第一章　基于单片机的网络编程概述

随着网络技术的迅猛发展，Internet 已经走进千家万户，越来越多的人拥有了随时随地上网的条件，享受着网络带来的方便快捷的生活。同时，随着嵌入式控制技术的成熟，网络也逐步与之结合，深入到工业、楼宇、家居智能化等领域，实现远程数据采集、远程控制等功能。网络化已经成为新一代嵌入式系统发展的一个重要趋势。试想不久的将来，坐在办公室的电脑前就能查看和控制家里的门窗和灯的状态，甚至可以在下班时把家里配好汤料的电饭煲打开，到家就能闻到扑鼻而来的香味了。

盛行全球的 Internet 网络是基于 TCP/IP 协议族为基础组建的，TCP/IP 是网络通讯系统互联的事实标准。研究嵌入式系统的网络化，就要先从 TCP/IP 的概念入手。

1．1　TCP/IP 的概念及分层结构

TCP/IP 协议是传输控制协议的简称，它实际上是一个协议族，包括许多相关协议。其中最核心的协议是 IP（网际协议）和 TCP（传输控制协议），其它还包括 ARP（地址解析协议）、RARP（逆地址解析协议）、ICMP（Internet 控制报文协议）、UDP（用户数据报协议）、IGMP（Internet 组管理协议）、DNS（域名系统）、TFTP（简单文件传送协议）、BOOTP（引导程序协议）、SNMP（简单网络管理协议）、Telnet（远程控制协议）、FTP（文件传送协议）、SMTP（简单邮件传送协议）等重要协议。并且，随着网络技术的发展，还会不断有新的协议加入到 TCP/IP 协议族。这些协议规范了不同的场景下的网络互连，实际应用中可以根据系统的需要使用其中的一些协议。

从 TCP/IP 协议的数量就可以看出，Internet 网络是一个比较复杂的系统，能适配多种应用场景，根据使用协议的不同而实现不同的功能。为了降低网络设计的复杂性，设计者将以分层的方式组织 TCP/IP 协议，每一层可能包括不同通信服务的多种协议。从最底层的硬件开始，每一层都建立在其下一层的基础上，并负责向其上一层提供服务。只有相邻层才能通过软件接口联系起来，非相邻层没有直接的联系。

TCP/IP 的分层见图 1－1，分为物理链路层、网络层、传输层和应用层四层。

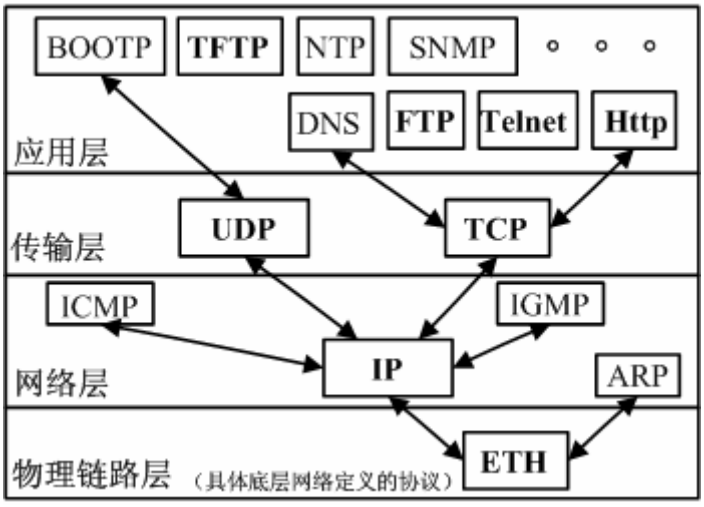

图 1－1 TCP/IP 协议族的分层结构

物理链路层：包含网络芯片的硬件和基于硬件上的芯片级驱动。随着网络物理介质的不同和使用网络芯片的不同，需要选择不同的通信方式和修改相应的驱动程序，但只要对外提供的接口不变，网络层的程序是不用修改的。例如以太网（Ethernet）和通过 Modem 上网的方式不同，驱动不同，但不影响网络层。

网络层：接收物理链路层过滤后的数据，并对通过识别不同的分组信息后传给传输层中不同的协议。著名的 IP(网际协议)是网络层的协议，它支持将多种网络技术互联为一个逻辑网络。IP 提供不可靠的、无连接的、尽最大努力交付的分组传输机制，为两个物理设备之间的信息传递提供最好的传输服务。所有具有网络层的因特网设备都会运行 IP 协议。

传输层：传输层的两个重要协议 TCP（传输控制协议）和 UDP（用户数据报），都是端到端的协议，根据应用程序需要的服务的不同可以选择其中的一个协议。发送时，TCP 和 UDP 都将报文头和数据打包放在 IP 的数据段中发送出去。接收 IP 分组后，剥离 IP 的首部，得出是 TCP 还是 UTP 协议，再根据其首部中端口的不同，交给应用层程序处理。

应用层：这一层的功能最终面向用户，因此非常丰富，并且千差万别。每一个应用层协议都是为了解决某一类应用问题而规定的，是通信双方都需要遵循该协议才能正常通讯。比如，telnet 提供远程登陆服务；FTP 提供应用级的文件传输服务；SMTP 提供简单的电子邮件发送服务；http 提供网页浏览服务；还有域名服务系统 DNS、简单网络管理协议 SNMP 等等。

1．2　TCP/IP 网络的通信流程

前面说过，TCP/IP 分为四层。那么为什么要这样设计，分层之间怎样通信？我们从生活中的信件传递说起。

如果你要告诉对方一些事情，写到信纸上后，需要装到写有对方邮编、地址和收件人的信封里，然后给邮递员。邮递员在邮局里需要将同一城市的信件和物品打包成一个编织袋，袋子上写着始发城市和目的城市，如深圳邮局到北京邮局。然后这个编织袋会经过飞机、火车或汽车送往北京邮局。以上过程都属于“封装”的过程。

编织袋到目的城市后，对方邮递员拆开编织袋，将信封送到接收人手中。对方拆开信件后才能读出真正的内容。这个过程属于“解包”的过程。

虽然发件人要说的事情不在信封上，信封看起来是个多余的东西。但没有信封的话，每个邮递员要读信才知道发件人要干什么，这样做至少有两点问题：1）邮递员并不关心信件的内容，但它要读完信件才知道要发送的地址，非常浪费时间。2）侵犯了隐私权。所以通过统一格式的信封不仅减轻了邮递员的负担，同时也不会担心别人侵犯了个人的隐私。对物流来说也一样，增加编织袋后，物流就只关心发送站和目的站，不关心里面是信件还是包裹，要送给哪个人。

网络通信也是这样，发送方需要一层层“封装”信息，接收方需要一层层“解包”信息。这样不仅方便实现和维护，而且由于模块化设计，隔离层之间的变动不会相互影响，相对比较安全可靠。

基于分层设计的 TCP/IP 通信的基本过程如图 1－2。

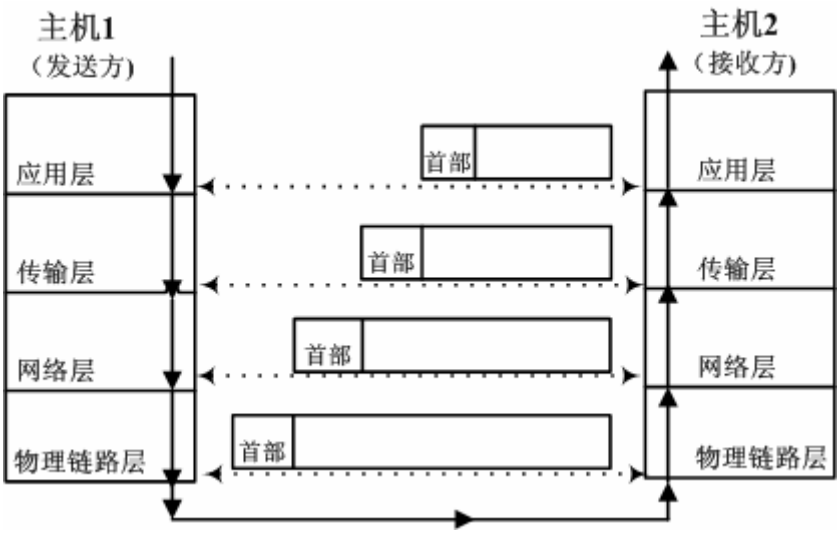

图 1－2 TCP/IP 数据传输过程

当发送方的每一层收到其上一层传来的数据后，都要加上本层的首部，然后再传给其下一层。这一层并不知道上一层给它的数据中哪些是用户需要发送的真正数据，它把上一层的协议首部和数据都看成自己的数据。这个过程就称之为“封装”。比如网络层的 IP 协议接收到传输层 TCP 送过来的数据后，它并不知道传输的是 FTP 还是 http 或其它应用层协议的内容，只知道传过来的数据都是 TCP 数据，因此它增加一个 IP 首部，在首部的协议字段中填写 TCP 协议值（6），并且填写好总长度，校验和等选项后送给物理链路层的以太网协议。以太网层软件再在所有数据前增加一个以太网帧首部后发送出去。

在接收方每一层收到相邻的下一层送来的数据后，将本层协议的首部去掉后交给其上一层。这个过程就称之为“解包”。经过层层剥离后，真正的数据交给等待数据的应用程序。

1．3 单片机控制的网络硬件框图

嵌入式系统应用广泛，一些大型的通信基站（常常建立在高山上的铁塔）里的控制器就是 ARM 或 PowerPC 等嵌入式处理器，其网络功能比电脑里的通讯要强很多。但开发软件需要考虑很多场景，代码都是几十上百万行，开发测试人员上百人，投入成本很大，不适宜个人学习 TCP/IP 协议和常用的简单控制。我们要研究的基于单片机控制的网络应用，软硬件要求起点低，成本很低，开发出来具有良好的经济价值。

硬件平台结构如 1－3 所示：其中单片机选 51 系列单片机，要求程序存储器大于 16K（一般的增强型 51 如 PHILIPS 公司的 P89C51RD2，STC 的 STC89C516RD 的程序存储器都接近 64K），由于单片机内部 RAM 比较小，因此需要外接 32K 的 RAM，一般选用 62256 芯片。与以太网接口的芯片采用 RealTek 公司的 RTL8019AS，RTL8019AS 是 10Mb/s 以太网接口芯片，ISA 接口。同时通过 MAX232 与 PC 机或其它调试机连接，可以显示调试或相关信息。如果需要使用 EEPROM 存储相关的 IP、MAC 等信息，可以外接 I2C 接口的芯片，如 24C02。不过很多单片机内部都集成了 EEPROM，如 STC 的单片机，因此也可以省略存储电路。为保证程序的可靠，看门狗也是必须的，不过很多单片机也是片内集成硬件看门狗，具体可参考芯片手册。由此可见，实现单片机上网的硬件电路比较

简单，因此成本也低。由于单片机最小系统（晶振，电源等）和所接芯片（MAX323,RTL8019,RAM 等）都有成熟的电路模块，可参考相关芯片手册，这里不再详述。与 RTL8019AS 的连线介绍也可参考《第三章 网络芯片的驱动》。

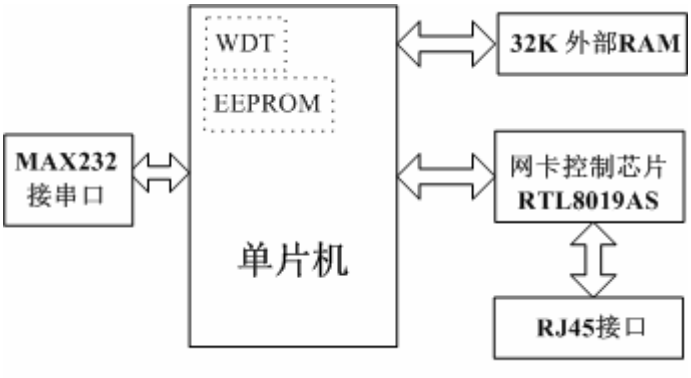

图 1－3 基于单片机控制的硬件框图

1．4 常用嵌入式 TCP/IP 协议栈

随着网络的发展，越来越多的人投入到网络协议栈的研究，开发出许多优秀的协议栈。一些著名的嵌入式操作系统都带有强大的 TCP/IP 功能，如 vxWorks，linux。同时，也涌现了许多一些源代码公开的免费协议栈，目前较为著名的免费开源协议栈有：lwIP、uIP、openTCP、TinyTCP 等。下面介绍在嵌入式系统应用很广泛的 lwIP 和 uIP 协议栈。

lwIP：是 TCP/IP 协议栈的一个小型实现，支持的协议功能比较完整，包括带 IP 和 ICMP 的 TCP 和 UDP 传输层。一般需要多任务环境支持（也可以移植到没有操作系统的环境下运行），仅协议栈的代码占用 ROM>40KB，RAM>10K 以上，不适合 8 位机系统。目前流行的移植版本是基于 ucos2 操作系统上和 ARM7 控制系统。

uIP：uIP 是专门为 8 位和 16 单片机设计的一个非常小的 TCP/IP 协议栈。uIP 完全用 C 编写，可以很方便的移植到各种不同的结构和操作系统上。一个编译的栈可以有几 KB R OM 或几百字节 RAM 中运行。并且其硬件处理层、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，极大的节省空间和时间。由于结构简单、功能可靠，很多 8 位单片机都移植 uIP 协议栈。

1．5 单片机 TCP/IP 网络特点

单片机在嵌入式领域中属于很低端的处理器，其资源和处理能力有限。首先，它的程序存储空间 ROM 和数据存储空间 RAM 都不能超过 64K。其次，它的 CPU 频率一般在 12M～40M 之间，指令的执行速度相对慢，导致网络传输速度很难上去。据计算，12M 的 51 单片机的最快网络传送速度大概为 25KB/s。

这些特点，导致单片机上不可能运行大而全的 TCP/IP 协议栈。需要使用精简的嵌入式协议栈。虽然处理速度和通讯流量不允许很大，但对于一般的工业、楼宇、家居智能化等领域也足够应付。更重要的是，它的成本很低，具有很高的经济价值。

考虑到这些特点，uIP 协议栈是一个不错的选择。因此本书以图 1－3 基于单片机控制的硬件和 uIP 协议栈介绍单片机的 TCP/IP 网络编程与应用。

第二章　uIP 协议栈分析

2．1　uIP 特性

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，所有代码和相关说明文档可以到 http://dunkels.com/adam/uip/下载。最新版本是 uIP1.0 版本，本书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。

2．2　uIP 架构

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2－1所示。

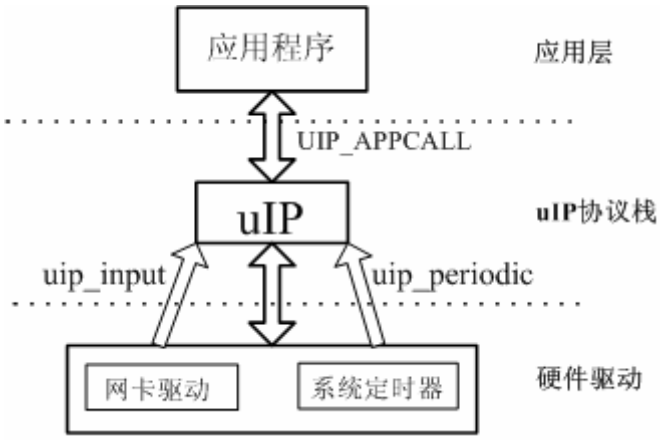

图2－1 uIP在系统中的位置

从上图可以看出，uIP 协议栈主要提供了三个函数供系统底层调用。即 uip_init(), uip_input() 和 uip_periodic() 。其与应用程序的主要接口是

UIP_APPCALL( )。

uip_init()是系统初始化时调用的，主要初始化协议栈的侦听端口和默认所有连接是关闭的。

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。

2．3　uIP 在 MCS-51 单片机上的移植

1．为此项目建立一个 keil C 工程，建立 src 目录存放源文件。

2．通过阅读 uip-1.0\unix\main.c，了解 uIP 的的主循环代码架构，并将 main.c 放到 src 目录下。

3．仿照 uip-1.0\unix\tapdev.c 写网卡驱动程序，与具体硬件相关。这一步比较费点时间，不过好在大部分网卡芯片的驱动程序都有代码借鉴或移植。驱动需要提供三个函数，以 RTL9019AS 驱动为例。

etherdev_init（）：网卡初始化函数，初始化网卡的工作模式。

u16_t etherdev_read(void) ：读包函数。将网卡收到的数据放入全局缓存区 uip_buf 中，返回包的长度，赋给 uip_len。

void etherdev_send(void)：发包函数。将全局缓存区 uip_buf 里的数据（长度放在 uip_len 中）发送出去。

所以，收包和发包主要是操作 uip_buf 和 uip_len。具体驱动分析可参考《第三章　网络芯片的驱动》。

4．由于 uIP 协议栈需要使用时钟，为 TCP 和 ARP 的定时器服务。因此使用单片机的定时器 0 用作时钟，每 20ms 让计数 tick_cnt 加 1，这样，25 次计数（0.5S）满了后可以调用 TCP 的定时处理程序。10S 后可以调用 ARP 老化程序。对 uIP1.0 版本，增加了 timer.c/timer.h，专门用来管理时钟，都放到 src 下。

5．uIP 协议栈的主要内容在 uip-1.0\uip\下的 uip.c/uip.h 中，放到 src 下。如果需要 ARP 协议，需要将 uip_arp.c 和 uip_arp.h 也放到 src 下。

6．uipopt.h/uip-conf.h 是配置文件，用来设置本地的 IP 地址、网关地址、MAC 地址、全局缓冲区的大小、支持的最大连接数、侦听数、ARP 表大小等。需要放在 src 下，并且根据需要配置。在 V1.00 版本中对配置做了如下修改：

（1）配置 IP 地址，默认先关 IP，在初始化中再设定。

```
#define UIP_FIXEDADDR   0
#define UIP_IPADDR0     192
#define UIP_IPADDR1     168
#define UIP_IPADDR2     1
#define UIP_IPADDR3     9
#define UIP_NETMASK0    255
#define UIP_NETMASK1    255
#define UIP_NETMASK2    255
#define UIP_NETMASK3    0
#define UIP_DRIPADDR0   192
#define UIP_DRIPADDR1   168
#define UIP_DRIPADDR2   1
#define UIP_DRIPADDR3   1
```

（2）使能 MAC 地址

```
#define UIP_FIXEDETHADDR 1
#define UIP_ETHADDR0    0x00
#define UIP_ETHADDR1    0x4f
#define UIP_ETHADDR2    0x49
#define UIP_ETHADDR3    0x12
#define UIP_ETHADDR4    0x12
#define UIP_ETHADDR5    0x13
```

（3）使能 ping 功能

```
#define UIP_PINGADDRCONF  1
```

（4）关闭主动请求连接的功能

```
#define UIP_ACTIVE_OPEN 0
```

（5）将 uip_tcp_appstate_t 定位 u8_t 类型。

（6）由于单片机是大端结构，因此宏定义需要修改

```
#define UIP_CONF_BYTE_ORDER      UIP_BIG_ENDI      AN
```

（7）暂时不移植打印信息，先关闭

```
#define UIP_CONF_LOGGING        0
```

（8）定义数据结构类型

```
typedef unsigned char u8_t;
typedef unsigned int u16_t;
typedef unsigned long u32_t;
```

7. 如果使用 keil C 的小模式编译，需要在大部分的 RAM 的变量前增加 xdata。

8. data 为 keil C 的关键词，代码中所有出现 data 的地方（主要是参数、局部变量、结构体成员）改为 pucdata 或 ucdata。

9. 解决编译过程中的错误。uIP 协议栈为 C 语言编写，编译过程中的问题比较少，并且容易解决。

2．4 uIP 的主控制循环

通过实际的代码说明 uIP 协议栈的主控制循环。

```
void main(void)
```

```
{
    /*省略部分代码*/
    /*设置 TCP 超时处理时间和 ARP 老化时间*/
  tim    er_set(&periodic_timer, CLOCK_CONF_SECOND / 2);
  tim    er_set(&arp_timer, CLOCK_CONF_SECOND * 10);
    /*定时器初始化*/
  init_T    imer();
    /*协议栈初始化*/
  uip_init();
    uip_arp_init();
    /*应用层初始化*/
  exam    ple1_init();
    /*驱动层初始化*/
  etherdev_init();
    /*IP 地址、网关、掩码设置*/
  uip_ipaddr(ipaddr    , 192,168,1,9);
  uip_sethostaddr(ipaddr);
  uip_ipaddr(ipaddr    , 192,168,1,16);
  uip_setdraddr(ipaddr);
  uip_ipaddr(ipaddr    , 255,255,255,0);
  uip_setnetm    ask(ipaddr);
    /*主循环*/
    while(1)
    {

    /*从网卡读数据*/
  uip_len     = etherdev_read();
    /*如果存在数据则按协议处理*/
  if(uip_len     > 0)
    {
         /*收到的是 IP 数据，调用 uip_input()处理*/
    if(BUF->type        == htons(UIP_ETHTYPE_IP))
          {
            uip_arp_ipin();
            uip_input();
            /*处理完成后，如果 uip_buf 中有数据，则调用 etherdev_send 发
            送出去*/
            if(uip_len > 0)
               {
            uip_arp_out();
               etherdev_send();
            }
          }
     /*收到的是 ARP 数据，调用 uip_arp_arpin()处理*/
```

```
else        if(BUF->type == htons(UIP_ETHTYPE_ARP)) {
    uip_arp_arpin();
          if(uip_len > 0)
            {
              etherdev_send();
          }
         }
       }
     /*查看 0.5S 是否到了，到了则调用 uip_periodic 处理 TCP 超时程序*/
    els       e if(timer_expired(&periodic_timer))
       {
            timer_reset(&periodic_timer);
      for(i          = 0; i < UIP_CONNS; i++)
           {
             uip_periodic(i);
             if(uip_len > 0)
              {
                uip_arp_out();
                etherdev_send();
              }
              }
              /*查看 10S 是否到了，到了则调用 ARP 处理程序*/
       if(tim            er_expired(&arp_timer))
              {
                timer_reset(&arp_timer);
               uip_arp_  timer();
              }
        }
   }
  return  ;
}
```

2．5　uIP 协议栈提供的主要接口

提供的接口在 uip.h 中，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。

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()

第三章　网络芯片的驱动

从最底层驱动来看，单片机是怎么从物理线路上接收到数据？接收的又是什么样的数据？通过对本章以太网帧管理和网络芯片驱动的介绍，可以了解数据是怎样上送和下发的，从而了解物理层和数据链路层的工作原理。

3．1　以太网的帧结构

现在是网络时代，大家都喜欢在网上冲冲浪。但电脑能上网的前提条件是电脑硬件中必须有网卡，如板载网卡、独立网卡、或无线网卡等。网卡是电脑与网络相互连接的必需设备。

单片机要上网，无疑也需要一个网卡。网卡的功能主要有：一是将需要发送的数据封装成物理传输帧，并通过网线(或电磁波)发送到网络上去；二是识别和接收网络上其它设备传过来的物理传输帧，并组合成数据提供给本机 CPU。网线中传输的是电信号，因此网卡具备将数据转化为电信号发出去，并将网线上的电信号解析成数据的功能。同时，要想电信号都能被其它网卡解析和接收，网卡必须满足网络传输规定的机械和电气规范。

从 TCP/IP 的协议分层的结构上看，网卡至少完成物理层的功能。但为了方便控制器的使用，现在的网卡还将数据链路层的功能集成到芯片中。网卡能起到过滤作用，只接受广播帧/组播帧或专门发往本机物理地址的数据帧，对本网络上的其余帧不予理会。网卡接收到一帧完整的数据后，就可以将数据交给网络层处理。

对于使用网卡的开发者来说，对网卡的物理结构，硬件检测机制和冲突退避算法不需深究，主要要了解下相应的物理传输帧结构。根据组建的网络类型不同（如以太网，令牌环网等）和使用的底层网络协议不同，帧结构也不相同。就常用的以太网来说，也有 Ethernet_II，IEEE 802.3，IEEE 802.2 等协议。我们以 IEEE 802.3 数据帧来分析数据链路层和网卡的工作内容。

IEEE 802.3 是电气和电子工程师协会（IEEE）制定的一种描述物理层和数据链路层的实现方法的网络协议，主题是在多种物理媒体上以多种速率采用 CSMA/CD 访问方式。其规定的帧结构如下：

表 3－1　以太网（802.3）帧结构

PR	SD	DA	SA	LENGTH/TYPE	DATA	PAD	FCS
56 位	8 位	48 位	48 位	16 位	n ≤1500（字节）	可选	32 位

表一　以太网（802.3）帧结构

FR（前导码）：包括了 7 个字节的二进制“1”、“0”间隔的代码，即 1010…10 共 56 位。当帧在链路上传输时，接收方就能建立起同步，因为这种“1”、“0”间隔的传输波形为一个周期性方波。同时也指明了传输的速率（10M 和 100M 的方波频率不一样，所以 100M 网卡可以兼容 10M 网卡）。

SD（帧数据定界符）：它是长度为 1 个字节的 10101011 二进制序列，此码表示表示下面跟着的是真正的数据。

DA（目的地址）：目的以太网的物理地址，由 48 位二进制组成(6 个字节)，说明该帧传输给哪个网卡。如果地址为 FFFFFFFFFFFF（广播地址），则该网络

上的所有网卡都能接收到本帧数据。这个地址和下面的 SA 就是我们常说的网卡的 MAC 地址。具体信息我们待会介绍。

SA（源地址）：48 位，说明该帧的数据是哪个网卡发的，即发送端的网卡物理地址（MAC）。

LENGTH/TYPE（长度/数据类型）：指示后面的数据属于什么类型。如 0800H 表示数据为 IP 包，0806H 表示数据为 ARP 包。这样，交给网络层后就可以由相应的协议对后面的数据解析。如果这个字段小于 0600H 的值，则表示数据包的长度，在单片机的网络编程中不考虑这种用法。

DATA（数据段）：由网络层负责发送和解析的数据，因为以太网帧传输的数据包最小不能小于 64 字节，最大不能超过 1518 字节。除去 14 字节为 DA、SA、TYPE 以及 4 字节的 FCS，DATA 不能超过 1500 字节。如果不够 46（64－18）字节，余下的由 PAD 填充。

PAD（填充位）：当 DATA 的数据不足 46 字节时，缺少的字节需要补上（可补任意值）。

FCS（帧校验序列）：由 32 位（4 字节）循环冗余校检码（CRC）组成，其校验范围不包括前导码 FR 及帧数据定界符 SD。此序列由发送端网卡自动生成，自动填充到帧的最后。一般情况下，接收端网卡对收到的数据校验后也不会将 FCS 放到数据中上报。

由于网卡的自动管理，并且前导码 FR 和帧数据定界符 SD 的值是固定的，也由网卡自动生成和插入。所以，网络层向网卡发送的数据或者网络层接收到的数据一般是由 DA、SA、TYPE 和 DATA 组成（DATA 不足 46 字节需要用 PAD 补齐）。如表二。假如网络层的一个 IP 包要发送出去，首先要填充接收网卡的地址和本网卡的地址（MAC 地址），同时将 TYPE 填充成 0800H，紧跟着就是发送的数据。网卡获取到这些数据后会组成物理传输帧发送出去。

表 3－2　网络层管理的帧结构

DA	SA	TYPE	DATA	PAD
48 位	48 位	16 位	n ≤1500（字节）	可选

由以太网的帧结构知道，在数据链路层就需要使用 MAC 地址（即物理地址）进行通讯，MAC 地址是数据的第一道关卡，由硬件自动识别来接收。因此，MAC 地址就像是是网络设备的“身份证”一样，需要具有全球唯一性。我们在实验室里做测试，可以修改 MAC 地址，但也应该保证本地网络里 MAC 的唯一性。

以太网的 MAC 地址由 48bit (6 字节)组成，如 08:02:10:3A:85:23 就是一个 MAC 地址。前 24 位（08:02:10）是由生产网卡的厂商向 IEEE 申请的厂商地址，后 24 位（3A:85:23）是由厂家自己分配。每个厂商必须确保它所制造的每个以太网设备都具有相同的前三个字节以及不同的后三个字节，这样就可保证世界上每个以太网设备都具有唯一的 MAC 地址。网卡的 MAC 地址通常是由生产厂家烧入网卡的 EPROM 中（NE2000 系列网卡常用 93C46，在网卡上可以找到）。

MAC 地址又可以分成 3 类：

（1）广播地址：只能用作目的地址。如果一个以太网帧的目的地址是广播

地址，则网络中的所有设备都能接收和处理该帧。广播地址的每一位都是 1，即：FF:FF:FF:FF:FF:FF。

（2）组播地址：也只能用作目的地址。如果一个帧的目的地址是组播地址，那么网络中预先定义的一组设备都能接收并处理该设备。以太网网 MAC 的最高有效字节的最低位为 1 表示以太网帧是组播帧。如 01:03:52:3A:85:23 就是组播地址。101 的最低位为 1。

（3）单机地址：除了广播和组播的地址就是单机地址，非广播和组播数据包就要与这个地址匹配了才能被接收和处理。

3．2　以太网的芯片 RTL8019 介绍及编程

网卡主要由网络芯片、数据汞、总线插槽接口、EEPROM、晶振、RJ45 接口、指示灯、固定片等器件等组成，其核心是网络芯片。这节我们重点介绍 Realtek 公司生产 RTL8019AS 芯片，看看它是怎样实现上节提到的以太网的帧结构的。

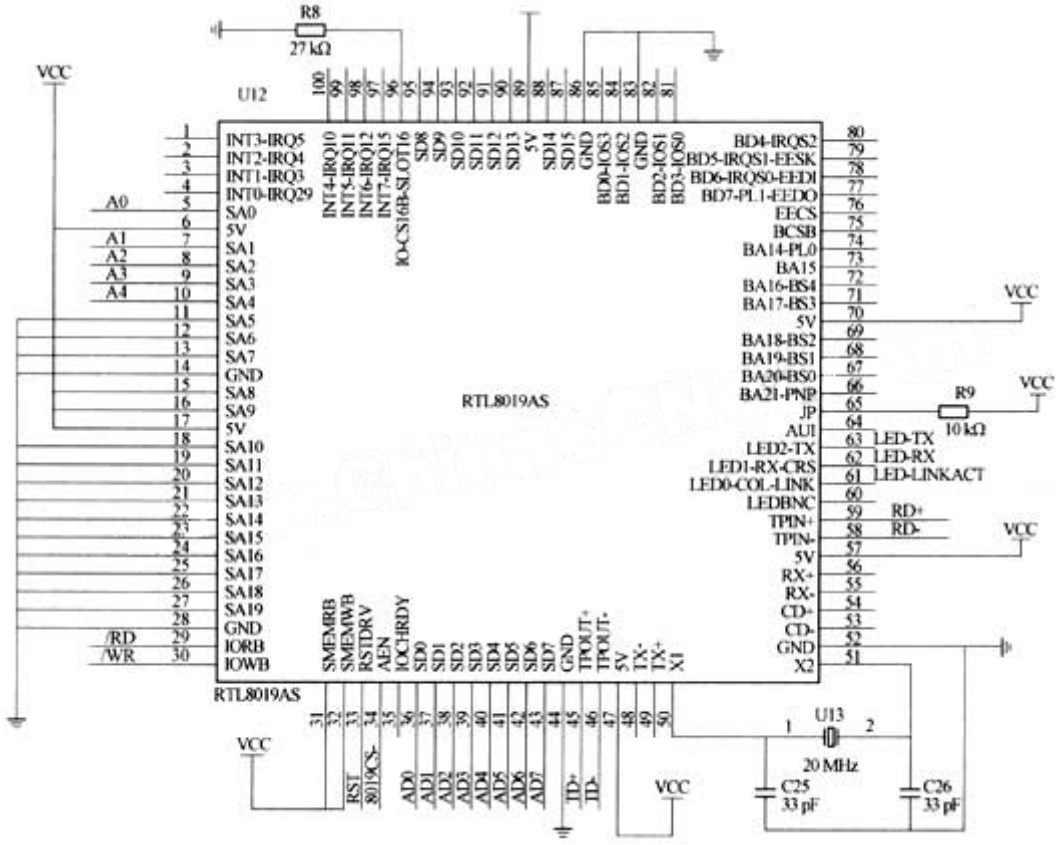

图 3－1 R TL8019 硬件电路图

3.2.1　RTL8019 的工作方式及与单片机的接口

RTL8019 是 NE2000 系列中一种，因此其它 NE2000 系列网卡编程思路也相符合的。（NE2000 是一个由 Novell 公司所创立并且被业界广泛采用的网络卡的标准，大多数 ISA 网卡都与 NE2000 兼容，如 Nat Sem i 的 DP83902，Davicom 的 DM9008,NSI 的 DP8390,MXIC 的 MX98905 等）。RTL8019 是 10M 网卡芯片。

RTL8019 有 3 种工作方式：（1）跳线方式，网卡的 i/o 和中断由跳线决定。（2）免跳线方式，网卡的 i/o 和中断由外接的 93c46 里的内容决定。（3）即插即

用方式，由软件进行自动配置 plug and play（PnP）。电脑上常使用即插即用方式方式，单片机不予考虑。对免跳线方式，需要在芯片外部接 EEPROM 93c46（很多电脑用的网卡上能找到这芯片，大家有兴趣可以看下），用来保存网卡的 MAC 地址等参数，同时，需要单片机操作 RTL8019 来控制 93c46 的读写，不仅费钱还费力，一般也不采用。参数保存可放在单片机自身的 EEPROM 里，上电后写入网卡芯片就可以了。因此，单片机比较适合使用跳线方式。

怎么才能让 RTL8019 工作在跳线方式下呢？这个由 JP 管脚（第 65 脚）决定。当 JP 管脚为低电平时，RTL8019 工作在即插即用方式或免跳线方式，高电平时处于跳线方式。因此，我们将 JP 管脚接高电平（VCC），使用跳线方式，网卡的 I/O 和中断就不是由 93C46 的内容决定而是由跳线决定。主要使用的跳线如下：

IOS3，IOS2，IOS1，IOS0 管脚（第 85，84，82，81 脚）决定了芯片的 I/O 地址。如下图，我们将 IS03-IO0 都悬空（内部下拉，即都为 0），则芯片 I/O 的起始地址是 300H。

IOS3	IOS2	IOS1	IOS0	I/O Base
0	0	0	0	300H
0	0	0	1	320H
0	0	1	0	340H
0	0	1	1	360H
1	0	0	0	380H
1	0	0	1	3A0H
1	0	1	0	3C0H
1	0	1	1	3E0H
0	1	0	0	200H
0	1	0	1	220H
0	1	1	0	240H
0	1	1	1	260H
1	1	0	0	280H
1	1	0	1	2A0H
1	1	1	0	2C0H
1	1	1	1	2E0H

芯片的中断线由 IRQS2，IRQS1，IRQS0 管脚（第 80，79，78 脚）决定。我们可以使用默认配置，并将网卡的中断 9，接到单片机的中断 INT0（P3.2）上。

IRQS2	IRQS1	IRQS0	Interrupt Line	Assigned ISA IRQ
0	0	0	INT0	IRQ2/9
0	0	1	INT1	IRQ3
0	1	0	INT2	IRQ4
0	1	1	INT3	IRQ5
1	0	0	INT4	IRQ10
1	0	1	INT5	IRQ11
1	1	0	INT6	IRQ12
1	1	1	INT7	IRQ15

芯片的 BTRM 地址由 BS4,BS3,BS2,BS1 管脚（第 72，71，69，68，67）决定，BTRM 在电脑里用来做无盘工作站的时候用到，这样可以从网卡进行引导，而需要从硬盘的 c 盘等引导系统。由于单片机里不需要使用，可以不管。

AUI 管脚（第 64 脚），该管脚决定使用 AUI 还是 BNC 接口。高电平时使用 AUI 接口，悬空（为内部下拉的低电平）时使用 BNC 接口。网卡的接口一般是 BNC 的可以支持 8 线双绞或同轴电缆。因此将该引脚悬空即可。

网络媒体类型由 PL0,PL1（第 74，77 脚）决定。

PL1 PL0		网络媒体类型
0 0		TP/CX 自动检测（10BaseT 链环测试时激活的）
0 1		10BASET 链环测试是不可用的。
1 0		10BASE5
1 1		10BASE2

其它重要管脚的连线：

所有 GND 管脚接地，VCC 管脚接＋5V 的电源。

RSTDRV(第 33 脚)：用来复位网卡，我们一般将它连到单片机的管脚上，用来快速复位 RTL8019，这个管脚拉高复位持续时间至少 2ms，建议拉高 100ms 左右再拉低来达到复位效果。

IOCS16B 管脚（第 96 脚）是 16 位/8 位数据位的选择脚。如果这个管脚下拉，则选择 8 位模式，如果这个管脚接高电平，将选择 16 位的模式。由于单片机是 8 位的数据总线，因此这个管脚需要下拉。

IOWB（第 30 脚），IORB（第 29 脚）接到单片机的 P3.6，P3.7（/WR ，/RD），用来控制读写时序。

X1 和 X2（第 50，51 脚）用来接 20M 的晶振。TPIN+，TPIN-管脚（59，58）和 HD,LD 管脚（45，46）都接隔离变压器上，通过 RJ45 接口与网络相连。

由于使用 8 位数据位，SD0—SD7 管脚接到单片机的 P0.0－p0.7 管脚，SD8－SD15 不需要使用。

SA0—SA19 为网卡的地址线，共 20 根，这个需要怎么接呢？前面说过，IOS3，IOS2，IOS1，IOS0 管脚决定了芯片的 I/O 地址。假如将 IS03-IO0 都悬空（内部下拉，即都为 0），则芯片 I/O 的起始地址是 300H。从 RTL8019 的芯片手册中知道，I/O 访问的寄存器只有 32 个，在后面会介绍这些寄存器。即需要直接访问的网卡寄存器是 300H－31FH。将它转化为二进制的地址线的电平就很直观了。如下表。

地址线	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
300H	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
...	0	0	0		0	0	0	0		0	0		0	1	1	0	0	0	X	X	X	X	X
31FH	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1	1

从表中知道，SA19－SA10,SA7－SA6 固定接低电平（地），SA9-SA8 固定接高电平（VCC）。只要控制 SA4－SA0 就可以访问到 300H 到 31FH 的 32 个寄存器。于是，我们将这 5 根地址线接到单片机的 P0.4－P0.1 上。为了采用外部地址访问，将 P2 口的 p2.7 接到 RTL8019 的片选 CS 上。这样，如果单片机访问 8000H

地址就相当于访问 RTL8019 的 300H 了，访问 801FH 地址就相当于访问 RTL8019 的 31FH。地址映射关系为：

单片机地址端口 P2，P0	网卡(I/O)
10000000 00000000 （8000H)	300H
10000000 000XXXXX	3XXH
10000000 0001 1111 (801FH)	31FH

程序里对寄存器使用如下定义来访问：

```
#include <absacc.h>
/* 定义了 XBYTE：#define XBYTE ((unsigned char volatile xdata *) 0)  */
#define reg00  XBYTE[0x8000] /   * 300H*/
#define reg01  XBYTE[0x8001] /   * 301H*/
#define reg02  XBYTE[0x8002]
#define reg03  XBYTE[0x8003]
#define reg04  XBYTE[0x8004]
#define reg05  XBYTE[0x8005]
#define reg06  XBYTE[0x8006]
#define reg07  XBYTE[0x8007]
#define reg08  XBYTE[0x8008]
#define reg09  XBYTE[0x8009]
#define reg0a  XBYTE[0x800a]
#define reg0b  XBYTE[0x800b]
#define reg0c  XBYTE[0x800c]
#define reg0d  XBYTE[0x800d]
#define reg0e  XBYTE[0x800e]
#define reg0f  XBYTE[0x800f]
#define reg10  XBYTE[0x8010]
#define reg18  XBYTE[0x8018]
#define reg1c  XBYTE[0x801c]  /*    31cH*/
#define reg1f  XBYTE[0x801f]  /*    31fH*/
```

单片机通过地址线访问的 RTL8019 寄存器也就是以上 32 个寄存器。但 RTL8019 内部的空间不仅仅是这 32 个寄存器，它还有 256 字节的 PROM 和 16K 字节的 RAM BUFFER。如下图。PROM 内容是网卡复位时从 93c46 里读出来的部分参数和配置。由于单片机使用跳线方式，不接 93C46，所以这段空间不用关注。RAM BUFFER 用来存放需要网口发送出去的数据和网卡自动接收到的数据。相当于以太网帧的缓存。单片机对这段缓存的操作并不需要按地址去访问，而是通过前面的 32 个寄存器间接读取。

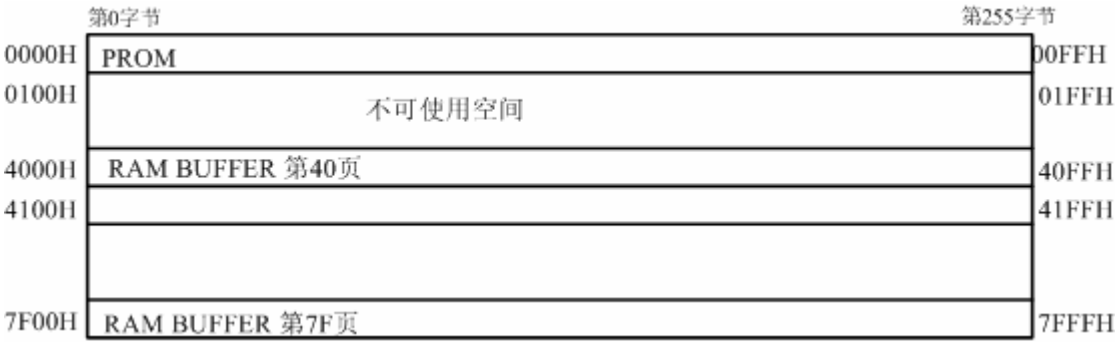

图 3－2 R TL8019 RAM BUFFER 结构

RAM BUFFER 空间以页为存储单位,每 256 字节为一页。如 4000H 到 40FFH 为一页，我们称之为第 40 页。从 40 页到 7F 页共有 64 页。以页为单位要怎么使用呢？打个比方，我们把 4600H－7FFFH 的地址作为接收缓存，当网卡收到 64 字节的一帧数据时，存放在 4600H 开始的页上，再收到 280 字节的一帧数据时，并不接着 4664 地址放，而是放在 4700 开始的页，占用两页空间。具体对 RAM BUFFER 的操作在网卡数据收发时描述。

3.2.2 RTL8019 的寄存器介绍

从上节知道，能通过地址线访问 RTL8019 的寄存器共有 32 个，每个寄存器是 8bit（1 字节）结构。其中 reg00－reg0f 共 16 个寄存器为通用寄存器，reg10－reg17 共 8 个寄存器为远程 DMA 数据读写寄存器，对 8 位数据位的单片机来说，只需用到 reg10，用来对 RAM BUFFER 里数据的获取和写入。reg18－reg1f 共 8 个寄存器为复位寄存器，只要对其中偶数地址的寄存器读或者写，就会复位网卡，我们称为热复位网卡（使用 RSTDRV 管脚复位称为冷复位，一般使用冷复位就行了）。对单片机来说，也只需使用 reg18。

下面重点介绍 reg00－reg0f 寄存器。RTL8019 通用寄存器有四页（page），也就是说 reg00－reg0f 处在不同页的话含义可能不一样，当寄存器处在第一页和第 4 页时，对同一寄存器读和写的含义也可能不一样。详细见下表。这样计算，RTL8019 可读写的通用寄存器远不止 16 个。怎么去访问呢？（1）通过命令寄存器 CR 中的 PS1 和 PS0 决定访问的页面。（2）地址线决定访问的寄存器是 0x0 到 0xf 中的哪个。（3）读/写信号决定是读操作还是写操作中的寄存器。假设定义 ucTemp 为 unsigned char 型变量，PS1=0,PS0=0,使用 reg01＝0x46，即将 0x46 写入 PSTART。使用 ucTemp＝reg01，即将 CLDA0 中的值读到 ucTemp 中。

寄存器（地址）	第 0 页		第 1 页	第 2 页	第 3 页	
	读操作	写操作	读/写	只能读	读操作	写操作
reg00	CR	CR	CR	CR	CR	CR
reg01	CLDA0	PSTART	PAR0	PSTART	9346CR	9346CR
reg02	CLDA1	PSTOP	PAR1	PSTOP	BPAGE	BPAGE
reg03	BNRY	BNRY	PAR2	—	CONFIG0	—
reg04	TSR	TPSR	PAR3	TPSR	CONFIG1	CONFIG1
reg05	NCR	TBCR0	PAR4	—	CONFIG2	CONFIG2
reg06	FIFO	TBCR1	PAR5	—	CONFIG3	CONFIG3

reg07	ISR	ISR	CURR	—	—	TEST
reg08	CRDA0	RSAR0	MAR0	—	CSNSAV	—
reg09	CRDA1	RSAR1	MAR1	—	—	HLTCLK
reg0a	8019ID0	RBCR0	MAR2	—	—	—
reg0b	8019ID1	RBCR1	MAR3	—	INTR	—
reg0c	RSR	RCR	MAR4	RCR	—	FMWP
reg0d	CNRT0	TCR	MAR5	TCR	CONFIG4	—
reg0e	CNTR1	DCR	MAR6	DCR	—	—
reg0f	CNTR2	IMR	MAR7	IMR	—	—

由于第 3 页不是标准的 NE2000 系列定义的寄存器，并且主要是对 93c64 的操作和网卡的配置，在单片机使用跳线方式下可以不用理会。下面重点介绍前 3 页的寄存器。当然，要完全掌握寄存器的使用，还是要根据后面的程序分析下。

1．CR（命令寄存器）

从寄存器表知道，每页的 reg00 都是 CR 命令，因为在任何一页要跳转到其它页需要依赖 CR 命令。其功能有：（1）选择寄存器中的页面。（2）定义数据传输模式。（3）启动/停止数据收发。

CR 命令的位格式如下：

位	7 6 5 4 3	2 1 0						
名字	PS1	PS0	RD2 RD	1 RD0 TX	P ST		A	STP

CR 的相应位的含义：

位	符号	描述			
7－6	PS1 PS0	PS1 PS0		含义	
		0 0		选择 0 页	
		0 1		选择 1 页	
		1 0		选择 2 页	
		1 1		选择 3 页	
5－3 RD2-RD0		RD2 RD1 RD0			含义
		0 0 0			不允许
		0 0 1			远程读
		0 1 0			远程写
		0 1 1			发送包
		1	×	×	结束或完成远程 DMA
2 TXP		这一位在置 1 时启动本地 DMA 进行帧发送，在发送完毕或终止时自动清零。写 0 没影响。			
1－0 ST	A STP	STA：命令生效位，当设置 CR 寄存器后将此位置 1，开始命令执行。STP：设置 1 时，网卡停止收发数据。			
		STA STP		功能	
		1 0		开始执行命令	

		0 1		停止命令

2．CLDA0、CLDA1（当前的本地 DMA 寄存器寄存器）

这两个寄存器在第 0 页的 reg01 和 reg02，只读，可以通过读这两个寄存器得到当前的本地 DMA 地址。关于本地 DMA 和远程 DMA 的介绍放在数据收发中讲解。

3．BNRY（边界寄存器）

BNRY在第0页的reg03，可读可写，用于防止接收缓存环把还没被主机（单片机）读的数据给写了。BNRY由单片机控制，其内容是指向某一页的地址，如58H，则说明单片机将前面的页的内容读走了，再读数据的话从58H的页开始读。读完后将下一页的地址写入BNRY。

4．TSR（发送状态寄存器）

在第 0 页的 reg04，只读。显示数据发送的状态。

位	符号	描述
7	OWC	超出窗口冲突范围，置1时说明因发送冲突而终止包的传输应。这时正常的重发操作将被启动.
6	CDH	网卡监测在一个传输完成后帧间的间隙的第一个6.4微秒期间的CD信号跳动在发送这个跳动失败时置1.
5	—	总为1
4	CRS	当传送过程中发现丢失了包就置1.
3	ABT	置1表明网卡由于冲突次数超过上限而取消传输.
2	COL	显示传输中至少发生了一次冲突，其次数记录在NCR中.
1	—	总为1.
0	PTX	显示传输完成，没错误.

5．NCR（冲突数寄存器）

这个寄存器记录发送一个包过程所发生的冲突次数，只读。

6．FIFO（FIFO 寄存器）

设置为回环模式时，发送出去的数据返回来后存放在FIFO寄存器中，可以用来检测芯片。

7．ISR（FIFO 寄存器）

在第0页的reg07，可读写，反映网卡的中断状态，上电后必须清零。

位	符号	描述
7	RST	当网卡进入复位状态时置1,当一条命令由CR确认后清零。当接收缓冲区溢出时置1,而读出一个包以后自动清零。
6	RDC	当远程DMA操作完成后置1.
5	CNT	当MSB的一个或多个网络计数器启动时置1.
4	OVW	当接收缓冲区满时置1.
3	TXE	发送一个包时 因冲突等原因失败时置1.
2	RXE	当接收到一个包时发现一下错误时置1,（CRC 校验错误、帧队列失败、丢失包）.

1	PTX	这一位显示发送无错误.
0	PRX	这一位显示接收无错误.

8．IMR（中断允许寄存器寄存器）

所有位都对应 ISR 寄存器中的相应位，相应位置1则使能相应的中断。

9．CRDA0、CRDA1（当前的远程 DMA 地址寄存器）

这两个寄存器在第 0 页的 reg08 和 reg09，只读，可以通过读这两个寄存器得到当前的远程 DMA 地址。

10．8019ID0、8019ID1（RTL8019ID 寄存器）

记录 RTL8019 的芯片 ID，用来区分其它 NE2000 芯片，分别存放 50H 和 70H。

11．RSR（接收状态寄存器）

位	符号	描述
7	DFR	延时状态，发送冲突时置1。
6	DIS	置1表示接收禁止。
5	PHY	当接收到的时组播包或广播包时置1，接收到本机MAC地址的包时清零
4	MPA	由于在监测模式或缺少缓冲区而丢失了接收包则置1，然后计数器CNTR2加1
3	—	总为1
2	FAE	接收的包有对齐错误时则置1，计数器CNTR0加1。
1	CRC	接收的包CRC错误时也置1，计数器CNTR1加1
0	PRX	显示接收完成没错误

12．CNRT0、CNRT0、CNRT0（接收错误统计寄存器）

见 RSR 寄存器中描述。分别统计了接收的包中对齐错误，CRC 错误，包丢失错误的计数。

13．PSTART、PSTOP（开始页和结束页寄存器）

用来设置接收缓存环的开始页地址和结束页地址。这个地址从 40H 到 80H 间选取。注意：PSTOP 是指结束页地址的下一页。所以最大页为 7FH，则 PSTOP 可设置为 80H。

14．TPSR（发送开始页寄存器）

用于设置要发送的包的起始页。即发送缓存起始地址的高 8 位。

15．TBCR0、TBCR1（发送字节计数）

用于设置要发送的包的长度(字节)。

16．RSAR0、RSAR1（远程起始地址寄存器）

用于设置远程 DMA 的起始地址。

17．RBCR0、RBCR1（远程自动增加计数寄存器）

用于记录远程 DMA 的数据字节数。

18．RCR（接收配置寄存器）

接收配置寄存器指定接收帧的操作方式和接收哪些帧的类型。

位	符号	描述
7	—	总为1。
6	—	总为1。
5	MON	置1监测模式。

4	PRO	如果为1，所有包的目标MAC地址都接收； 如果为0,只有当目标MAC地址和PAR0-5 的值一一对应才接收。
3	AM	如果为1，接收目标地址为组播地址的包； 如果为0，不接收目标地址为组播地址的包。
2	AB	如果为1，接收目标地址为广播地址的包。 如果为0，不接收目标地址为广播地址的包。
1	AR	1：长度小于64个字节的包也接收。 0：长度小于64个字节的包不接收。
0	SEP	1：包有接收错误也接收； 0：包有接收错误不接收。

19．TCR（传输配置寄存器）

传输配置寄存器决定网卡在发送帧时的行为方式，如是否填充 CRC。

位	符号	描述
7	—	总为1。
6	—	总为1。
5	—	总为1。
4	OFST	冲突消除使能
3	ATD	自动传输禁止。0:普通操作；1:接收组播模式。
2—1	LB1,LB0	LB1,LB0都为0时为普通模式。LB1为0、LB0为1时内部回环模式。 LB1为1时外部回环模式。
0	CRC	如果置1则在发送时不添加CRC，功能否则在发送时附加CRC。

20．DCR（数据配置寄存器）

数据配置寄存器决定了网卡的访问宽度是 8 位还是 16 位，是否配置环回。

位	符号	描述
7	—	总为1。
6—5	FT1,FT0	FIFO 的选择。
4	ARM	远程自动初始化。1，禁止帧命令发送。0，允许执行帧命令发送。
3	LS	回环选择.0:选择回环模式TCR的位1- 2则需要定义为回环的类型。1：普通模式。
2	LAS	该位必须设为0，网卡只支持双16 位的DMA模式则上电为1.
1	BOS	在字中高位字节在前INTEL 为0 反之为1。
0	WTS	字传输选择。0：字节长度的DMA传输。1：字长度的DMA传输。对单片机而言，选择8位数据宽度。

21．PAR0—PAR5（MAC 地址寄存器）

记录本机以太网的 MAC 地址。接收时依靠此地址过滤不是本机的数据包。

22．CURR（当前页寄存器）

存放着接收包的第一页的地址，由网卡自动控制。

23．MAR0－MAR7（组播地址寄存器）

提供给组播地址的过滤位。为了接收所有的组播帧，可以将组播地址寄存器中的所有位置 1。

3.2.3 RTL8019 的 RAM BUFFER 介绍

前面说过，RTL8019有64页RAM BUFFER用来作为发送和接收数据包的缓存。要收发数据，就需要单片机对其进行管理。从原理上讲，RAM BUFFER是一个双口RAM，也就是说单片机可以往RAM中读写数据，网卡控制器也可以自动往里面读写数据。前面所提到的“远程DMA地址”就是单片机往RAM BUFFER中读写数据用到的地址。“本地DMA地址”是指网卡控制器自动读写RAM BUFFER用到的地址。所以，“远程”和“本地”的概念仅仅为了区分是主机(单片机)还是网卡控制器操作RAM BUFFER。

单片机远程读/写操作：将读/写RAM BUFFER中的数据的起始地址Addr写到远程起始地址寄存器RSAR1、RSAR0（RSAR1=Addr >> 8；RSAR0=Addr&0xFF），将要读取的数据长度len写到远程自动增加计数寄存器RBCR1、RBCR0（RBCR=len >> 8；RBCR0=len&0xFF）,然后在CR寄存器中设置读/写，就能通过远程DMA数据读写寄存器（reg10）来读写RAM BUFFER。读数据就是将数据一个接一个从reg10取出,写数据就是将数据一个一个写入reg10,网络控制器会将这个数据写入RAM BUFFER相应的数据。

我们可以这样分配 RAM BUFFE R：发送缓存使用第 40H 页到 45H 页共1536（6x256）个字节，这样可以容纳最长的一个以太网数据帧。发送数据的时候，先通过“远程写”将数据写到 40H 页开始的 RAM BUFFER 中，然后将发送的开始页（0x40）写到发送开始页寄存器 TPSR 中，要发送的字节写入 TBCR1、TBCR0 中，最后将 CR 寄存器 RD2－RD0 置为 011，启动发送包。

图 3－3 RTL8019 的 RAM BUFFER 分配图

接收缓存使用余下的所有页（46H页到7FH页）。如上图。于是将PSTART=0x46，PSTOP＝0x80。接收缓存区其实是一个循环FIFO 队列。CURR为写入页指针，受网卡控制，每收到一页数据CURR自动加1。当CURR＝PSTOP（0x80）时自动指向PSTART（0x46）。BNRY 为读出页指针，由主机（单片机）程序控制。单片机每读完一页数据后将BNRY加1，一个数据包若占用5页，则读

完后将BNRY加5，同样，BNRY到PSTOP后翻转到PSTART。当CURR追上BNRY或CURR=0x7F&&BNRY=0x46 时说明接收缓冲区已经满了，网卡会停止接收数据包。

3.2.4 RTL8019 的初始化

以下几节主要通过程序讲解RTL 8019的使用。在介绍程序之前，假定寄存器按照如下规则定义的。可以看出，是按照寄存器手册命名的，其含义见前面介绍。

```
#define CR          reg00
   #define    PS1     0x80
   #define    PS0     0x40
   #define    RD2     0x20
   #define    RD1     0x10
   #define    RD0     0x08
   #define    TXP     0x04
   #define    STA     0x02
   #define    STP     0x01
#define RDMA_REG    reg10
#define RESET_REG    reg18

/*  页 0   读/写寄存器 */
#define BNRY        reg03
#define ISR         reg07
   #define RST       0x80
   #define RDC       0x40
   #define CNT       0x20
   #define OVW       0x10
   #define TXE       0x08
   #define RXE       0x04
   #define PTX       0x02
   #define PRX       0x01

/*  页 0   只读寄存器 */
#define CLDA0       reg01
#define CLDA1       reg02
#define TSR         reg04
#define NCR         reg05
#define FIFO        reg06
#define CRDA0       reg08
#define CRDA1       reg09
#define _8019ID0     reg0a
#define _8019ID1     reg0b
#define RSR              reg0c
#define CNTR0       reg0d
```

```
#define CNTR1       reg0e
#define CNTR2       reg0f

/*  页 0   只写寄存器 */
#define PSTART      reg01
#define PSTOP       reg02
#define TPSR        reg04
#define TBCR0       reg05
#define TBCR1       reg06
#define RSAR0       reg08
#define RSAR1       reg09
#define RBCR0       reg0a
#define RBCR1       reg0b
#define RCR         reg0c
   #define    MON     0x20
   #define    PRO     0x10
   #define    AM      0x08
   #define    AB      0x04
   #define    AR      0x02
   #define    SEP     0x01
#define TCR         reg0d
   #define    OFST    0x10
   #define    ATD     0x08
   #define    LB1     0x04
   #define    LB0     0x02
   #define    CRC     0x01
#define DCR             reg0e
   #define    FT1     0x40
   #define    FT0     0x20
   #define    ARM     0x10
   #define    LS      0x08
   #define    LAS     0x04
   #define    BOS     0x02
   #define    WTS     0x01
#define IMR             reg0f
```

RTL8019的初始化主要设置接收和发送缓存，设置读页指针和写页指针，配置接收和发送模式，同时需要设置本机MAC地址。具体实现可以参考下面的示例代码。由于使用的是uIP网络协议栈，因此网卡的MAC需要与协议层封装的MAC统一，因此使用预先定义好的宏定义UIP_ETHADDR0 －UIP_ETHADDR5。当然也可以从EEPROM中读取，可实现动态修改MAC的功能。

示例代码：

```
void etherdev_init(void)
{
```

```
    /*先复位芯片*/
  etherd   ev_reset( );
/*    先停止收发*/
   CR = RD2|STP;
  page(0);
 /*  40H页到45H页共1536（6x256）个字节为发送缓存，其余为接收缓存*/
  /*这样发送缓存可以容纳最长的一个以太网数据帧。*/
  /*设置接收缓存的起始和结束地址*/
PST    ART = ETH_RX_PAGE_START;
   PSTOP = ETH_RX_PAGE_STOP;
    /*BNRY指向接收缓存的首页*/
   BNRY = ETH_RX_PAGE_START;
    /*TPSR，发送缓存区首地址*/
   TPSR = ETH_TX_PAGE_START;
   RBCR1 = 0x00;
   RBCR0 = 0x00;
/*RCR    接收配置寄存器1100 1100,*/
    /*接收广播包，组播包，目的MAC与本机相同的包，*/
    /*接收包不小于64字节，包有接收错误不接收*/
   RCR = 0xcc;
    /*TCR传输配置寄存器 1 110 0000*/
    /*普通模式，发送时附加CRC*/
   TCR = 0xe0;
    /*DCR数据配置寄存器 1 100 1000*/
    /* 8位数据dma,字节长度的DMA传输*/
   DCR = 0xc8;
    /*IMR关所有中断*/
   IMR = 0x00;
  page(1);
    /*CURR  指向接收缓存的首页*/
   CURR = ETH_RX_PAGE_START;
    /*组播地址寄存器*/
  MAR0    = 0x00;
  MAR1    = 0x41;
  MAR2    = 0x00;
  MAR3    = 0x00;
  MAR4    = 0x00;
  MAR5    = 0x00;
  MAR6    = 0x00;
  MAR7    = 0x00;
  /****W   rite MAC address/以下为写入MAC地址*****/
  P   AR0 = UIP_ETHADDR0;
  P   AR1 = UIP_ETHADDR1;
  P   AR2 = UIP_ETHADDR2;
```

```
    P    AR3 = UIP_ETHADDR3;
    P    AR4 = UIP_ETHADDR4;
    P    AR5 = UIP_ETHADDR5;
  page(0);
      CR = RD2|STA;
       /*ISR清所有中断*/
       ISR = 0xff;
}
```

用到的page( )函数为页选择函数，入参是0－3。

```
void page(u8_t pnum)
{
 u8_t      temp;
     temp = CR;
 tem     p=temp & (~(PS1|PS0|TXP)); /*TXP位也应该清0，防止发送数据*/
     pnum = pnum <<6;
     CR = temp | pnum;
}
```

用到的etherdev_reset（）是冷复位单片机，即将RSTDRV拉高100ms来完全网卡。rst_pin 是单片机的管脚，类似定义：sbit rst_pin = P1^6。

```
void etherdev_reset(void)
{
      /*将复位管脚拉高100ms来复位芯片*/
    rst_pin     = 1;
    etherdev_delay_m    s(100);
    rst_pin     = 0;
    etherdev_delay_m    s(100);
}
```

3.2.5 RTL8019 的发包程序

对 uIP 网络协议栈来说，需要发送的数据已经存放在全局内存 uip_buf 里，数据包长度大于 54 字节（TCP+ETH 头部）时，使用 uip_appdata 指向应用层数据。而需要发送数据的长度使用 uip_len 记录。因此，发送数据的步骤：

（1）启动远程写，将 uip_buf 中的数据写到 RAM BUFFER 的发送缓冲区中。

（2）将发送的首页赋给 TPSR，长度（小于 60 时算 60）赋给 TBCR1, TBCR0。启动发送。

当然，放在 uip_buf 的前 14 个字节按照以太网帧（表 3－2）的 DA,SA,TYPE 排列好的，这样可以发送到对应 MAC 地址的网卡中。

示例代码：

```
void etherdev_send(void)
{
  u16_t    i;
  u8_t    *ptr;

  ptr    = uip_buf;
```

```
 page(0);
     /*先结束远程 DMA*/
     CR = RD2 | STA;
     /*等待上次发送完成*/
   while(    CR & TXP) continue;
     /*清远程 DMA 中断标志*/
   ISR     |= RDC;
     /*启动远程写，将数据写到 RAM BUFFER 中*/
   RSAR0     = 0x00;
   RSAR1     = ETH_TX_PAGE_START;
   RBCR0     = (u8_t)(uip_len & 0xFF);
   RBCR1     = (u8_t)(uip_len >> 8 );
     CR = RD1 | STA;
     /*uIP 将要发送的 TCP 头部数据放到 uip_buf 中，
       同时用 uip_appdata 指向应用层数据*/
     for(i = 0; i < uip_len; i++)
     {
         if(i == 40 + UIP_LLH_LEN)
          {
        ptr            = (u8_t *)uip_appdata;
          }
          RDMA_REG = *ptr++;
     }
     /*等待远程 DMA 操作完成*/
   while(!(ISR     & RDC)) continue;
     CR = RD2 | STA;
     /*配置网卡发送数据包*/
   TPSR     = ETH_TX_PAGE_START;
   if(uip_len     < ETH_MIN_PACKET_LEN)
     {
          uip_len = ETH_MIN_PACKET_LEN;
     }
   TBCR0     = (u8_t)(uip_len & 0xFF);
   TBCR1     = (u8_t)(uip_len >> 8);
     CR  = RD2  |TXP  | STA;
}
```

3.2.6 RTL8019 的收包程序

前面介绍过，以太网帧依次由 PR、SD、DA、SA、TYPE、DATA（PAD）、FCS 组成。其中 PR、SD、FCS 由网卡自动添加完成。所以发送依次往 RAM BUFFER 里存放 DA（6 字节）、SA（6 字节）、TYPE（2 字节）、DATA（最少 46 字节）就可以了。那接收的数据帧是否也这样排列呢？对 RTL8019 有一点点区别，它在接收数据的时候添加了 4 字节的头部。其中 1 字节状态（与 RSR 寄存器内容一致），1 字节指向下一个以太帧的页指针（也说明了本帧数据占用了几

页），还有 2 字节表示本帧数据长度。接收的数据结构如下图。

Receive Status	Next packet pointer	Receive bytes	DA	SA	TYPE	DATA	PAD
8 位	8 位	16 位	48 位	48 位	16 位	n ≤1500（字节）	可选

RTL 8019 接收的数据帧的结构

uIP 的数据接收后放在全局内存 uip_buf 中，接收的数据长度放在 uip_len 中，因此，发现一个完好的以太网帧后，将前 4 个字节去掉后依次存放在 uip_buf 中。

RTL8019 收包采用查询模式，当发现 RAM BUFFE R 接收缓冲区中有未处理的数据时，先读取 4 字节，计算出数据的长度和下一帧的页地址，然后启动远程读操作，将数据存放在 uip_buf 中。当然，还可以根据网络的实际情况增加一些传输中产生错误（如接收缓存满）的处理代码。

示例代码：

```
u16_t etherdev_poll(void)
{
  u16_t    len = 0;
  u16_t    i;
 u8_t     status;
  u8_t    next_rx_packet;
  u8_t    current;
    /*检查接收是否正确*/
    if(ISR & PRX)
    {
        ISR = RDC;
        /*先读 RTL8019 添加的 4 个字节*/
        RSAR0 = 0x00;
        RSAR1 = BNRY;
    RBCR0       = 0x04;
    RBCR1       = 0x00;

        CR = RD0 | STA;
        /*获取本帧数据的长度和下一帧页地址*/
        status = RDMA_REG;
    next_rx_packet       = RDMA_REG;
    len       = (RDMA_REG); //RDMA);
    len       += (RDMA_REG<<8);//<< 8;
        len -= 4;
      /*等待远程 DMA 操作完成*/
   while(!(ISR       & RDC)) continue;
      /*启动远程读数据*/
        CR = RD2 | STA;
        if(len <= UIP_BUFSIZE)
        {
            ISR = RDC;
```

```
        RSAR0       = 0x04;
                RSAR1 = BNRY;
        RBCR0       = (u8_t)(len & 0xFF);
        RBCR1       = (u8_t)(len >> 8);
                CR = RD0 | STA;
                for(i = 0; i < len; i++)
                {
            *(uip_buf           + i) = RDMA_REG;
                }
               while(!(ISR & RDC)) continue;
               CR = RD2 | STA;
            }
            else
            {
                len = 0;
            }
            /*更新 BNRY*/
        BNR     Y = next_rx_packet;
            page(1);
            current = CURR;
            page(0);
            /*RAM BUFFER 中没有数据时清 PRX*/
            if(next_rx_packet == current)
            {
                ISR = PRX;
            }
        }
        return len;
}
```