的数据成功送达
        //接收到一个新的 TCP 数据包
        if (uip_newdata())//收到客户端发过来的数据
        {
            if((tcp_server_sta&(1<<6))==0)//还未收到数据
            {
                if(uip_len>199) ((u8*)uip_appdata)[199]=0;
                strcpy((char*)tcp_server_databuf,uip_appdata);
                tcp_server_sta|=1<<6;//表示收到客户端数据
            }
        }else if(tcp_server_sta&(1<<5))//有数据需要发送
        {
            s->textptr=tcp_server_databuf;
            s->textlen=strlen((const char*)tcp_server_databuf);
            tcp_server_sta&=~(1<<5);//清除标记
        }
        //当需要重发、新数据到达、数据包送达、连接建立时，通知 uip 发送数据
        if(uip_rexmit()||uip_newdata()||uip_acked()||uip_connected()||uip_poll())
        {
            tcp_server_senddata();
        }
    }
```

该函数通过 uip_newdata()判断是否接收到客户端发来的数据，如果是，则将数据拷贝到 tcp_server_databuf 缓存区，并标记收到客户端数据。当有数据要发送（KEY0 按下）的时候，将需要发送的数据通过 tcp_server_senddata 函数发送出去。

最后，我们看看 TCP 客户端 appcall 函数：tcp_client_demo_appcall，该函数代码同 TCP 服务端代码十分相似，该函数在 tcp_server_demo.c 里面实现，代码如下：

```
    u8 tcp_client_databuf[200];         //发送数据缓存
    u8 tcp_client_sta;                  //客户端状态
    //[7]:0,无连接;1,已经连接;
    //[6]:0,无数据;1,收到客户端数据
    //[5]:0,无数据;1,有数据需要发送
    //这是一个 TCP  客户端应用回调函数。
    //该函数通过 UIP_APPCALL(tcp_demo_appcall)调用,实现 Web Client 的功能.
//当 uip 事件发生时，UIP_APPCALL 函数会被调用,根据所属端口(1400),确定是否执行该函数。
//例如 ：当一个 TCP 连接被创建时、有新的数据到达、数据已经被应答、数据需要重发等事件
    void tcp_client_demo_appcall(void)
    {
        struct tcp_demo_appstate *s = (struct tcp_demo_appstate *)&uip_conn->appstate;
        if(uip_aborted())tcp_client_aborted();          //连接终止
        if(uip_timedout())tcp_client_timedout();        //连接超时
        if(uip_closed())tcp_client_closed();            //连接关闭
        if(uip_connected())tcp_client_connected();      //连接成功
        if(uip_acked())tcp_client_acked();              //发送的数据成功送达
        //接收到一个新的 TCP 数据包
        if (uip_newdata())
        {
            if((tcp_client_sta&(1<<6))==0)//还未收到数据
            {
                if(uip_len>199) ((u8*)uip_appdata)[199]=0;
                strcpy((char*)tcp_client_databuf,uip_appdata);
                tcp_client_sta|=1<<6;//表示收到客户端数据
            }
        }else if(tcp_client_sta&(1<<5))//有数据需要发送
        {
            s->textptr=tcp_client_databuf;
            s->textlen=strlen((const char*)tcp_client_databuf);
            tcp_client_sta&=~(1<<5);//清除标记
        }
        //当需要重发、新数据到达、数据包送达、连接建立时，通知 uip 发送数据
        if(uip_rexmit()||uip_newdata()||uip_acked()||uip_connected()||uip_poll())
        {
            tcp_client_senddata();
        }
    }
```

该函数也是通过 uip_newdata()判断是否接收到服务端发来的数据，如果是，则将数据拷贝到 tcp_client_databuf 缓存区，并标记收到服务端数据。当有数据要发送（KEY2 按下）的时候，将需要发送的数据通过 tcp_client_senddata 函数发送出去。

uIP 通过 clock-arch 里面的 clock_time 获取时间节拍，我们通过在 timerx.c 里面初始化定时器 6，用于提供 clock_time 时钟节拍，每 10ms 加 1，这里代码就不贴出来了，请大家查看光盘

源码。

最后在 test.c 里面，我们要实现好几个函数，但是这里仅贴出 main 函数以及 uip_polling 函数，该部分如下：

```
#define BUF ((struct uip_eth_hdr *)&uip_buf[0])
int main(void)
{
    u8 key;
    u8 tcnt=0;
    u8 tcp_server_tsta=0XFF;
    u8 tcp_client_tsta=0XFF;
    uip_ipaddr_t ipaddr;
    Stm32_Clock_Init(9);          //系统时钟设置
    uart_init(72,9600);           //串口初始化为 9600
    delay_init(72);               //延时初始化
    LED_Init();                   //初始化与 LED 连接的硬件接口
    LCD_Init();                   //初始化 LCD
    KEY_Init();                   //初始化按键
    RTC_Init();                   //初始化 RTC
    Adc_Init();                   //初始化 ADC
    usmart_dev.init(72);          //初始化 USMART
    POINT_COLOR=RED;              //设置为红色
    LCD_ShowString(60,10,200,16,16,"WarShip STM32");
    LCD_ShowString(60,30,200,16,16,"ENC28J60 TEST");
    LCD_ShowString(60,50,200,16,16,"ATOM@ALIENTEK");
    while(tapdev_init())      //初始化 ENC28J60 错误
    {
        LCD_ShowString(60,70,200,16,16,"ENC28J60 Init Error!"); delay_ms(200);
        LCD_Fill(60,70,240,86,WHITE);//清除之前显示
    };
    uip_init();                   //uIP 初始化
    LCD_ShowString(60,70,200,16,16,"KEY0:Server Send Msg");
    LCD_ShowString(60,90,200,16,16,"KEY2:Client Send Msg");
    LCD_ShowString(60,110,200,16,16,"IP:192.168.1.16");
    LCD_ShowString(60,130,200,16,16,"MASK:255.255.255.0");
    LCD_ShowString(60,150,200,16,16,"GATEWAY:192.168.1.1");
    LCD_ShowString(30,200,200,16,16,"TCP RX:");
    LCD_ShowString(30,220,200,16,16,"TCP TX:");
    LCD_ShowString(30,270,200,16,16,"TCP RX:");
    LCD_ShowString(30,290,200,16,16,"TCP TX:");
    POINT_COLOR=BLUE;
    uip_ipaddr(ipaddr, 192,168,1,16); //设置本地设置 IP 地址
    uip_sethostaddr(ipaddr);
    uip_ipaddr(ipaddr, 192,168,1,1);  //设置网关 IP 地址(其实就是你路由器的 IP 地址)
```

```
uip_setdraddr(ipaddr);
uip_ipaddr(ipaddr, 255,255,255,0);//设置网络掩码
uip_setnetmask(ipaddr);
uip_listen(HTONS(1200));                //监听 1200 端口,用于 TCP Server
uip_listen(HTONS(80));                  //监听 80 端口,用于 Web Server
tcp_client_reconnect();                 //尝试连接到 TCP Server 端,用于 TCP Client
while (1)
{
    uip_polling(); //处理 uip 事件，必须插入到用户程序的循环体中
    key=KEY_Scan(0);
    if(tcp_server_tsta!=tcp_server_sta)//TCP Server 状态改变
    {
        if(tcp_server_sta&(1<<7))LCD_ShowString(30,180,200,16,16,"TCP Server
        Connected    ");
        else LCD_ShowString(30,180,200,16,16,"TCP Server Disconnected");
        if(tcp_server_sta&(1<<6))   //收到新数据
        {
            LCD_Fill(86,200,240,216,WHITE);   //清除之前显示
            LCD_ShowString(86,200,154,16,16,tcp_server_databuf);
            printf("TCP Server RX:%s\r\n",tcp_server_databuf);//打印数据
            tcp_server_sta&=~(1<<6);          //标记数据已经被处理
        }
        tcp_server_tsta=tcp_server_sta;
    }
    if(key==KEY_RIGHT)//TCP Server  请求发送数据
    {
        if(tcp_server_sta&(1<<7))   //连接还存在
        {
            sprintf((char*)tcp_server_databuf,"TCP Server OK %d\r\n",tcnt);
            LCD_Fill(86,220,240,236,WHITE);                         //清除之前显示
            LCD_ShowString(86,220,154,16,16,tcp_server_databuf);   //显示发送数据
            tcp_server_sta|=1<<5;//标记有数据需要发送
            tcnt++;
        }
    }
    if(tcp_client_tsta!=tcp_client_sta)//TCP Client 状态改变
    {
        if(tcp_client_sta&(1<<7))LCD_ShowString(30,250,200,16,16,"TCP Client
        Connected    ");
        else LCD_ShowString(30,250,200,16,16,"TCP Client Disconnected");
        if(tcp_client_sta&(1<<6))   //收到新数据
        {
            LCD_Fill(86,270,240,286,WHITE);   //清除之前显示
```

```
                LCD_ShowString(86,270,154,16,16,tcp_client_databuf);
                printf("TCP Client RX:%s\r\n",tcp_client_databuf);//打印数据
                tcp_client_sta&=~(1<<6);          //标记数据已经被处理
            }
            tcp_client_tsta=tcp_client_sta;
        }
        if(key==KEY_LEFT)//TCP Client 请求发送数据
        {
            if(tcp_client_sta&(1<<7))     //连接还存在
            {
                sprintf((char*)tcp_client_databuf,"TCP Client OK %d\r\n",tcnt);
                LCD_Fill(86,290,240,306,WHITE);          //清除之前显示
                LCD_ShowString(86,290,154,16,16,tcp_client_databuf);//显示发送数据
                tcp_client_sta|=1<<5;//标记有数据需要发送
                tcnt++;
            }
        }
        delay_ms(1);
    }
}
//uip 事件处理函数
//必须将该函数插入用户主循环,循环调用.
void uip_polling(void)
{
    u8 i;
    static struct timer periodic_timer, arp_timer;
    static u8 timer_ok=0;
    if(timer_ok==0)//仅初始化一次
    {
        timer_ok = 1;
        timer_set(&periodic_timer,CLOCK_SECOND/2); //创建 1 个 0.5 秒的定时器
        timer_set(&arp_timer,CLOCK_SECOND*10);     //创建 1 个 10 秒的定时器
    }
    uip_len=tapdev_read();  //从网络读取一个 IP 包,得到数据长度.uip_len 在 uip.c 中定义
    if(uip_len>0)               //有数据
    {
        //处理 IP 数据包(只有校验通过的 IP 包才会被接收)
        if(BUF->type == htons(UIP_ETHTYPE_IP))//是否是 IP 包?
        {
            uip_arp_ipin();      //去除以太网头结构，更新 ARP 表
            uip_input();         //IP 包处理
            //当上面的函数执行后，如果需要发送数据，则全局变量 uip_len > 0
            //需要发送的数据在 uip_buf, 长度是 uip_len   (这是 2 个全局变量)
```

```
            if(uip_len>0)//需要回应数据
            {
                uip_arp_out();//加以太网头结构，在主动连接时可能要构造 ARP 请求
                tapdev_send();//发送数据到以太网
            }
        }else if (BUF->type==htons(UIP_ETHTYPE_ARP))//处理 arp 报文,是否是 ARP 包?
        {
            uip_arp_arpin();
            //当上面的函数执行后，如果需要发送数据，则全局变量 uip_len>0
            //需要发送的数据在 uip_buf, 长度是 uip_len(这是 2 个全局变量)
            if(uip_len>0)tapdev_send();//需要发送数据,则通过 tapdev_send 发送
        }
    }
    else if(timer_expired(&periodic_timer))     //0.5 秒定时器超时
    {
        timer_reset(&periodic_timer);           //复位 0.5 秒定时器
        //轮流处理每个 TCP 连接, UIP_CONNS 缺省是 40 个
        for(i=0;i<UIP_CONNS;i++)
        {
            uip_periodic(i);    //处理 TCP 通信事件
            //当上面的函数执行后，如果需要发送数据，则全局变量 uip_len>0
            //需要发送的数据在 uip_buf, 长度是 uip_len (这是 2 个全局变量)
            if(uip_len>0)
            {
                uip_arp_out();//加以太网头结构，在主动连接时可能要构造 ARP 请求
                tapdev_send();//发送数据到以太网
            }
        }
#if UIP_UDP  //UIP_UDP
        //轮流处理每个 UDP 连接, UIP_UDP_CONNS 缺省是 10 个
        for(i=0;i<UIP_UDP_CONNS;i++)
        {
            uip_udp_periodic(i);    //处理 UDP 通信事件
            //当上面的函数执行后，如果需要发送数据，则全局变量 uip_len>0
            //需要发送的数据在 uip_buf, 长度是 uip_len (这是 2 个全局变量)
            if(uip_len > 0)
            {
                uip_arp_out();//加以太网头结构，在主动连接时可能要构造 ARP 请求
                tapdev_send();//发送数据到以太网
            }
        }
#endif
        //每隔 10 秒调用 1 次 ARP 定时器函数 用于定期 ARP 处理,ARP 表 10 秒更新一次，
```

```
            //旧的条目会被抛弃
            if(timer_expired(&arp_timer))
            {
                timer_reset(&arp_timer);
                uip_arp_timer();
            }
        }
    }
```

其中 main 函数相对比较简单，先初始化网卡（ENC28J60）和 uIP 等，然后设置 IP 地址（192.168.1.16）及监听端口（1200 和 80），就开始轮询 uip_polling 函数，实现 uIP 事件处理，同时扫描按键，实现数据发送处理。当有收到数据的时候，将其显示在 LCD 上，同时通过串口发送到电脑。注意，这里 main 函数调用的 tcp_client_reconnect 函数，用于本地（STM32）TCP Client 去连接外部服务端，该函数设置服务端 IP 地址为 192.168.1.103（就是你电脑的 IP 地址），连接端口为 1400，只要没有连上，该函数就会不停的尝试连接。

uip_polling 函数，第一次调用的时候创建两个定时器，当收到包的时候（uip_len>0），先区分是 IP 包还是 ARP 包，针对不同的包做不同处理，对我们来说主要是通过 uip_input 处理 IP 包，实现数据处理。当没有收到包的时候（uip_len=0），通过定时器定时处理各个 TCP/UDP 连接以及 ARP 表处理。

软件设计部分就为大家介绍到这里。

57.4 下载验证

在代码编译成功之后，我们通过下载代码到战舰 STM32 开发板上（假设网络模块已经连接上开发板），LCD 显示如图 57.4.1 所示界面：

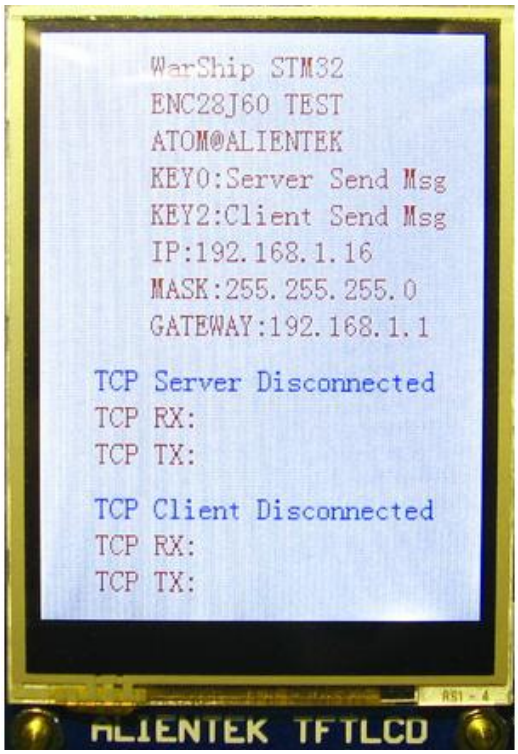

图 57.4.1 初始界面

可以看到，此时 TCP Server 和 TCP Client 都是没有连接的，我们打开：网络调试助手 V3.7.exe 这个软件（该软件在光盘有提供），然后选择 TCP Server，设置本地 IP 地址为：192.168.1.103（默认就是），设置本地端口为 1400，点击连接按钮，就会收到开发板发过来的消息，此时我们按开发板的 KEY2，就会发送数据给网络调试助手，同时也可以通过网络调试助手发送数据到 STM32 开发板。如图 57.4.2 所示：

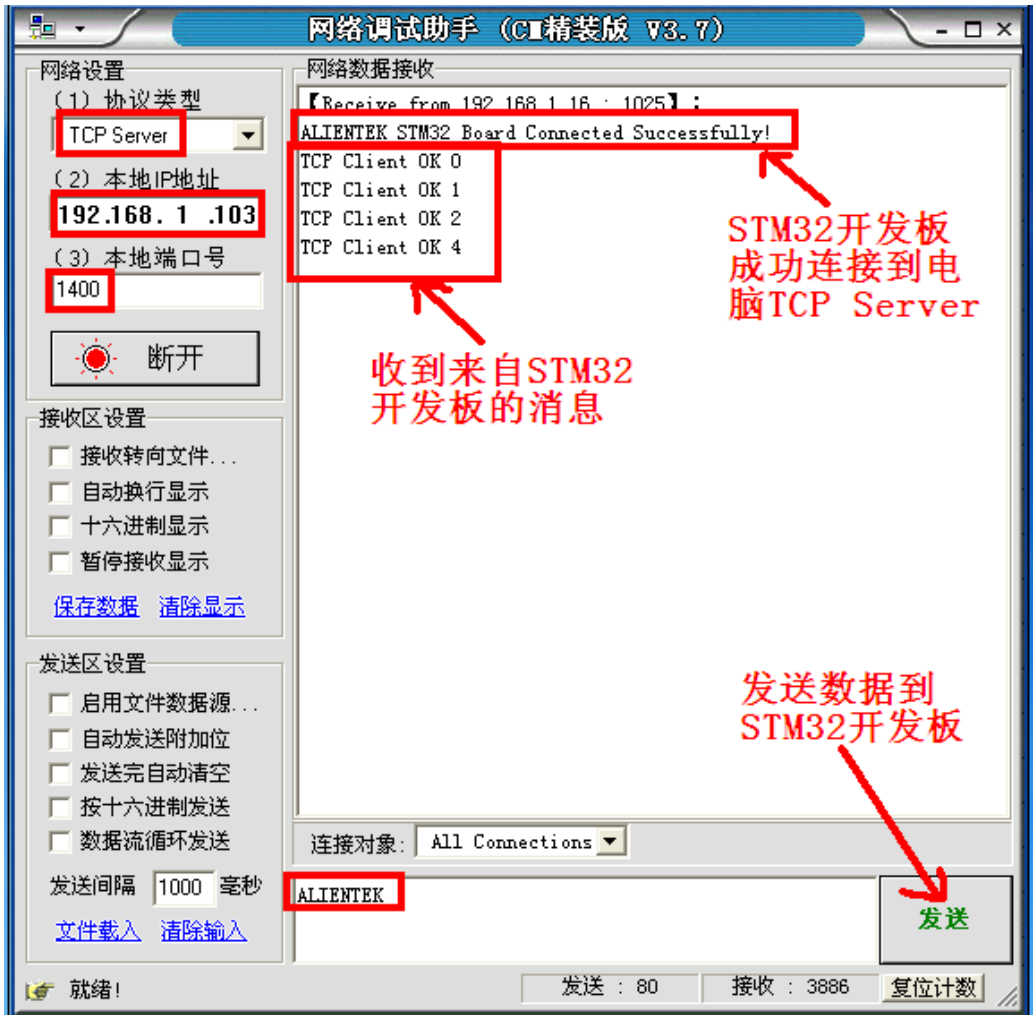

图 57.4.2 STM32 TCP Client 测试

在连接成功建立的时候，会在战舰 STM32 开发板上面显示 TCP Client 的连接状态，然后如果收到来自电脑 TCP Server 端的数据，也会在 LCD 上面显示，并打印到串口。这是我们实现的 TCP Client 功能。

如果我们在网络调试助手，选择协议类型为 TCP Client，然后设置服务器 IP 地址为 192.168.1.16（就是我们 STM32 开发板设置的 IP 地址），然后设置服务器端口为 1200，点击连接，同样可以收到开发板发过来的消息，此时我们按开发板的 KEY0 按键，就可以发送数据到网络调试助手，同时网络调试助手也可以发送数据到我们的开发板。如图 57.4.3 所示：

图 57.4.3 STM32 TCP Server 测试

在连接成功建立的时候，会在战舰 STM32 开发板上面显示 TCP Server 的连接状态，然后如果收到来自电脑 TCP Client 端的数据，便会在 LCD 上面显示，并打印到串口。这是我们实现的 TCP Server 功能。

最后，我们测试 WEB 服务器功能。打开浏览器，输入 http://192.168.1.16 ，就可以看到如下界面，如图 57.4.4 所示：

图 57.4.4 STM32 WEB Server 测试

此时，我们点击网页上的 DS0 状态反转和 DS1 状态反转按钮，就可以控制 DS0 和 DS1 的亮灭了。同时在该界面还显示了 STM32 的温度和 RTC 时间，每次刷新的时候，进行数据更新，另外浏览器每 10 秒钟会自动刷新一次，以更新时间和温度信息。