基于嵌入式 Web 服务器监控系统的研究与设计

摘　要：本系统采用 STC89C58RD+单片机和以太网控制器 RTL8019AS 作为主要器件，采用 uIP0.9 嵌入式 TCP/IP 协议栈，构建了一个接入 Internet 的低成本嵌入式监控系统平台。同时通过该平台实现了温度的网络监测和开关量的网络控制功能。

关键词： TCP/IP、uIP0.9、STC89C58RD+、RTL8019AS、嵌入式监控平台

Abstract: STC89C58RD+ MCU and Ethernet controller RTL8019AS are used as the main devices in this design. And uIP0.9 embedded TCP / IP protocol stack is also used. These elements provided a low-cost embedded platform to access the Internet. At the same time, the monitoring of the temperature and switch controlling is achieved by the platform.

Keywords: TCP/IP、uIP0.9、STC89C58RD+、RTL8019AS、embeded monitoring and controlling platform

Frank Luo Long.luoaoutlook.com

目　　录

1 前言 1

2 总体方案设计 2

2.1 方案设计 2
2.1.1 方案一 2
2.1.2 方案二 2
2.2 方案论证 3
2.3 方案选择 3

3 单元模块设计 4

3.1 系统功能模块设计 4
3.1.1 电源模块设计 4
3.1.2 单片机最小系统 4
3.1.3 串口通信电路设计 5
3.1.4 SRAM 外扩电路设计 5
3.1.5 RTL8019AS 网络接口电路设计 6
3.1.6 简单应用电路 7
3.2 特殊器件介绍 8
3.2.1 STC89C58RD+ 8
3.2.2 RTL8019AS 8

4 软件设计 9

4.1 TCP/IP 协议族 9
4.1.1 TCP/IP 协议简介 9
4.1.2 数据包封装 9
4.1.3 数据帧的分用 10
4.1.4 地址解析协议 ARP 10
4.1.5 网际协议 IP 11
4.1.6 控制报文协议 ICMP 11
4.1.7 控制传输协议 TCP 12
4.2 uIP 协议栈 13
4.2.1 uIP 协议栈的实现方法 13
4.2.2 uIP 协议栈的接口 14
4.2.3 uIP 在 51 系列单片机上的应用 16
4.2.4 uIP 底层接口 16
4.2.5 uIP 应用层接口 20
4.3 超文本传输协议:HTTP 20
4.3.1 HTTP 的请求方法 21
4.3.2 HTTP 的响应 21
4.3.3 简单的 Web 服务器 22
4.3.4 HTML 介绍 22

Frank_huo_long.luo@outlook.com

4.4 环形缓冲区的串口驱动 23
5 系统调试 28
5.1 ARP 协议调试 28
5.2 PING 命令调试 28
5.3 TCP 协议调试 29
5.4 HTTP 协议调试 31
6 总结与体会 32
7 谢辞 33
8 参考文献 34
附录： 35
附录一：系统原理图 35
附录二：系统硬件实物图 36
附录三：核心代码 37
环形缓冲区的串口驱动： 37
RTL8019AS 驱动程序： 39
UIP_APPCALL 代码： 43
uIP 协议栈框架代码 48
附录四:外文翻译 50

Frank Luo long.luoeoutlook.com

1 前言

随着互联网络硬件和软件的迅猛发展，网络用户呈指数增长，在使用计算机进行网络互联的同时，各种家电设备、仪器仪表以及工业生产中的数据采集与控制设备在逐步走向网络化，以共享网络中庞大的信息资源。

目前基于网络应用的设备通常都是使用 PC 机或 ARM 等 32 位微处理器作为硬件，以 windows、linux 等作为系统软件，这种结构开发周期短，功能强、数据传输速高，但是也存在成本高、体积大、功耗大缺点。

针对以上的缺点，提出了以低成本的单片机和网络控制器为主要硬件构建一个接入 Internet 的低成本嵌入式平台的需求。

本平台主要由 MCS51 系列单片机 STC89C58RD+、以太网控制器 RTL8019AS、通讯模块、电源模块等硬件部分组成；可以有效的降低硬件开发平台的成本。软件部分主要由 uIP0.9 嵌入式 TCP/IP 协议栈、HTTP 解析程序等部分组成，并采用了 C 语言编程，使系统软件易维护、易升级。同时该平台还具有 RS232 转以太网接口的功能。

为方便大家的使用，本次设计中给出了该平台的简单运用实例，实现了温度的网络监测和开关量的网络控制功能。

。

Frank Luo long.luo@outlook.com

2 总体方案设计

2.1 方案设计

2.1.1 方案一

采用带硬件协议栈的网络芯片+高新能的 ARM 处理器来实现

该方案采用带硬件协议栈的网络芯片(比如 w3100a)和高性能的 ARM 处理器作为系统的核心器件。其中网络芯片主要功能是完成 TCP/IP 各层协议的处理和总线数据接口到 RJ45 网络接口的电气转换，同时还具有数据缓冲的作用；ARM 处理器主要完成应用层协议的处理、数据采集以及控制的执行等。其框图如图 2.1。

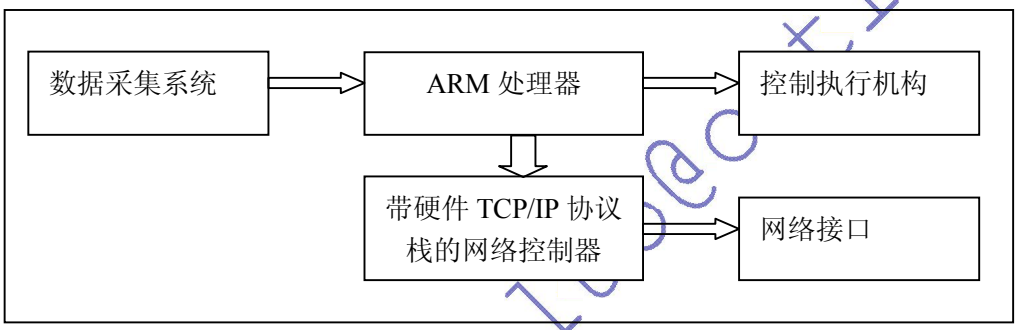


图 2.1 方案一系统框图

2.1.2 方案二

采用普通的网络芯片+单片机来实现

该系统采用普通的单片机作为核心控制器件，它不仅要实现应用层，还要在软件上实现 TCP/IP 协议栈，同时还要进行数据的采集和控制命令的发送；而网络芯片只负责数据链路层（DLC）和物理层（MAC）的实现。其框图如图 2.2 所示。

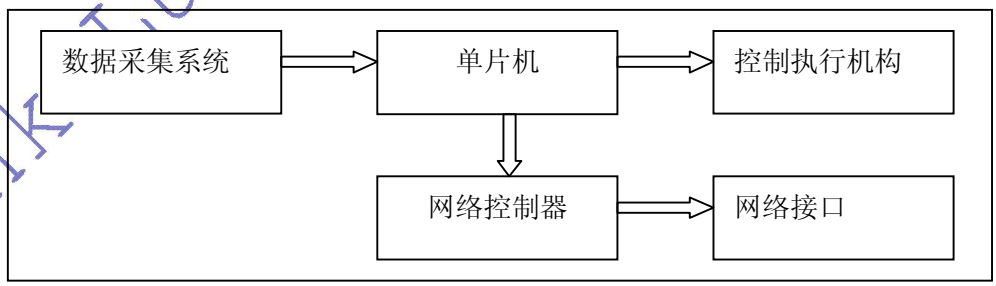


图 2.2 方案二系统框图

Frank Tuo Tong 1006@163.com

2.2 方案论证

方案一采用了带硬件 TCP/IP 协议栈的网络芯片，与软件协议栈相比，大大减轻了处理器的负担；采用了高性能的 ARM 处理器进行资料的采集和控制命令的发送，通讯速度可以大大提高，可以在很苛刻的场合下使用，若采用实时操作系统，还可以满足实时系统的要求。但该方案成本高，开发相对比较困难，由于成本的原因限制了其应用范围。

方案二采用通用了单片机和通用的网络芯片，虽然普通单片机速度上比不上 ARM，采用了软件协议栈，增加了处理器的负担，但是，该方案所用器件成本低廉，开发简单，应用范围广泛，在工控场合可以替代以前的 232、485、CAN 等总线。

2.3 方案选择

综上所述，考虑到系统成本、开发的复杂以及应用场合等问题，选择方案二。该方案具有稳定性高、低成本，易开发，应用广泛的特点。其系统总体框图如图 2.1 所示。

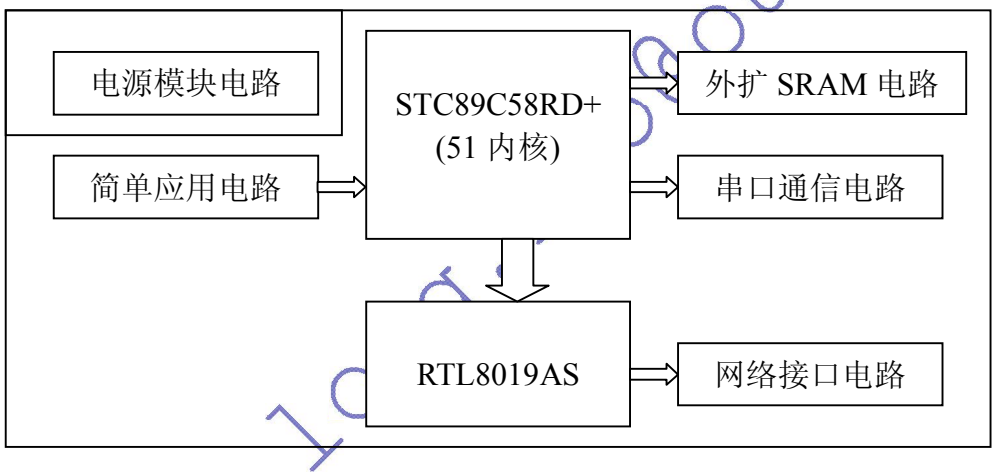


图 2.1 系统总体框图

Frank Luo luo_lo@outlook.com

3 单元模块设计

3.1 系统功能模块设计

该系统主要由电源模块、微处理器、网络控制芯片、外扩 SRAM、串口以及简单应用电路构成。其中，微处理器负责 TCP/IP 协议栈的应用层协议的实现；外扩 SRAM 作为系统的数据缓冲区；网络控制芯片实现 TCP/IP 的底层协议（DLC、MAC）；串口电路可以使该系统成为一个 RJ45 转 RS232 的转换器；简单应用电路为该监控 Web 服务器的简单监视控制应用实例。

3.1.1 电源模块设计

该系统所用的器件电压等级比较统一，都采用 5V 供电，并且功耗也不大，故电源模块很简单，只要能提供稳定的 5V 电压就满足要求了。其电路图如图 3.2 所示。

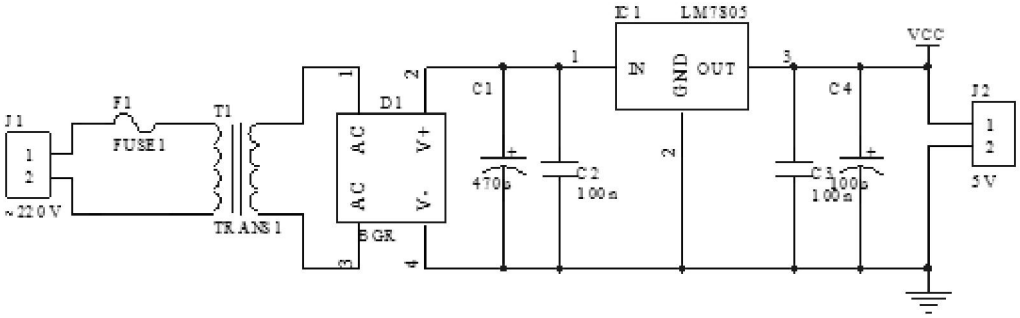


图 3.1 电源模块电路

3.1.2 单片机最小系统

一个单片机需要加上一定的外部电路组成最小系统才能工作。最小系统一般包括复位电路、时钟电路、下载接口和供电电路。该系统的单片机最小系统如图 3.2 所示。

Frank Luo 1ong@hotmail.com

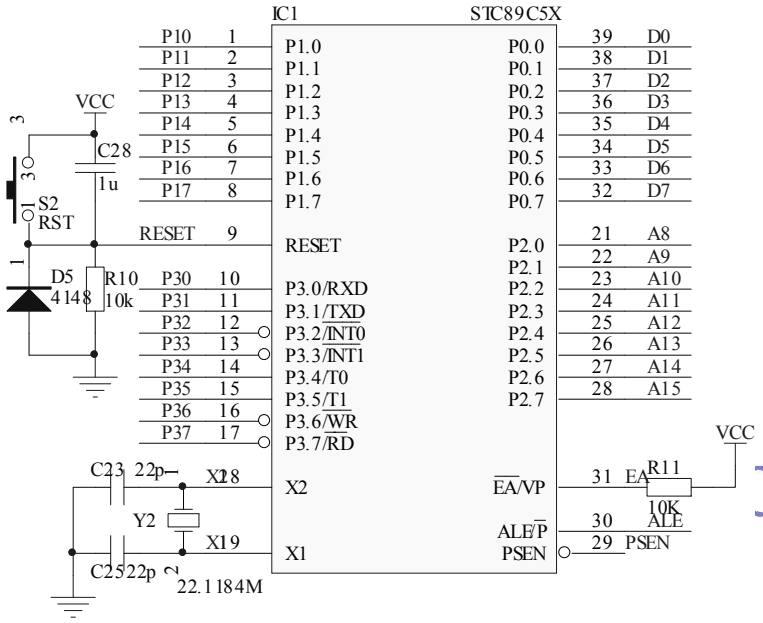


图 3.2 单片机最小系统

3.1.3 串口通信电路设计

在该系统中，串口通信电路可以进行系统功能的扩展（即作为 RJ45 转 RS232 的转换器），同时，也是更新单片机程序的接口，因为 STC89C58RD+采用串口下载。其电路图与一般的串口电平转换电路没什么区别，如图 3.3 所示。

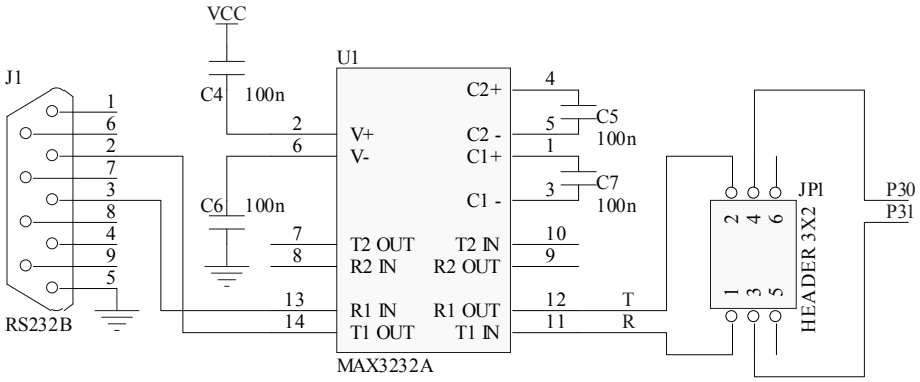


图 3.3 串口通信电路

3.1.4 SRAM 外扩电路设计

由于 STC89C58RD+自身带有 1K 的 SRAM，对于小系统来说是比较充裕的，通过将 IP 数据包配置进行优化，该系统也可以不用外扩 SRAM，但这样系统性能会大打折扣。为了系统综合性能和以后的升级扩展，采用总线方式外扩了 32K 的 SRAM，不仅可以配置 IP 包长为最大值，串口缓冲区也可以配置得比较大，而且可以运行简单的实时操作系统，并且成本也没增加多少。其电路图如图 3.4。

Frank

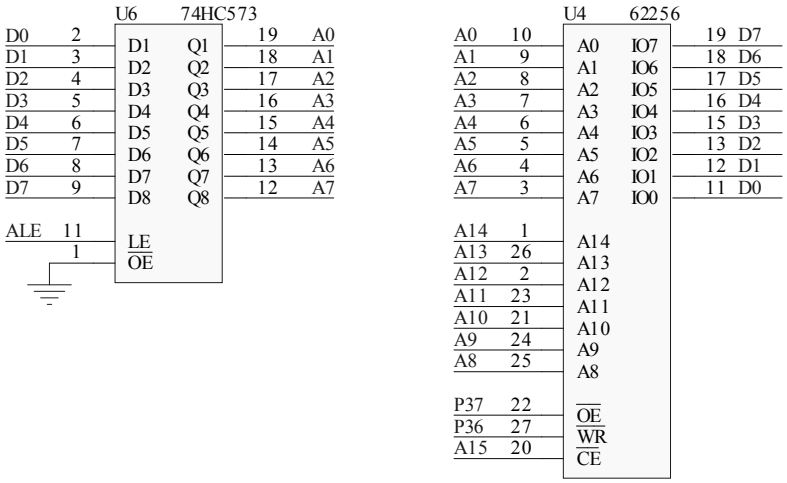


图 3.4 外扩 SRAM 电路

3.1.5 RTL8019AS 网络接口电路设计

该部分电路设计是该平台的关键部分，必须熟悉 RTL8019AS 的工作原理，才能很好进行该部分电路的设计。

RTL8019AS 工作方式的选择:

RTL8019AS 支持 3 种工作方式。

1) 即插即用方式：I/O 位元址和中断都由操作系统管理,用户不必过多干预,当然这种方式要耗费系统资源多一些;

2) 跳线方式：网卡的 I/O 地址和中断都由跳线决定;

3) 免跳线方式：网卡的 I/O 和中断由外接的 E2PROM 93C46 中的内容决定

在这三种方式里，本系统使用第二种跳线工作方式（即将 JP 接 VCC），第一种方式对于 51 系列单片机来说无法实现，未用第三种方式又可以省掉一片 93C46 芯片，不但减少了连接，而且降低了成本。

数据线位宽的选择:

由于 51 单片机的总线位宽是 8 位，通过一个 27K 电阻将 IOCS16B 引脚下拉到 GND，使 RTL8019AS 工作在 8 位位宽模式，将单片机 P0 口与 SD0～SD7 相连，这样就完成了数据线的连接。

寄存器基地址的配置:

RTL8019AS 的寄存器和 NE2000 网卡兼容，采用分页方式访问 4 页寄存器，占用地址空间 0x00～0x1F。采用 5 位地址线 A0～A4 就可以进行访问。但还需要进行 IO 位元址的配置，IO 位元址由引脚 IOS3～IOS0 决定。RTL8019AS 的引脚悬空时，输入状态为低电平,因为芯片引脚内部已经接了一个 100k 的下拉电阻，故默认情况下基址为 0x300。当 SA8、SA9=11 时才选择 RTL8019AS 内部寄存器。由于系统外扩了 32K 的 SRAM，地址范围

Frank.luo.long.luo@outlook.com

为 0x0000～0x7FFF，用 A15 接 SA8 和 SA9，这样 RTL8019AS 寄存器地址范围为 0x8000～0x801F。

为方便操作，将 RTL8019AS 寄存器定义为：

```
#define Reg00 (*(unsigned char volatile xdata *)0x8000)
#define Reg01 (*(unsigned char volatile xdata *)0x8001)
……
#define Reg1f (*(unsigned char volatile xdata *)0x801f)
```

完整的网络接口电路图如图 3.5 所示。

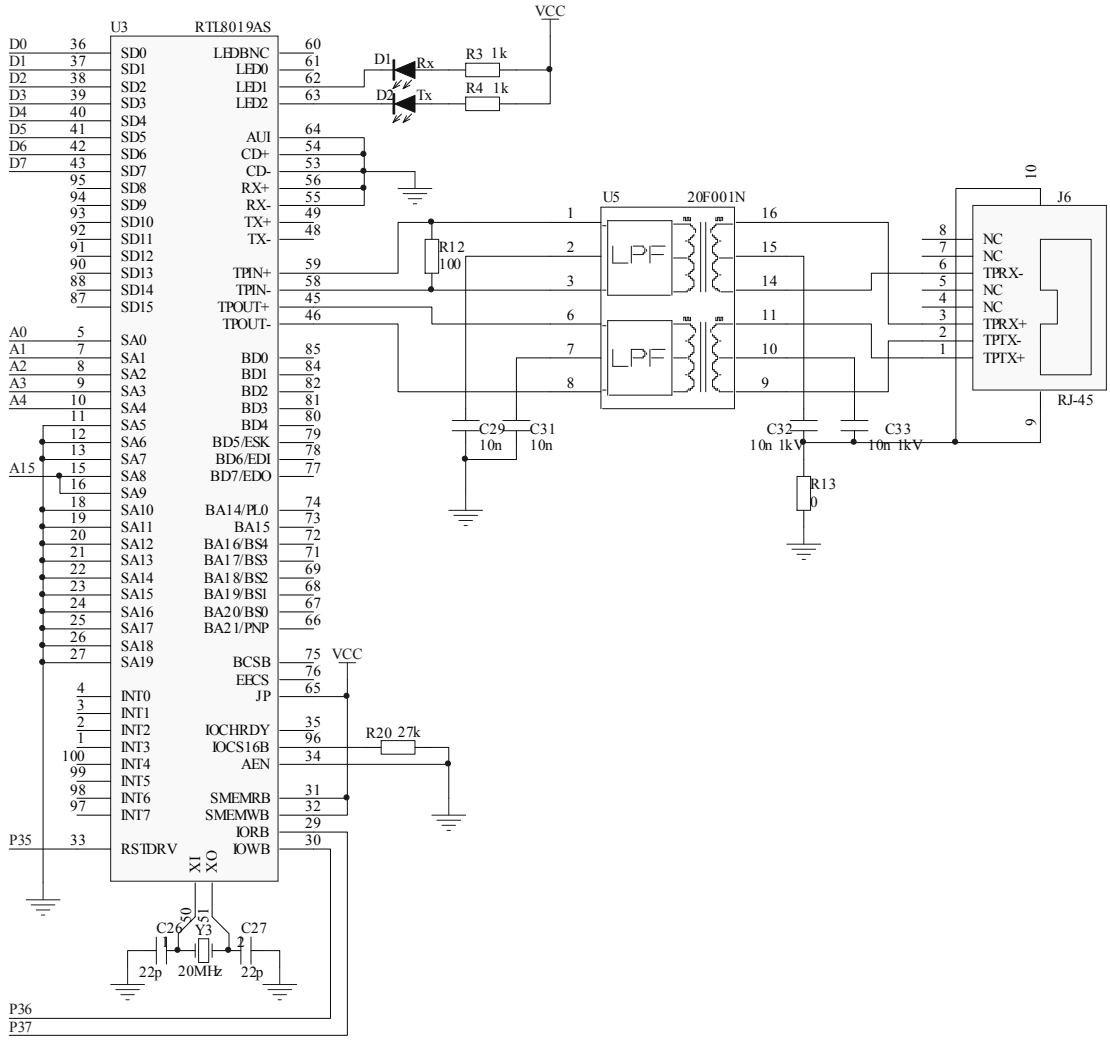


图 3.5 网络接口电路

3.1.6 简单应用电路

该电路作为监控系统的简单应用。其中 DS18B20 用于对环境温度的监视，其温度值通过以太网传输到 PC，通过网页进行显示。控制电路用两个发光二极管代替，在网页上

控制其亮与灭，即开关量的控制。由于其电路过于简单，这里就不给出其电路了。

3.2 特殊器件介绍

该系统采用的器件比较多，但大多比较常用，这里主要介绍和传统 51 单片机有些差别的 STC89C58RD+和网卡芯片 RTL8019AS。

3.2.1 STC89C58RD+

由于该系统软件代码比较大，就单独 uip0.9 协议栈就要 8K，再加上应用层协议和网页文件等，普通的 51 单片机不能满足要求。故选择 STC89C58RD+，该单片机是宏晶公司生产的一款与 Intel 8051 管脚和指令兼容的高性能单片机，拥有 32K 的程序存储空间和 1K 的 SRAM，并且可以采用 6 时钟每机器周期，可以使处理速度快一倍。该单片机还有如下特点：

★ 加密性强,无法解密
★ 超低功耗
★ 超强抗干扰
★ 在系统可编程,无需编程器,可远程升级
★ 降低单片机时钟对外部电磁辐射
★ 如选 6 时钟/ 机器周期，外部时钟频率可降一半
★ 单片机内部自带看门狗电路

3.2.2 RTL8019AS

网络接口采用了台湾 REALTEK 公司生产的一种 NE2000 兼容的 ISA 总线以太网控制芯片，虽然 ISA 接口在 PC 上消失，但该芯片还在生产，货源充足，价格低廉，应用十分广泛。其主要特点如下

- 该芯片符合 IEEE 802.3 10Base2 和 10BaseT 标准
- 网络传输速率为 10Mbit/sec，支持 CSMA/CD 传输协议
- 自动奇偶检测及纠错
- 支持即插即用 PnP， 非即插即用 Non-PnP；可通过软件进行设置并能够设置中断入口输出地址等网络参数
- 支持两种接口类型并能自动侦测介质类型。BNC 口用于连接同轴电缆，RJ45 口用于非屏蔽双绞线
- 支持全双工模式

Frank.Luo long.luo@outlook.com

4 软件设计

4.1 TCP/IP 协议族

对于嵌入式系统，尤其是像 51 这样的 8 位的 MCU，其资源少，要运行完整的 TCP/IP 是不行的，只能运行精简的 TCP/IP 协议栈。这样，在兼容完整的 TCP/IP 的同时大大降低了产品的成本。由于篇幅的原因，这里只对使用的部分做简单介绍。

4.1.1 TCP/IP 协议简介

TCP/IP 是发展至今最成功最广泛的通信协议。Internet 就是其成功的见证。该协议有两个模型（OSI 模型和 TCP/IP 参考模型），其中 OSI 模型将 TCP/IP 分为 7 层，为应用层、表示层、会话层、传输层、网络层、数据链路层和物理层；而 TCP/IP 参考模型分为应用层、运输层、网络层和链路层。图 4.1 表示了各协议所处的层次。

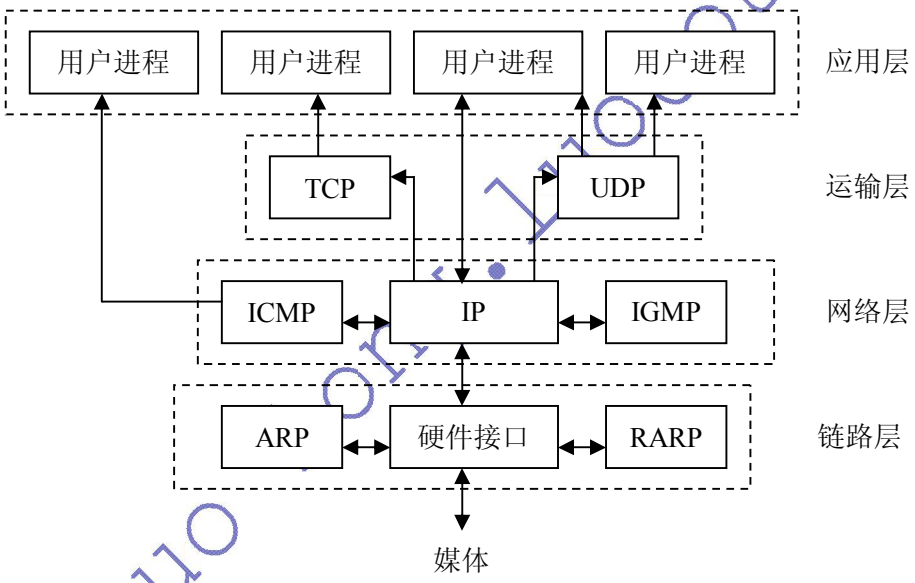


图 4.1 TCP/IP 参考模型中不同协议所的层次

4.1.2 数据包封装

TCP/IP 协议可同时支持 IEEE802 封装和以太网封装。IEEE802 封装的数据包长度字段和以太网的协议类型字段在同一位置，但由于数据包长度最大为 1500 字节，而协议类型从 0x0800 开始，故可以分辨两种封装。由于单片机资源的限制，没有必要同时支持两种协议，这里使用的是以太网封装。以太网封装的数据包如图 4.2 所示。

|----以太网封闭（RFC894）----|

目的地址	源地址	类型	数据	CRC
6B	6B	2B	46B~1500B	4B

图 4.2 数据包封装

Frank_Luo_1@outlook.com

4.1.3 数据帧的分用

当目的主机收到一个以太网数据帧时，数据就开始从协议栈中由底向上传送，同时去掉各层协议加上的报文首部。每层协议盒都要去检查报文首部中的协议标识，以确定接收数据的上层协议。这个过程称作分用。其分用过程如图 4.3 所示。

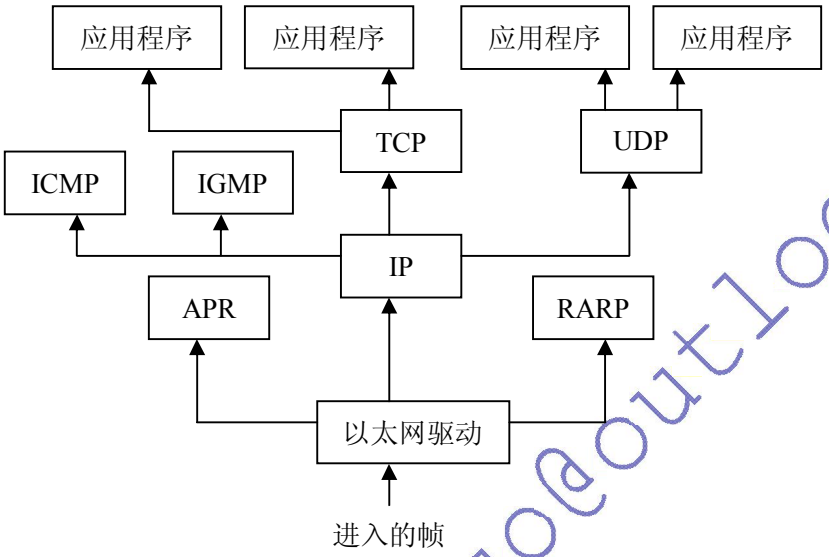


图 4.3 以太网数据帧的分用过程

4.1.4 地址解析协议 ARP

说到 ARP，一般就会想到物理地址（MAC）地址。在网络底层的物理传输过程中，是通过物理地址来识别主机的，它一般也是全球唯一的，由以太网地址管理机构（IEEE）将以太网地址，分配给各网卡生产商。但也不是绝对的。因为 MAC 在局域网内作为网络接口地址，而越过局域网后使用的是 IP，只要在局域网内 MAC 地址不重合，一般不会出问题。虽然在该系统中不会涉及到多播与组播，但在配置 MAC 时也要遵循一些规则。其中全为 1 的地址为多播，最高位为基数的地址为多播地址，合法的 MAC 地址做好位为偶数。

ARP 数据包内容共有 28 字节，加上以太网的首部 14 字节，总共 42 字节，因为最小的以太网数据帧的大小为 60 字节，因此 ARP 数据包都需要再添加任意的数据以补足 60 字节。ARP 数据包格式如图 4.4 所示。

以太网目的地址	以太网源地址	帧类型	硬件类型	协议类型	硬件地址长度	协议地址长度	OP	发送端 MAC 地址	发送端 IP 地址	目的 MAC 地址	目的 IP 地址
6B	6B	2B	2B	2B	1B	1B	2B	6B	4B	6B	4B

图 4.4 ARP 数据包格式

Frank Luo Long.luo@outlook.com

- 其中帧类型为 0x0806，代表为 ARP 协议
- 硬件类型为 0x0x0001 代表以太网
- 协议类型为 0x0800 表示使用的是 IP 协议
- 硬件地址长度为 6，表示使用的是 48 位 MAC 地址
- 协议地址长度为 4 表示采用的是 IPV4
- OP 可为 1（ARP 请求）和 2（ARP 应答）

在发送一个 ARP 请求时，目的 MAC 为全 1，由接收方填充自己的 MAC 地址

4.1.5 网际协议 IP

IP 是 TCP/IP 协议族中最为核心的协议，目前的版本是 IPV4。所有的 TCP、UDP、ICMP、IGMP 等数据都以 IP 数据包格式传输，IP 提供不可靠的、无连接的数据传送服务。其数据报的格式如图 4.5 所示。

<table>
<tr><td>4 位版本</td><td>4 位首部长度</td><td>8 位服务类型(TOS)</td><td colspan="2">16 位总长度（字节数）</td></tr>
<tr><td colspan="3">16 位标识</td><td>3 位标志</td><td>13 位片偏移</td></tr>
<tr><td colspan="2">8 位生存时间(TTL)</td><td>8 位协议</td><td colspan="2">16 位首部校验和</td></tr>
<tr><td colspan="5">32 位源 IP 地址</td></tr>
<tr><td colspan="5">32 位目的 IP 地址</td></tr>
<tr><td colspan="5">选项（如果有）</td></tr>
<tr><td colspan="5">数据</td></tr>
</table>

图 4.5 IP 数据报文格式

- 协议版本为 4
- 首部长度指首部占 32 bit 字的数目
- 总长度以字节为单位表示整个 IP 数据包长度
- 标识是数据包的 ID 号用于识别不同的 IP 数据包
- 生存时间 TTL 字段设置了数据包可以经过的最多路由器数，一般每经过路由器时减一，因此，通过 TTL 可以算出数据包到达目的地所经过的路由器个数
- 协议字段表示创建该数据包的高级协议类型 IP 用该协议对数据进行分用，如 1 表示 ICMP 协议，6 表示 TCP 协议，17 表示 UDP 协议等
- 另外，我们用单片机实现 TCP/IP 协议时作了一些简化，不考虑数据分片和优先权因此，在此我们不讨论服务类型、标志以及片偏移，只需填 0 即可。

4.1.6 控制报文协议 ICMP

ICMP 经常被认为是 IP 层的一个组成部分，它传递差错报文以及其他需要注意的信

息。ICMP 报文通常被 IP 或更高层协议（TCP 或 UDP）使用。由于该系统采用的是简化的协议栈，没有支持完整的 ICMP，只是实现了常见的回应请求与应答报文，即可以通过 PING 命令进行可到达性的判断。ICMP 报文格式如图 4.6 所示。

8 位类型（0 或 8）	8 位代码（0）	16 位校验和
标识符		序号
选项数据		

图 4.6 ICMP 报文格式

- 类型 0 为回应应答报文，8 为回应请求报文
- 整个数据包均参与检验

4.1.7 控制传输协议 TCP

数据传输一般采用 UDP 和 TCP 协议，由于 UDP 是一个简单的面向数据报的运输层协议，它不提供可靠性的服务，只是把应用程序传给 IP 层的数据发送出去，而不能保证它们能到达目的地。而 TCP 提供面向连接可靠的字节流服务，面向连接指的是两个 TCP 的应用（其中一个是客户机，一个是服务器）。在彼此交换数据之前必须先建立一个 TCP 连接（即 3 次握手），并且还添加了超时重传和确认序号机制来保证数据传输的可靠性。因此，该系统没有添加 UDP 的支持，这里主要介绍 TCP，TCP 数据包格式如图 4.7 所示。

16 位源端口								16 位目的端口
32 位序号								
32 位确认号								
4 位首部长度	保留（6 位）	URG	ACK	PSH	RST	SYN	FIN	16 位窗口指针
16 位校验和								16 位紧急指针
选项（如果有）								
数据								

图 4.7 TCP 数据报格式

每个 TCP 段都包含源端和目的端的端口号，用于寻找发端和收端应用进程。这两个值加上 IP 首部中的源端 IP 地址和目的端 IP 地址唯一确定一个 TCP 连接。

有时，一个 IP 地址和一个端口号也称为一个插口（socket）。插口对(socket pair)(包含客户 IP 地址、客户端口号、服务器 IP 地址和服务器端口号的四元组)可唯一确定互联网络中每个 TCP 连接的双方。

序号用来标识从 TCP 发端向 TCP 收端发送的数据字节流，它表示在这个报文段中的的第一个数据字节。如果将字节流看作在两个应用程序间的单向流动，则 TCP 用序号对每个字节进行计数。序号是 32 bit 的无符号数，序号到达 2^32－1 后又从 0 开始。

当建立一个新的连接时，SYN 标志变 1。序号字段包含由这个主机选择的该连接的

Frank Liu 1006@outlook.com

初始序号 ISN（Initial Sequence Number） 。该主机要发送数据的第一个字节序号为这个 ISN 加 1，因为 S Y N 标志消耗了一个序号（将在下章详细介绍如何建立和终止连接，届时我们将看到 FIN 标志也要占用一个序号）。

既然每个传输的字节都被计数，确认序号包含发送确认的一端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加 1。只有 ACK 标志（下面介绍）为 1 时确认序号字段才有效。发送 ACK 无需任何代价，因为 32 bit 的确认序号字段和 ACK 标志一样，总是 TCP 首部的一部分。因此，我们看到一旦一个连接建立起来，这个字段总是被设置，ACK 标志也总是被设置为 1。

TCP 为应用层提供全双工服务。这意味数据能在两个方向上独立地进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

4.2 uIP 协议栈

现在网络上有很多 TCP/IP 协议栈，要么代码量很大，要么需要实时操作系统的支持，有的需要付费才能使用，有的又没有开源，适合 51 这样的 8 位处理器的还不多。

uIP 协议栈是一种免费的可实现的极小的 TCP/IP 协议栈，可以用于由 8 位或 16 位微处理器构建的嵌入式系统。它可以自由分发和使用于商业和非商业目的。uIP 使用 C 语言编写，使其方便移植。并且 uIP 协议栈的代码大小和 RAM 的需求比其它一般的 TCP/IP 栈要小，这就使得它可以方便的应用到各种低端系统上。所以该系统选择 uIP0.9 协议栈。下面对其进行简要介绍。

4.2.1 uIP 协议栈的实现方法

uIP 实现了 TCP/IP 协议栈的四个基本协议：ARP 地址解析协议，IP 网际互联协议，ICMP 网络控制报文协议和 TCP 传输控制协议。为了在 8 位 16 位处理器上应用，uIP 协议栈在各层协议实现时采用有针对性的方法，保持代码大小和存储器使用量最小。

1）实现 ARP 地址解析协议时为了节省存储器，ARP 应答包直接覆盖 ARP 请求包。

2）实现 IP 网络协议时对原协议进行了极大的简化，它没有实现分片和重组。

3）实现 ICMP 网络控制报文协议时，只实现 echo（回响）服务。uIP 在生成回响报文时并不重新分配存储器空间，而是直接修改 echo 请求报文来生成回响报文。将 ICMP 类型字段从“echo”类型改变成“echoreply”类型，重新计算校验和并修改校验和字段。

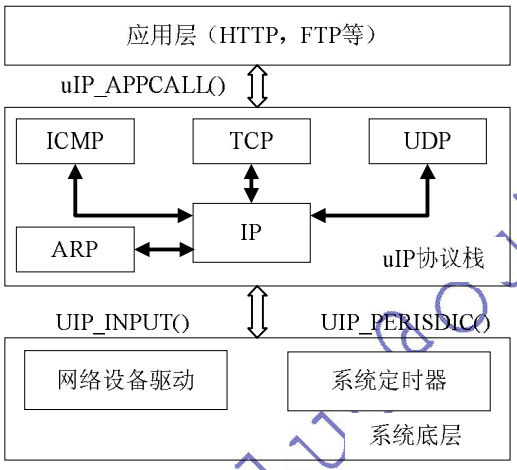
4）uIP 里的 TCP 没有实现发送和接收数据的滑动窗口。每个 TCP 连接的状态由 uip_conn 结构保存，uip_conn 结构包括当地和远端的 TCP 端口编号，远程主机的 IP 地址，重发时间值，上一段重发的编号，和连接的段的最大尺寸等信息。一个 uip_conn 结构数组用于保存所有的连接，数组的大小为支持的同时连接的最大数量。为了减少储存器的使用量，在处理重发时 uIP 并不缓存发送的数据包，而是由应用程序在需要重发

Frank_Liu_long.hotmail.com

时重新生成发送的数据。

4.2.2 uIP 协议栈的接口

uIP 协议栈为了具有最大的通用性，在实现时将底层硬件驱动和顶层应用层之外的所有协议集“打包”在一个“库”里。协议栈通过接口与底层硬件和顶层应用“通信”。通过这种方式，uIP 具有极高的通用性和独立性，移植到不同系统和实现不同的应用都很方便，很好的体现了 TCP/IP 协议平台无关性的特点。uIP 协议栈与系统底层和应用程序之间的接口关系如图 4.8 所示：



4.8 uIP 协议栈的接口关系

1）uIP 协议栈与系统底层的接口

uIP 与系统底层的接口包括与设备驱动的接口和与系统定时器的接口两类。

1.1）uIP 与设备驱动接口

uIP 通过函数 uip_input（）和全局变量 uip_buf、uip_len 来实现与设备驱动的接口。uip_buf 用于存放接收到的和要发送的数据包，为了减少存储器的使用，接收数据包和发送数据包使用相同的缓冲区。uip_len 表明接收发送缓冲区里的数据长度，通过判断 uip_len 的值是否为 0 来判断是否接收到新的数据，是否有数据要发送。当设备驱动接收到一个 IP 包并放到输入包缓存里（uip_buf）之后，应该调用 uip_input（）函数。uip_input（）函数是 uIP 协议栈的底层入口，由它处理收到的 IP 包。当 uip_input（）返回，若有数据要发送，则发送数据包放在包缓冲区里。包的大小由全局变量 uip_len 指明。如果 uip_len 是 0，没有包要发送；如果 uip_len 大于 0 则调用网络设备驱动发送数据包。

1.2）uIP 与系统计时接口

TCP/IP 协议要处理许多定时事件，例如包重发、ARP 表项更新。系统计时用于为所有 uIP 内部时钟事件计时。当周期计时激发，每一个 TCP 连接应该调用 uIP 函数 uip_periodic（）。TCP 连接编号作为参数传递给 uip_periodic（）函数。uip_periodic

Frank.Luo.long.luo@outlook.com

（）函数检查参数指定的连接的状态，如果需要重发则将重发数据放到包缓冲区（uip_buf）中并修改 uip_len 的值。当 uip_periodic（）函数返回后，应该检查 uip_len 的值，若不为 0，则将 uip_buf 缓冲区中的数据包发送到到网络上。

ARP 协议对于构建在以太网上的 TCP/IP 协议是必须的，但对于构建与其他网络接口（例如：串行链路）上的 TCP/IP 则不是必需的。为了结构化的目的，uIP 将 ARP 协议作为一个可添加的模块单独实现。因此，ARP 表项的定时更新要单独处理。系统定时器对 ARP 表的更新进行定时，定时时间到则调用 uip_arp_timer（）函数对过期表项进行清除。

2）uIP 协议栈与应用程序的接口

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。用户需将应用层入口程序作为接口提供给 uIP 协议栈，定义为宏 UIP_APPCALL（）。uIP 在接收到底层传来的数据包后，若需要送上层应用程序处理，它就调用 UIP_APPCALL（）。uIP 提供给应用程序的接口函数按功能描述如下：

2.1）接收数据接口：应用程序利用 uip_newdata（）函数检测是否有新数据到达。全局变量 uip_appdata 指针指向实际数据。数据的大小通过 uip_datalen（）函数获得。

2.2）发送数据接口：应用程序通过使用 uIP 函数 uip_send（）发送数据。uip_send（）函数采用两个参数；一个指针指向发送数据起始地址，另一个指明数据的长度。

2.3）重发数据接口：应用程序通过测试函数 uip_rexmit（）来判断是否需要重发数据，如果需要重发则调用 uip_send（）函数重发数据包。

2.4）关闭连接接口：应用程序通过调用 uip_close（）函数关闭当前连接。

2.5）报告错误接口：uIP 提供错误报告函数检测连接中出现的错误。应用程序可以使用两个测试函数 uip_aborted（）和 uip_timedout（）去测试那些错误情况。

2.6）轮询接口：当连接空闲时，uIP 会周期性地轮询应用程序，判断是否有数据要发送。应用程序使用测试函数 uip_poll（）去检查它是否被轮询过。

2.7）监听端口接口：uIP 维持一个监听知名 TCP 端口的列表。通过 uip_listen（）函数，一个新的监听端口打开并添加到监听列表中。当在一个监听端口上接收到一个新的连接请求时，uIP 产生一个新的连接和调用该端口对应的应用程序。

2.8）打开连接接口：在 uIP 里面通过使用 uip_connect（）函数打开一个新连接。这个函数打开一个新连接到指定的 IP 地址和端口，返回一个新连接的指针到 uip_conn 结构。如果没有空余的连接槽，函数返回空值。

2.9）数据流控制接口：uIP 提供函数 uip_stop（）和 uip_restart（）用于 TCP 连接的数据流控制。应用程序可以通过函数 uip_stop（）停止远程主机发送数据。当应用程序准备好接收更多数据，调用函数 uip_restart（）通知远程终端再次发送数据。函数 uip_stopped（）可以用于检查当前连接是否停止。

Frank_Huo_Long@hotmail.com

4.2.3 uIP 在 51 系列单片机上的应用

uIP 协议栈是以函数库的形式提供的，本身不提供底层网络驱动和上层应用程序。因此为了完成指定的功能，开发者必须添加以下几个模块：底层 RTL8019AS 网卡芯片的驱动、应用层基于 HTTP 协议的 WEB SERVER 的实现、系统定时器。

RTL8019AS 的驱动主要包括三部分：etherdev_init（）函数完成网卡芯片的上电初始化，包括设定网卡物理地址，设定收发缓冲区位置和大小等；etherdev_send（）函数完成数据的发送；etherdev_poll（）函数完成以太网数据的接收。底层网络设备驱动程序与 uIP 协议栈通过两个全局变量进行接口：变量 uip_buf 为收发缓冲区的首地址；uip_len 为收发的数据长度。eth_send（）函数将 uip_buf 里的 uip_len 长度的数据发送到以太网上。eth_rcve（）函数将接收到的数据存储到 uip_buf 指定的缓冲区中，同时修改 uip_len 的值。

uIP 提供的源代码中包括一个基于 HTTP 协议的 WEB SERVER 示例，该 WEB SERVER 通过简单的文件系统在数据存储器中存储静态页面，同时具有 CGI 功能。但在实际使用时，发现不是很稳定，并且对 CGI 的处理也不是很科学（每一个脚本行都要发送一个数据包，导致速度很慢，同时也变得不稳定）。该系统仅使用了 uIP 协议栈，而没有使用其 HTTP 部分。HTTP 的实现很简单，在很多场合下只要服务器响应客户机的 GET 请求即可，对于需要动态改变的网页，可以用缓冲区来存网页文件，再对其修改由一起发出，这样虽然增加了对 SRAM 的需求量，但换来了速度和稳定，由于系统外扩了 32K 的 SRAM，这点是可以满足的。

51 系列单片机上都有 2 到 3 个定时计数器，可以选择其中的一个来为 TCP/IP 协议中与时间有关的事件定时。需要由用户处理的定时事件包括：为 uip_periodic（）函数的执行提供基准，还要为 ARP 表项的更新定时。uip_periodic（）函数每 0.5 秒执行一次，ARP 表项每 10 秒更新一次。

uIP 的设置单独包含在一个叫 uipopt.h 的头文件里，都是以宏的形式定义方便于修改。用户应根据自己的应用在 uipopt.h 文件里设置本地的物理地址、IP 地址、网关地址、收发缓冲区的大小、支持的最大连接数、ARP 表大小等选项。

添加了必须的模块，对 uIP 进行了正确地配置后，需要编写主程序函数。针对该系统，主程序在完成初始化后将不停的进行查询，如果有新数据包到达则送 uip_input（）函数处理；如果没有新数据包到达则处理定时事件。框架代码见附录的核心代码。

4.2.4 uIP 底层接口

在前面的章节曾提到，使用 uIP 协议栈，要用户添加网卡的底层驱动，该驱动必须按照 uIP 的要求编写。其中需要编写的接口函数为网卡初始化函数为 etherdev_init()、数据包查询函数 etherdev_poll()和数据包发送函数 etherdev_send()。在介绍网卡驱

Frank_Luo_Long.Hotmail.com

动之前，首先介绍一下 RTL8019AS 的内部结构。

1）RTL8019AS 内部 RAM 分布

RTL8019AS 有两块 RAM，一块 16k 字节的 RAM，地址为 0x4000-0x7FFF，该地址为芯片内的存储地址，网卡芯片用它来存储转发数据；一块 32 字节 RAM，地址为 0x0000-0x001F，芯片内部地址空间的分布如图 4.15 所示 。

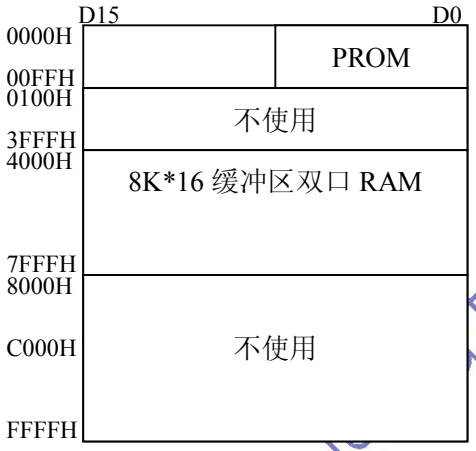


图 4.15 RTL8019AS 内部 RAM 分布

由图 4.15 可见,可用的 RAM 空间仅有上面提到的两部分。从 0x00-0xFF 的内存为 Prom，是 Ne2000 兼容的网卡都有的。实际上，与其它地址如 0x0020-0x00ff 存储的内容是重复的。其中 0x00-0x0B(工作于 8 位 DMA 模式)用于存放本节点 MAC 地址，奇偶地址内容是重复放置的，如 MAC 地址 52544C19F823，存在 0x00-0x0B 中为 525254544C4C1919F8F82323，单地址和双地址内容是重复的，但为了同时适应 8 位和 16 位的 DMA 操作，一般使用偶数地址的内容。

Prom 只保存上电时从 93C46 中读出的 MAC 地址(如果有 93C46 的话)。由于我们的设计中没有使用网卡和 93C46，因此也可以不使用从地址 0x0000-0x0001F 的 32 字节的 Prom。而物理地址不是由 Prom 而是 PAR0-PAR5 寄存器决定的 因此不使用 93C46 时可在程序中指定物理地址。

地址为 0x4000-0x7FFF 的 16K 存储区是分页的,每 256 个字节称为一页,共有 64 页，程序设计中使用这块 RAM 存储接收或发送的数据包。由于 RTL8019AS 接收或发送的数据是按页存储的，如果某页没有完全填满数据，则下包数据也不能继续使用该页，只能使用新的页，并且数据是按页连续存放的，芯片页地址 0x40--0x7f 代表的 RAM 空间是 0x4000-0x7fff。另外这 16K 的 RAM 可由用户进行配置,一部分用来存放接收的数据包 ，一部分用来存储待发送的数据包 。

2）RTL8019AS 的本地和远程 DMA

在 RTL8019AS 中地址为 0x4000-0x7FFF 的 16K 的存储器是双端口的，即有两套读写

Frank_Luo_long.hotmail.com

控制总线。对该 RAM 的读写采用的是 DMA 操作方式，这样做的好处是加快了访问速度。其中，与 CPU 侧相连的端口称为“远程 DMA”，与 8019 收发电路侧连接的端口称为“本地 DMA”。芯片内部有一些总线仲裁逻辑，负责两套总线的管理，使它们能同时对 RAM 读写而不产生冲突。

RTL8019AS 的本地 DMA 的操作是由控制器本身完成的，而远程 DMA 需要主机 CPU 给出起址和数据长度就可以读写芯片 RAM，每操作一次 RAM 地址自动加 1。这比普通 RAM 操作（每次要先发地址再处理数据）要快很多。发送缓冲区、接收 FIFO 缓冲区队列和远程 DMA 操作的示意图如图 4.16 所示。

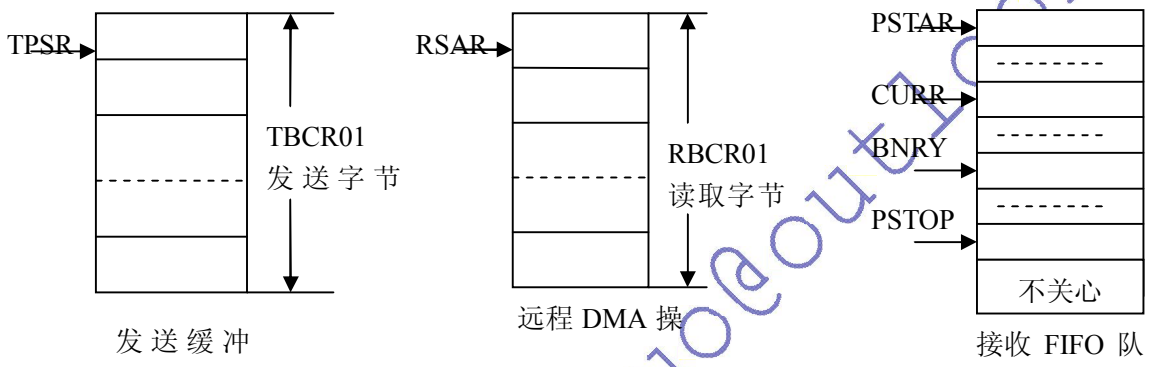


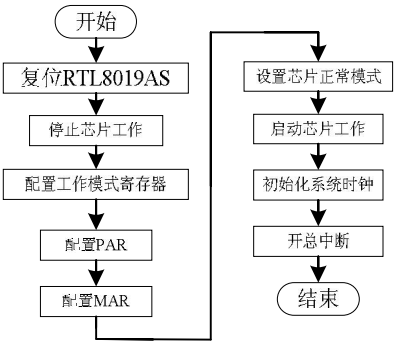
图 4.16 发送缓冲区、接收 FIFO 和远程 DMA 操作示意图

在 4.1.2 节谈到，该系统使用的是以太网数据包，但 RTL8019AS 收到的数据包并不是以太网数据帧的真子集，8019 在接到完整的一帧数据后，自动添加了以下信息：接收状态、下一页指针和以太网帧长度（共 4 字节）。这些数据成员的引入方便了驱动程序的设计，体现了软硬件互相配合协同工作的设计思路。

有了前面的介绍，接下来介绍 uIP 的接口驱动程序就变得简单了。

3) 网卡芯片初始化函数 etherdev_init()

卡芯片的初始化主要包括对芯片的复位（硬件复位和软件复位），配置其工作模式，配置 PAR 地址和 MAC 地址，最后还要对系统定时器初始化。其流程图如图 4.17 所示。详细程序代码见“附录三：核心代码”的“RTL8019AS 驱动程”序部分。



Frank_Luo_Long_Luo@outlook.com

图 4.17 初始化网卡流程图

4) 数据包发送函数 etherdev_send()

发送的过程是这样的：先将待发送的数据包存入芯片 RAM，给出发送缓冲区首地址和数据包长度(写入 TPSR、TBCR0,1)，启动发送命令(CR=0x3E)即可实现 8019 发送功能。8019 会自动按以太网协议完成发送并将结果写入状态寄存器。其流程图如图 4.18 所示。详细程序代码见“附录三：核心代码”的“RTL8019AS 驱动程”序部分。

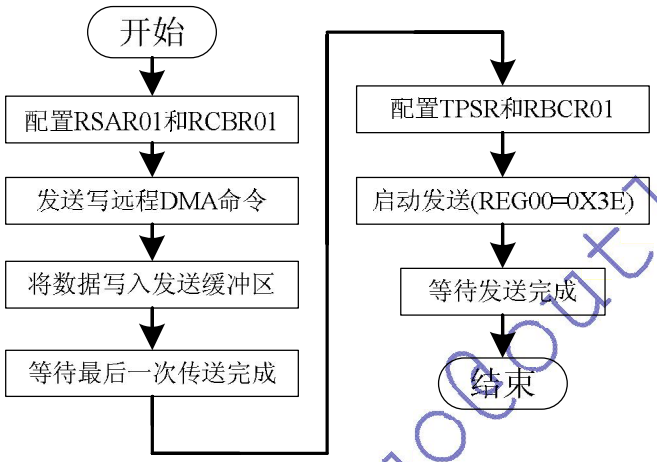
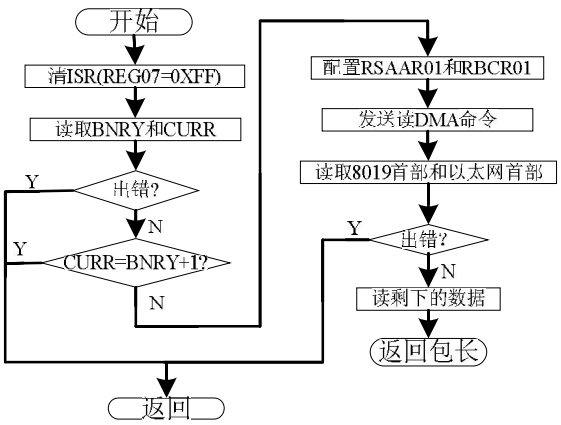


图 4.18 以太网数据包发送流程图

5) 数据包接收（查询）函数 etherdev_poll()

接收缓冲区构成一个循环 FIFO 队列，PSTART、PSTOP 两个寄存器限定了循环队列的开始和结束页，CURR 为写入指针，受芯片控制，BNRY 为读出指针，由主机程序控制。根据 CURR 是否=BNRY+1 可以判断是否收到新的数据包，新收到的数据包存于以 CURR 指出的地址为首址的 RAM 中。当 CURR==BNRY 时芯片停止接收数据包。接收数据包的流程图如图 4.19 所示。详细程序代码见“附录三：核心代码”的“RTL8019AS 驱动程”序部分。



Frank Luo long luoout look.com

图 4.19 以太网数据包接收流程图

4.2.5 uIP 应用层接口

uIP 本身没有提供底层和顶层的驱动支持，用户在使用时，需要自己编写这部分接口程序。前面对底层接口驱动进行了介绍，接下来介绍应用程序接口 UIP_APPCALL()。在 uIP 应用中，主程序框架基本是一样的，应用程序都写在 UIP_APPCALL()中，由 uIP 内核负责调用。该系统的应用层接口程序主要负责响应监听的 3 个端口（HTTP 端口 80、串口通信参数配置端口 8079 和用于串口转以太网接口的数据中转端口 8080）。其流程图如图 4.20 所示。详细程序代码见“附录三：核心代码”的“UIP_APPCALL 代码”序部分。

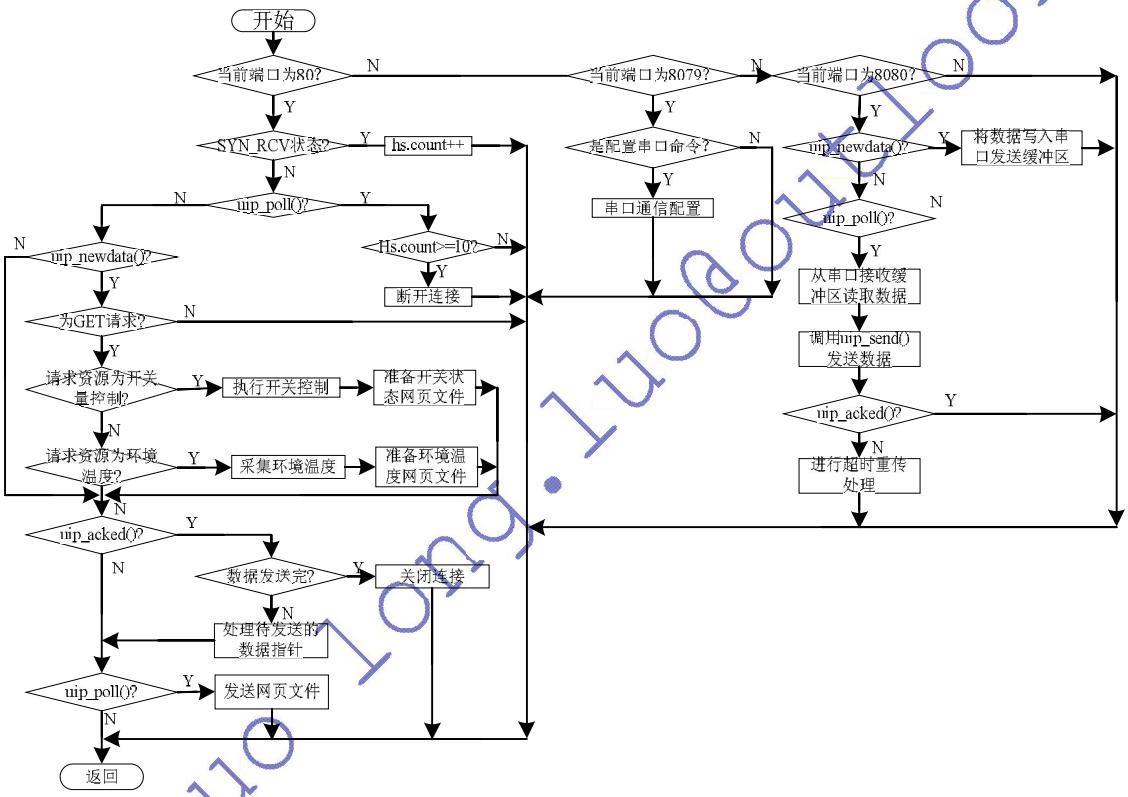


图 4.20 uIP 应用层接口程序流程图

4.3 超文本传输协议:HTTP

HTTP 是一个客户端和服务器端请求和应答的标准（TCP）。客户端是终端用户，服务器端是网站。通过使用 Web 浏览器，客户端发起一个到服务器上指定端口（默认端口为 80）的 HTTP 请求。应答的服务器上存储着（一些）资源，比如 HTML 文件和图像。在用户代理和源服务器中间可能存在多个中间层，比如代理，网关，或者隧道（tunnels）。尽管 TCP/IP 协议是互联网上最流行的应用，HTTP 协议并没有规定必须使用它和（基于）它支持的层。 事实上，HTTP 可以在任何其他互联网协议上，或者在其他网络上实现。HTTP 只假定（其下层协议提供）可靠的传输，任何能够提供这种保证的协议都可以被其

Frank Luo long.luo@outlook.com

使用。

4.3.1 HTTP 的请求方法

通常，由 HTTP 客户端发起一个请求，建立一个到服务器指定端口（默认是 80 端口）的 TCP 连接。HTTP 服务器则在那个端口监听客户端发送过来的请求。一旦收到请求，服务器（向客户端）发回一个状态行，比如"HTTP/1.0 200 OK"，和（响应的）消息，消息的消息体可能是请求的文件、错误消息、或者其它一些信息。

在 HTTP/1.0 中，定义了 3 种方法（GET、HEAD、POST），但最流行、使用最多的方法是 GET，GET 请求可应用于由 URI 指定的资源，并且生成响应资源的当前值。这一响应会返回至发出请求的客户机。在 GET 请求中可包含参数，它是根据用户的输入而构建的。一条请求消息的格式如图 4.9 所示。

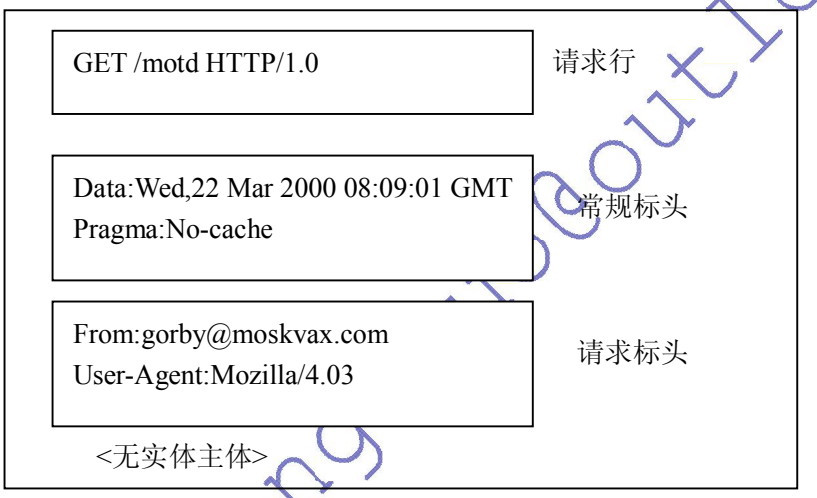


图 4.9 HTTP 请求消息格式

一般客户机第一次请求时，请求的是根目录“/”，消息的请求行为 GET / HTTP/1.0。在接下来的请求中（比如点击了一个链接 index），会给出请求资源名如 GET /index.htm HTTP/1.0。有时，在请求行的后面，不会给出浏览器支持的文件类型等信息。但为了使系统简单化，我们只关心请求行。

4.3.2 HTTP 的响应

如前所述，当服务器收到来自客户机的请求后，无论请求的资源是否存在，都会产生一个 HTTP 响应。响应消息以一个状态行开头，其中包括服务器的 HTTP 版本和响应的状态码。紧随状态行的是可选的常规响应标头，以及一个可选的消息主体。一条 HTTP 响应消息的格式如图 4.10 所示。

Frank_Liu_long@hotmail.com

HTTP/1.0 200 OK	状态行
Data:Wed,22 Mar 2000 08:09:03 GMT	常规标头
Server:Netscape-Enterprise/3.5.1	响应标头
Content-Length:23	实体标头
Welcome to Comer's Vax	实体主体

图 4.10 HTTP 响应消息格式

响应的类型很多，总共分为了成功类响应、重定向响应、客户机错误类响应、服务器错误类响应等。但是嵌入式系统种没有必要支持这么多，一般支持其中几个就可以满足要求。比如当请求的资料存在，则返回 HTTP/1.0 200 OK，若不存在，则返回 HTTP/1.0 404 NOT Found。

4.3.3 简单的 Web 服务器

这里通过介绍一个简单服务器的工作原理，以阐述一般 Web 服务器的工作过程。通过对该简单 web 服务器进行扩充，就可以设计出比较完善的满足实际需要的 web 服务器。

当我们在浏览器中输入一个网站的地址时，客户机就会与服务器的端口（一般是 80 端口）建立一个连接（3 次握手），连接建立后，就发送一个 GET 请求，第一次请求时请求的是根目录“\”，以后就请求实际的资源；服务器收到 GET 请求后，准备回复，若请求的资源存在，就回复“HTTP/1.0 200 OK”，并在后面付上资源数据。数据传送完后，服务器等待一段时间，若客户机没有再发出请求，就断开连接。

当然，这是一个最简单的 Web 服务器、客户机的工作过程，只实现了最简单的文件访问。但若这部分实现了，接下来的高级应用就简单了，比如要实现 CGI 处理表单，要添加脚本支持，只需添加响应的解释程序即可。

4.3.4 HTML 介绍

HTML 即超文本标记语言或超文本链接标示语言，是目前网络上应用最为广泛的语言，也是构成网页文档的主要语言。设计 HTML 语言的目的是为了使资源的访问变得更加方便。HTML 的结构包括头部（Head）、主体（Body）两大部分，其中头部描述浏览器所需的信息，而主体则包含所要说明的具体内容。本系统的 HTML 文档如下所示。

Frank.Liu@Outlook.com

```
<html>
<meta http-equiv=refresh content=3>
<body>
  <center>
    <h1>基于WEB的监控系统设计测试页面</h1>
    <h3>开关量控制:</h3>
    <br><p>点击按钮以开启或关闭LED 1对应LED灭 0对应LED亮</p>
  </center>
  <form action=swc.html>
    <table align=center>
      <tr align=center>
        <td>P1_7: <input type=submit name=sw7 value=1></td>
        <td>P1_6: <input type=submit name=sw6 value=1></td>
      </tr>
    </table>
  </form>
  <center>
  <h3>环境温度: 29.93&degC</h3>
</body>
</html>
```

完整的 HTML 语言是比较复杂的，功能也很强大，在嵌入式系统中，很多功能不能实现，也用不着。

对于本该系统，主要就使用了 FORM 的 ACTION 来实现开关量的控制。当改变开关按钮的值后，就会执行 ACTION 后面指定的请求（比如 GET 请求），服务器收到后，就可以应答请求并改变开关的状态。对于数据的监视，采用的是定时对网页进行刷新的方法以实现动态显示，比如<meta http-equiv="refresh" content="3">表示没隔 3 秒刷新一次。

4.4 环形缓冲区的串口驱动

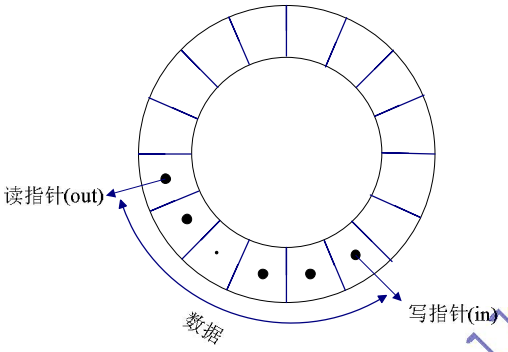
该系统不仅可以作为一个基于 Web 服务器的监控系统，还可以作为一个串口（RS232）到以太网接口（RJ45）的转换器。由于串口的传输速度比较慢，而以太网的传输速度比较快，在进行两种接口的转换时，就要解决速度匹配问题；并且串口传输是以字节为单位，而以太网传输是以数据包为单位，若从串口收到数据就马上从以太网以数据包的方式发送出去，这样会使带宽利用率很低，在数据量大时还会引起冲突，甚至丢失数据，所以不能以这种方式处理。

通过不断的尝试，发现用缓冲区可以解决以上问题。由于串口设备是低速设备，若采用查询方式进行收发，将占用大量宝贵的 CPU 资源，故采用中断方式处理串口数据的收发，并且设置了比较大的缓冲空间以保证在 CPU 负担重时不丢失数据。为了处理方便，定义缓冲区的数据结构如下，环形缓冲区的结构示意图如图 4.11 所示。

Frank Liu long.liuqian.com

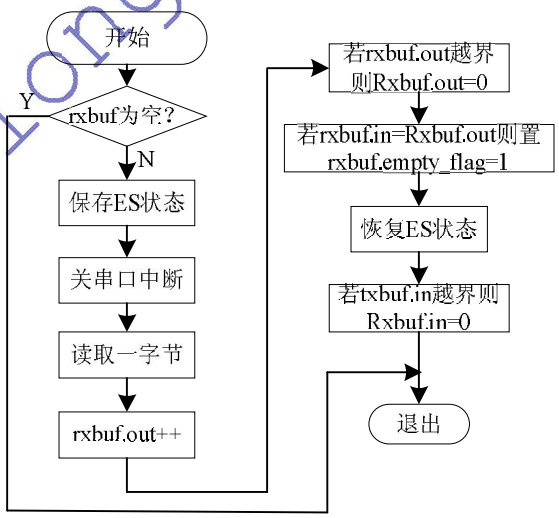
```
struct{
  unsigned int in;
  unsigned int out;
  unsigned char empty_flag;
  unsigned char dt[COM_RXD_BUFF_SIZE];
}rxbuf;
```

```
struct{
  unsigned int in;
  unsigned int out;
  unsigned char empty_flag;
  unsigned char dt[COM_TXD_BUFF_SIZE];
}txbuf;
```



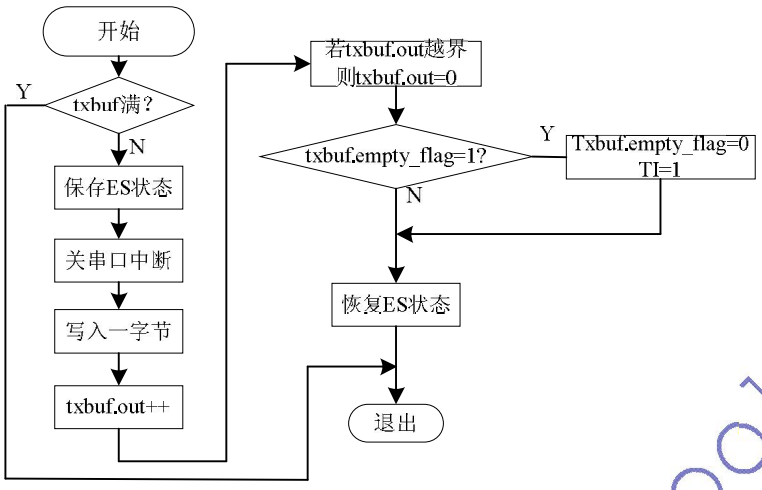
4.11 环形缓冲区示意图

要自动的实现环形缓冲区是不可能的，只能用软件来模拟，但指针越界后将重新指向开始，这样，看起来就像一个连续的环形。下面给出从接收缓冲区中读取一个字符(图 4.12)、写入一个字符到发送缓冲区（图 4.13）和串口收发中断（图 4.14）的流程图。详细程序代码见“附录三：核心代码”的“环形缓冲区的串口驱动”序部分。

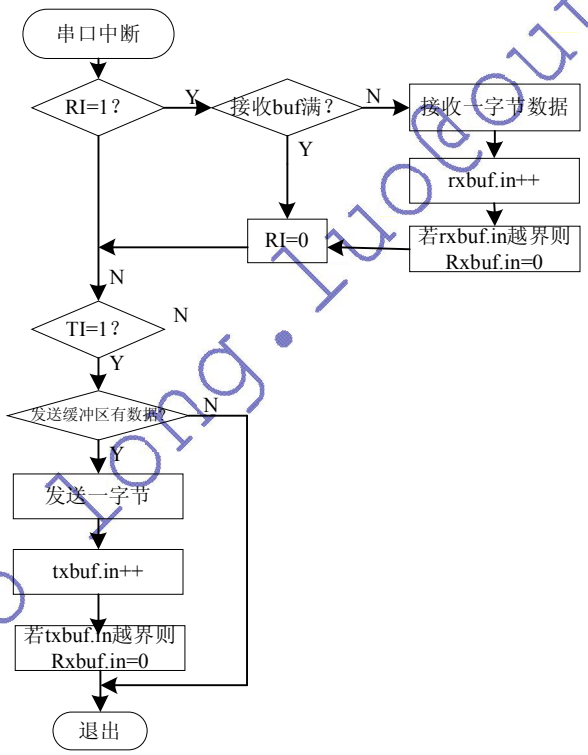


4.12 从接收缓冲区读取一字节

Frank Luo long.luo@outlook.com



4.13 写一字节到发送缓冲区



4.14 串口接收中断函数

但数据被送到缓冲区后，操作就比较方便了。比如要把接收缓冲区的数据通过以太网接口发送出去，只需要调用 uip_send(data,len)，并修该接收缓冲区的读指针即可。uip_send 的原型为 ：

```
#define uip_send(data, len)
do
{ uip_sappdata = (data);
   uip_slen = (len);
} while(0)
```

其中 uip_sappdata 为发送数据的起始指针，uip_slen 为数据的长度。

Frank Luo Long.luo@outlook.com

5 系统调试

前面完成了底层和应用层驱动程序与 uIP 的接口，接下来进行调试。这里主要采用的是 NAI 公司推出的功能强大的协议分析软件 Sniffer Pro。有关 Sniffer 的使用，请参考相关帮助。为了调试方便，我断开了与外部网络的连接，将单片机和 PC 机用一根交叉线相连，组成一个调试时用的小型局域网。

5.1 ARP 协议调试

首先调试的是 ARP 协议，先用 Sniffer 按 ARP 数据包的格式编辑好一个 ARP 请求数据包，然后发送出去，发送 ARP 请求 5 次，图 5.1 是用 Sniffer 捕获的数据包。第一个数据包是 PC（MAC 为 00-0A-EB-5C-8E-C6）发送的 ARP 广播，第二个数据包收到的回应，可以看到单片机系统的 MAC 为 00-01-02-03-04-05。后面 8 个数据包是重复的。

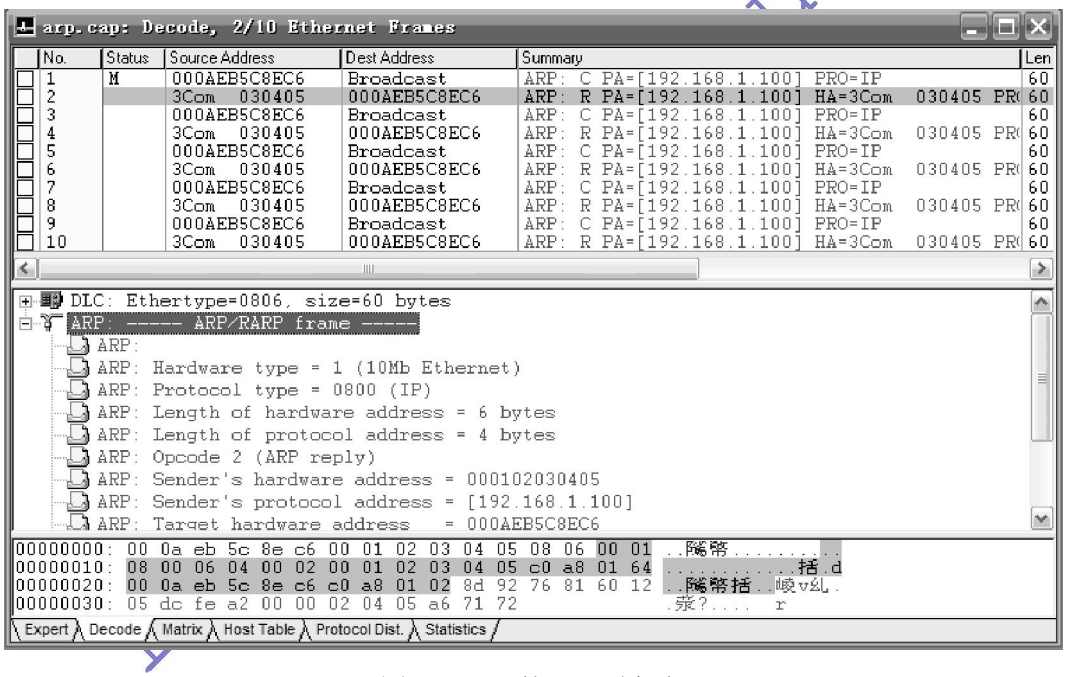


图 5.1 ARP 协议调试抓包

5.2 PING 命令调试

下面调试的是 PING 命令的实现。在 windows 命令提示符下输入 ping 192.168.1.100，回车后收到回应如图 5.2 所示。在默认情况下只执行 4 次 ping 请求，在应答信息中可以看到 PING 数据包的字节数，传送消耗的时间和 TTL 等。图 5.3 是执行 PING 命令过程的抓包，前面两个数据包为 ARP 请求与应答（在 PING 请求时，若系统 ARP 缓存中有目的 IP 的记录，则不会进行 ARP 请求），后面才是 PING 请求和应答。

Frank_xth@hotmail.com

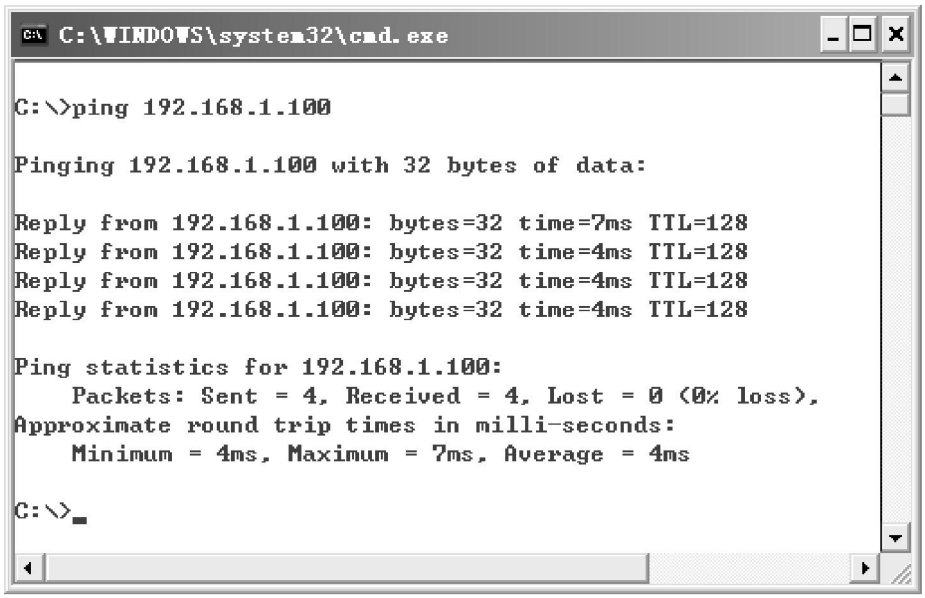


图 5.2 PING 调试记录

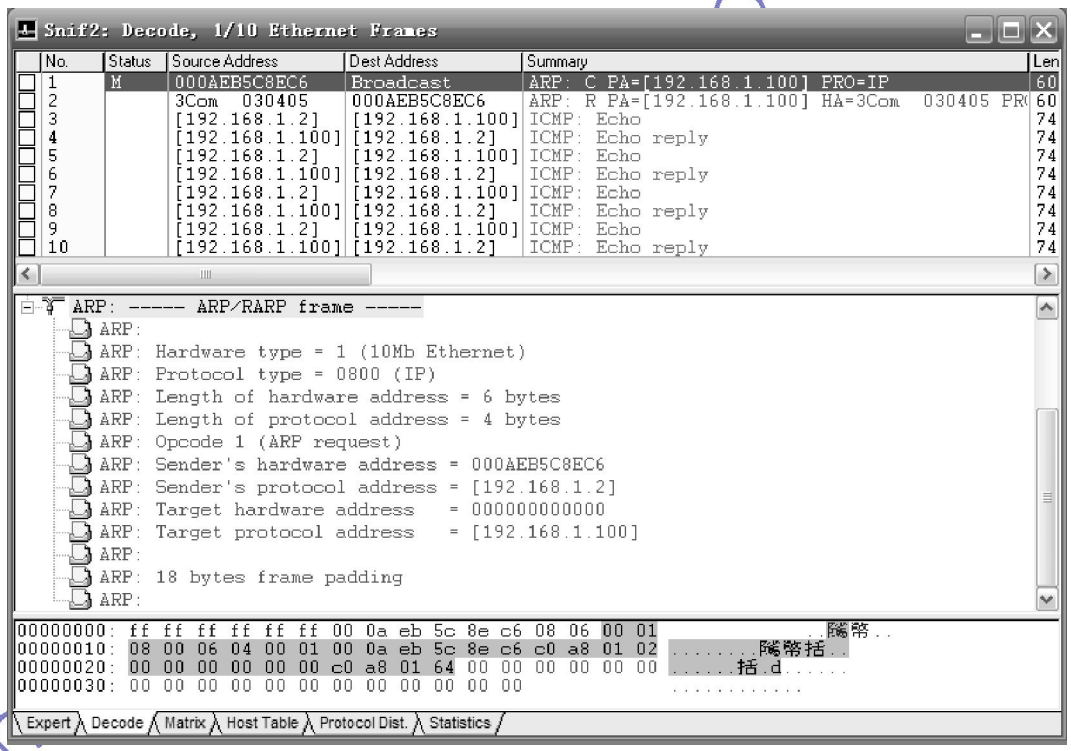


图 5.3 PING 命令调式抓包

5.3 TCP 协议调试

当 ARP 协议和 ICMP（PING）调试完成后，就可以调试 TCP/IP 中最核心的协议 TCP 了。由于 TCP 一般是以客户机和服务器的形式存在，故这里把 PC 配置成客户机，单片机系统配置为服务器。并且 TCP 在工作过程中，状态在不断的改变，所以要特别注意其状态标志。图 5.4 所示程序是使用 winsock 控件在 VB6.0 下编写的一个 RS232 转 RJ45

服务器测试程序。其中 8079 端口用于配置串口通信速率。8080 端口用于串口与以太网接口的数据传送。当 8080 端口打开后，在串口接收到的数据将被服务器转发到网络接口，最终在数据接收区显示。

图 5.4 串口到以太网接口的数据发送测试

图 5.5 是以太网接口到串口进行数据发送的测试界面。首先在图 5.4 所示的程序的数据发送区中填入要发送的数据，再通过 8080 端口发送给服务器，由服务器向串口转发，最终在串口调试助手的接收窗口中显示。

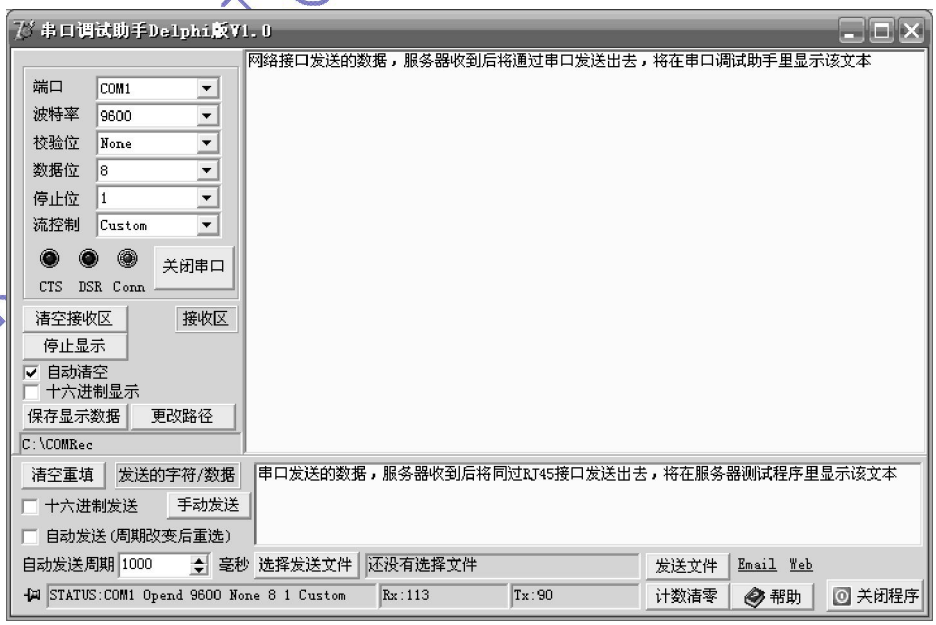


图 5.5 以太网接口到串口的数据发送测试

5.4 HTTP 协议调试

在 TCP 协议正常工作后，应用层的调试就变得十分简单了。图 5.6 所捕获的数据包记录的是用 IE 浏览器访问 Web 服务器资源的的过程。仔细看前三个数据包的 TCP 标志位，发现第一个数据包的 SYN 置位，第二个数据包的 SYN 和 ACK 置位，第三个的 ACK 置位，这就是 TCP 建立连接时的三次握手；随后一个数据包是 PC 发出的 GET 请求，接着收到服务器的应答 HTTP/1.0 200 OK，由于网页文件没有超过一个以太网帧的大小，网页文件直接附在应带包后面了；当网页数据传送完后，就断开连接（四次握手）。图 5.7 位 Web 监控页面的抓图。

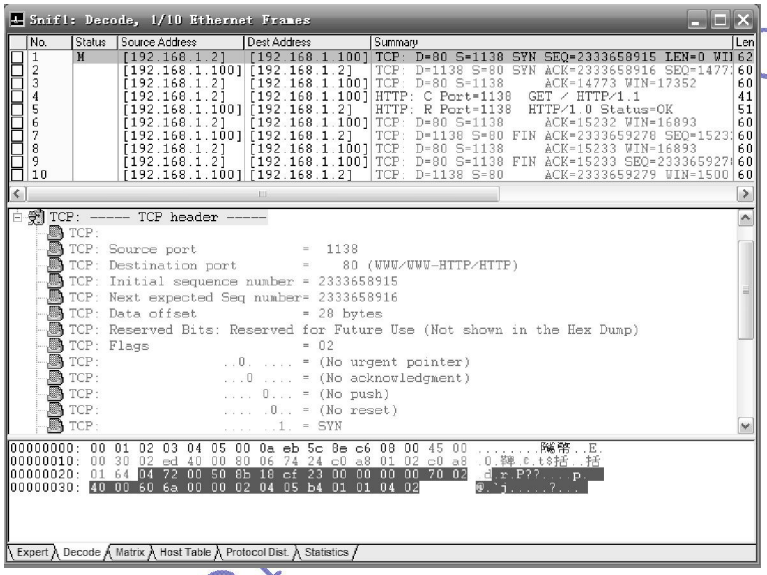


图 5.6 HTTP 请求抓包

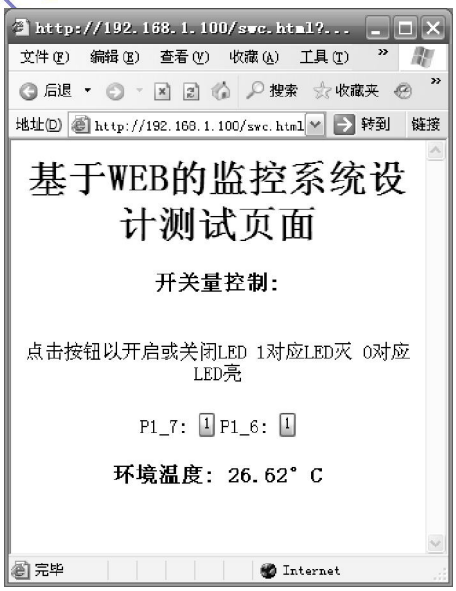


图 5.7 Web 监控页面

Frank Luo 10 book.com

6 总结与体会

虽然我们每天都在与网络接触，但对我来说，TCP/IP 是那么的神秘和陌生，这种神秘和陌生的感觉，给了我学习 TCP/IP 无限的动力。

TCP/IP 十分键全，有着广泛的应用，这不是一个普通的协议能办到的。这一点我在学习 TCP/IP 之初就作好了充分的心里准备。通过近一个月的努力，把《以太网》和《TCP/IP 详解卷》通读了一遍，弄清楚了 TCP/IP 的层次关系和对应的数据帧结构。但接下来做硬件时又让我犯难了。以前重未涉及到网络芯片的使用，在芯片选型上走了不少弯路，起初选择 DM9000A，因为它封装小，又是 100M 的以太网控制芯片，但不知是硬件电路问题还是初始化不对，始终连接不上。最后不得不放弃它而采用 10M 的 RTL8019AS。

在硬件确定后，接下来就开始编写软件。对于刚懂一点点网络知识的我，要在单片机上实现 TCP/IP 协议栈，基本是不可能的。幸亏致运公司的 DP-51PRO 试验箱附带了一个 TCP/IP 的应用试验。在理清了程序结构后，将其思路应用到自己的程序中，先后实现了 ARP、ICMP、UDP 和 TCP。但这只是在功能上实现了，作为学习协议的实现是不错，但要真正应用到系统中却显得力不从心。

接下来的时间里，都在寻找一个合适的 TCP/IP 协议栈，有的需要 RTOS 的支持，有的代码量很大，有的又过于精简……最终我找到了两款（ZLIP 和 uIP）并都移植到了自己的系统中，但由于 ZLIP 不是免费的，效率也没有 uIP 高，我最终选择采用 uIP 协议栈。在协议栈调试通过后，应用层的设计就变的简单了。

在设计的过程中，掌握了许多新的知识。首先，对 TCP/IP 有了一个比较全面的认识：了解了许多网络设备（集线器、交换机、路由等）的功能和工作原理，学会了一些简单的网络参数的配置，掌握了一些网络调试工具（PING、ARP、Sniffer、网络调试助手等）的使用，理解了 TCP/IP 分层处理的优点。其次，通过嵌入式 TCP/IP 协议栈的学习，了解了 TCP/IP 各层协议在嵌入式系统中的实现方法，并进一步掌握了 C 语言的应用技巧（结构体、指针、链表的使用），同时也学会了编写接口驱动程序的基本方法；通过对 HTML 语言的学习，学会编写简单的网页文件；还有，通过对 Web 服务器的学习，了解了 Web 服务器的工作原理，并成功实现了嵌入式 Web 服务器进行远程监控。

通过这次设计，体会到一件产品的设计开发，并不是想象的那么简单，既要注意局部，又要把握全局；同时也提高了信息检索的能力，掌握了许多调试技巧，培养了分析问题、解决问题的能力。这些宝贵的经验，将在以后的学习和工作中发挥重要的作用。

Frank.Luo long.luo@outlook.com

7 谢辞

本系统为嵌入式 Web 服务器监控系统的设计。虽然现在市面上有很多该类产品销售，但他们一般都是团队开发。由于其系统规模大，涉及知识广，软硬件调试复杂，对于个人而言，是相当有困难的。最终，在各种帮助下，还是基本实现了系统的功能,能够通过 PC 端的浏览器监视远程的温度数据，同时还可以进行开关量的远程控制，并且还添

Frank Luo long.luo@outlook.com

8 参考文献

[1] [美]Cilbert Held 著 戴志涛，郑岩 译．以太网（第三版）．北京：人民邮电出版社．2000

[2] [美] Ray,J. Cilbert Held 著 王建华 等译．TCP/IP 开发使用手册．北京：机械工业出版社．1999

[3] [美] W.Richard Stevens 著 范建华 等译．TCP/IP 详解卷 1．北京：机械工业出片社．2000

[4] [美] Jeremy Bentham 著 陈向群等译．嵌入式系统 Web 服务器 TCP/IP Lean．北京：机械工业出片社．2000

[5] [美] Balachander Krishnamurthy Jennifer Rexford 著 范群波 沈金河译．Web 协议与实践．北京：科学出版社．2003

[6] 沈文 黄力岱 吴宗锋. AVR 单片机 C 语言开发应用实例-TCP/IP 篇. 北京：清华大学出版社. 2005

[7] 秦锋．数据结构:C 语言版．合肥：中国科学技术大学出版社．2007

[8] [美]Ellie Quigley 著 杜炜译．Perl 实例精解．北京：清华大学出版社．2003

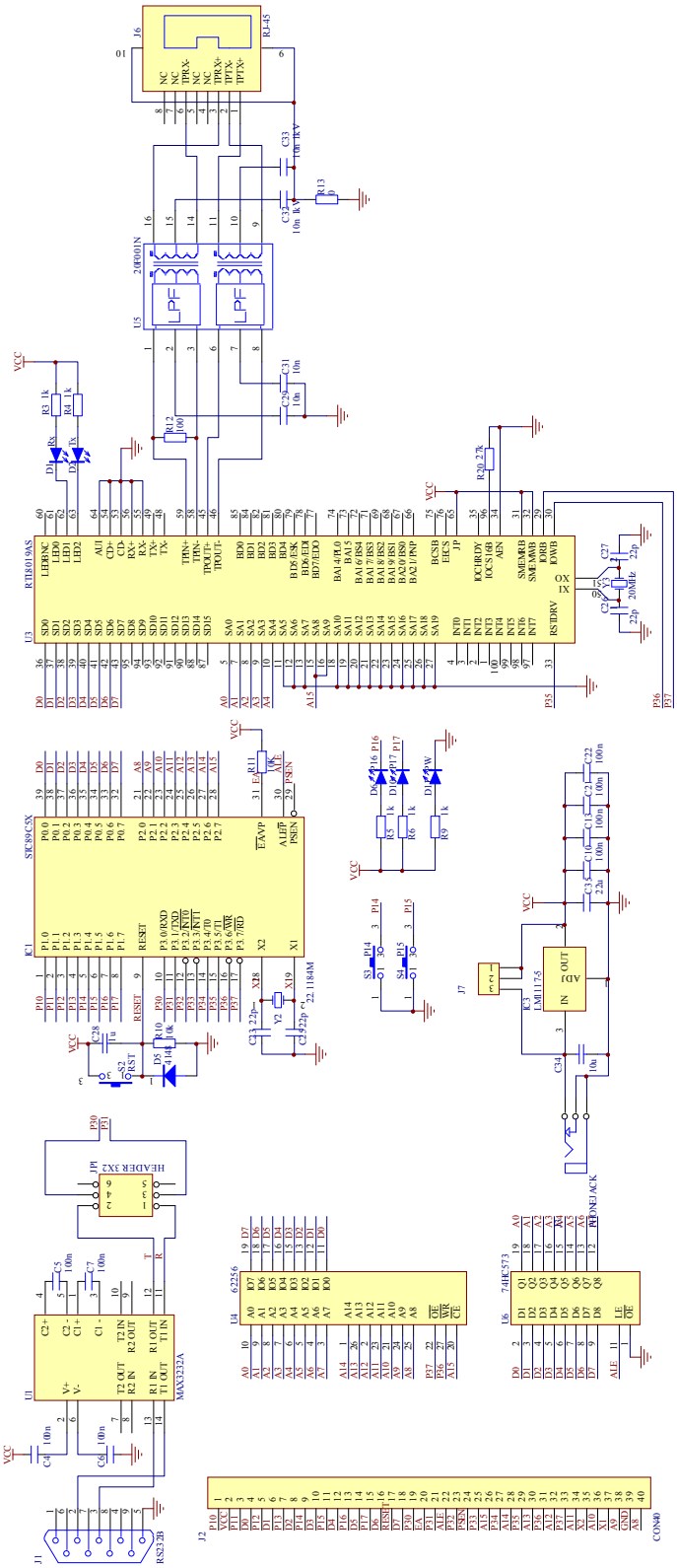
[9] 王忠飞，胥芳编 著．MCS-51 单片机原理及嵌入式系统应用．西安：西安电子科技大学出版社．2007

[10] RTL8019AS 数据手册

Frank Luo long.luo@outlook.com

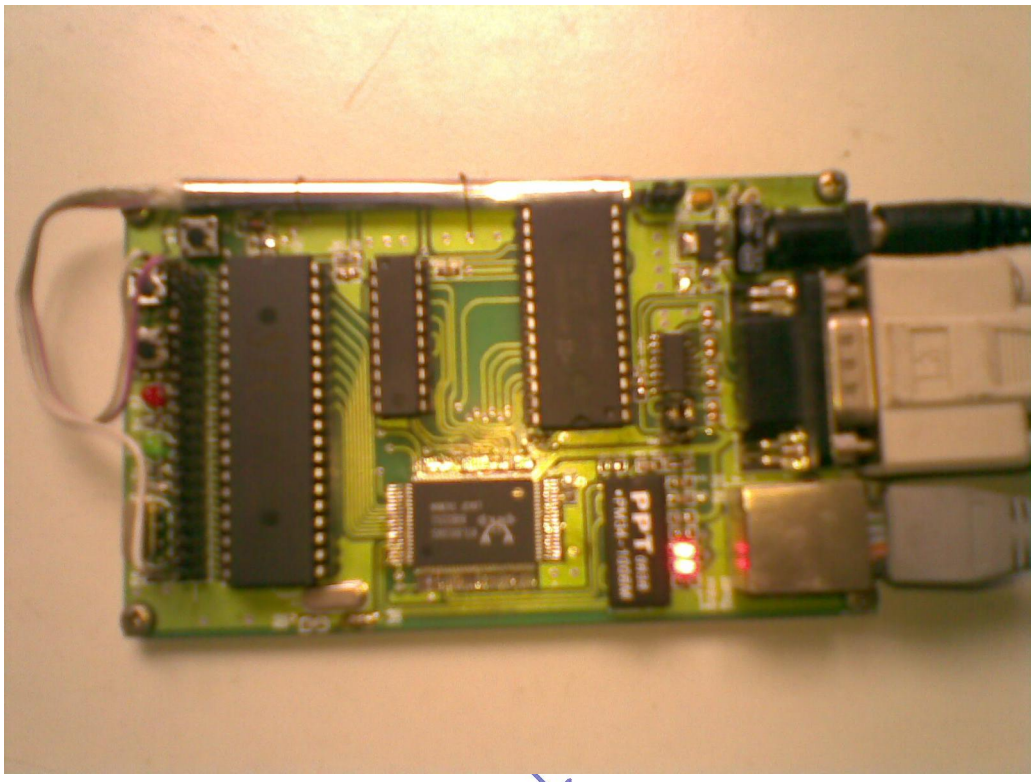
附录：

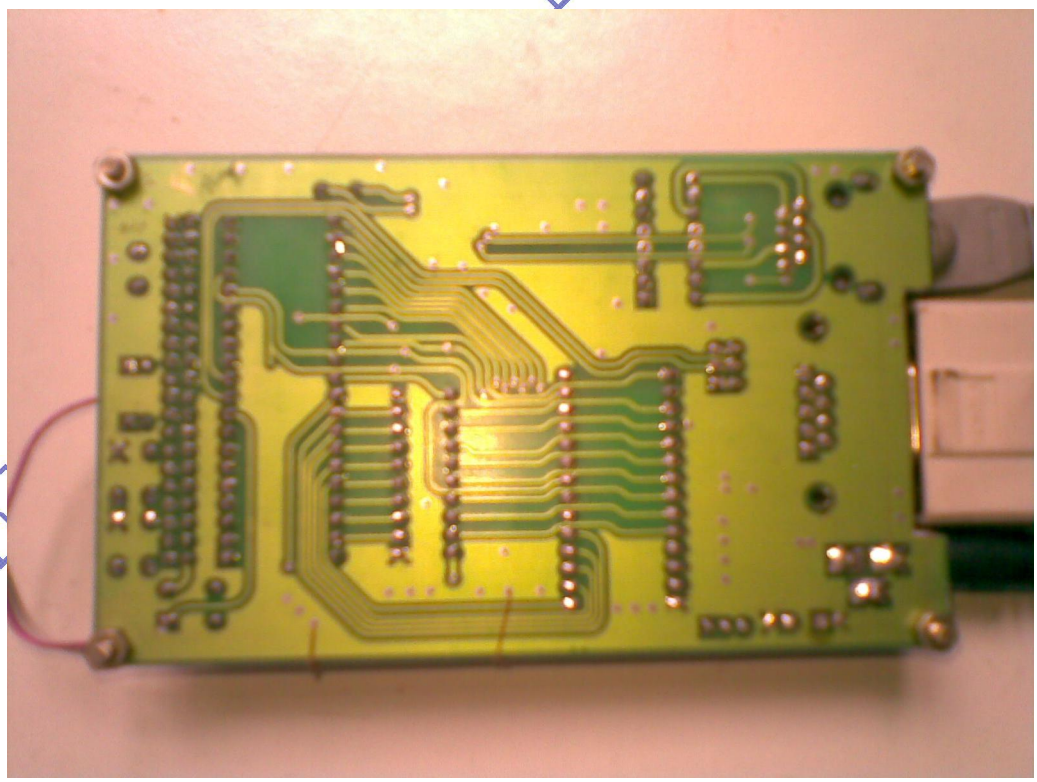
附录一：系统原理图



Frank Jook.com

附录二：系统硬件实物图





Fran .com

附录三：核心代码

环形缓冲区的串口驱动：

```
//-----------------------------------------------------------------------------------------
// fuction  : 收发缓冲区初始化
// parameter: none
// return   : none
//-----------------------------------------------------------------------------------------
void Rx_Buf_Init(void)
{    rxbuf.in  = 0;
     rxbuf.out = 0;
     rxbuf.empty_flag = 1;
     txbuf.in  = 0;
     txbuf.out = 0;
     txbuf.empty_flag = 1;
}
//-----------------------------------------------------------------------------------------
// fuction  : 从接收缓冲区读一字节
// parameter: 存读取字节的地址
// return   : 若有数据返回 0 否则返回 1
//-----------------------------------------------------------------------------------------
unsigned char getUartChar(unsigned char *pInChar)
{   if (rxbuf.empty_flag == 1) return (1);
    // Do not consume char if caller has no storage for it.
    if (pInChar)
    {   bit bdata preservedIeEs;
        *pInChar = rxbuf.dt[rxbuf.out];
        // Protect this section from the UART ISR.
        preservedIeEs = _testbit_(ES);         //这段程序不能被接收中断程序中断
        ++rxbuf.out;
         if( rxbuf.out == COM_RXD_BUFF_SIZE)
         {    rxbuf.out  = 0;
         }
        if (rxbuf.in == rxbuf.out) rxbuf.empty_flag = 1;
        ES = preservedIeEs;
        return 0;
    }
    return 1;
}
```

Frank Luo long.luo@outlook.com

```
//-----------------------------------------------------------------------------------------
// fuction  : 写一字节到发送缓冲区
// parameter: 要发送的字节
// return   : 若成功返回 0 否则返回 1
//-----------------------------------------------------------------------------------------
unsigned char putUartChar(unsigned char outChar)
{   if (txbuf.empty_flag || txbuf.in != txbuf.out)
    {   bit bdata preservedIeEs;
        txbuf.dt[txbuf.in] = outChar;
        // Protect this section from the UART ISR.
        preservedIeEs = _testbit_(ES);    //这段代码不能被发送中断程序中断
        ++txbuf.in;
         if(txbuf.in == COM_TXD_BUFF_SIZE)
         {    txbuf.in = 0;
         }
        // Set not empty and prime the transmitter.
        if (txbuf.empty_flag == 1)
        {   txbuf.empty_flag  = 0;
            TI                = 1;
        }
        ES = preservedIeEs;
        return 0;
    }
    return 1;
}
//-----------------------------------------------------------------------------------------
// fuction  : 串口收发中断程序
// parameter: none
// return   : none
//-----------------------------------------------------------------------------------------
static void uartIsr(void) interrupt 4 using 1
{   static unsigned data inChar;
    if (RI)
    {   // Buffer the char in hopes that there is room left in the ring. Drop
        // the char if not. In any case, ack. the receive interrupt.
        RI = 0;
        inChar   = SBUF;
        if (rxbuf.empty_flag || rxbuf.in != rxbuf.out)
        {   rxbuf.dt[rxbuf.in] = inChar;
            ++rxbuf.in;
              if(rxbuf.in == COM_RXD_BUFF_SIZE)
              {    rxbuf.in = 0;
```

Frank Luo long.luo@outlook.com

```
                }
                rxbuf.empty_flag = 0;
            }
        }
        if (TI)
        {   // If there are chars. left to send, send one and check for empty,
            // otherwise just ack. the transmit interrupt and leave.
            TI = 0;
            if (txbuf.in != txbuf.out)
            {     SBUF = txbuf.dt[txbuf.out];
                  ++txbuf.out;
                  if(txbuf.out  == COM_TXD_BUFF_SIZE)
                  {    txbuf.out  = 0;
                  }
            }
            else
            {   // Only declare empty after the last SCON.TI event occurs.
                txbuf.empty_flag = 1;
            }
        }
}
```

RTL8019AS 驱动程序：

```
//-----------------------------------------------------------------------------------------
// fuction  : RTL8019AS 寄存器页选择
// parameter: pagenumber: 选择的页
// return   : none
//-----------------------------------------------------------------------------------------
void page(unsigned char pagenumber)
{    unsigned char data temp;
     temp=Reg00;//command register
     //temp=temp&0x3f;
     temp=temp&0x3B ;//注意 txp 位不能要
     pagenumber=pagenumber <<6;
     temp=temp | pagenumber;
     Reg00=temp;
}
//-----------------------------------------------------------------------------------------
// fuction  : 配置 RTL8019AS 的 MAC 地址
// parameter: none
// return   : none
//-----------------------------------------------------------------------------------------
```

Frank Luo long.luo@outlook.com

```
void Set_Mac()
{   page(1);
    Reg01 = UIP_ETHADDR0;    Reg02 = UIP_ETHADDR1;
    Reg03 = UIP_ETHADDR2;    Reg04 = UIP_ETHADDR3;
    Reg05 = UIP_ETHADDR4;    Reg06 = UIP_ETHADDR5;
    page(0);
}
//-----------------------------------------------------------------------------------------
// fuction  : 初始化 RTL8019AS
// parameter: none
// return   : none
//-----------------------------------------------------------------------------------------
void etherdev_init(void)
{   unsigned char temp;
    ETH_RST_PIN=0;              //reset
    Delay_Ms(60);
    ETH_RST_PIN=1;     Delay_Ms(10);
    ETH_RST_PIN=0;     Delay_Ms(100);
    //use reset port to reset the ic
    temp=Reg18;   Reg18=temp;
    if(Reg04 & 0x80)      //ISR.RST
        return;
    Reg00=0x21;   //使芯片处于停止模式,这时进行寄存器设置
    Delay_Ms(100); //延时 10 毫秒,确保芯片进入停止模式
    page(0);
    Reg0a=0x00;   //清 rbcr0
    Reg0b=0x00;   //清 rbcr1
    Reg0c= 0xe0;  //RCR,监视模式,不接收数据包
    Reg0d= 0xe2;  //TCR,loop back 模式
    Reg01=0x4c;
    Reg02=0x60;
    Reg03=0x4c;
    Reg04=0x40;  //TPSR,发送起始页寄存器
    Reg07=0xff;  //清除所有中断标志位，中断状态寄存器
    Reg0f=0x00;  //中断屏蔽寄存器清 0，禁止中断
    Reg0e=0xc8;  // 数据配置寄存器，8 位 dma 方式
    page(1);
    Reg07=0x4d;   Reg08=0x00;   Reg09=0x00;   Reg0a=0x00;
    Reg0b=0x00;   Reg0c=0x00;   Reg0d=0x00;   Reg0e=0x00;
    Reg0f=0x00;   Reg00=0x22;  //这时让芯片开始工作
    Set_Mac();  //将芯片物理地址写入到 MAR 寄存器
    page(0);
```

Frank Luo long.luo@outlook.com

```
    //Reg0c=0xcc;  //将芯片设置成正常模式,跟外部网络连接
    Reg0c=0xc4;      //接收广播，不接收多播
    Reg0d=0xe0;
    Reg00=0x22;  //启动芯片开始工作
    Reg07=0xff;  //清除所有中断标志位
    TR0  = 0;
    TMOD &= 0xF0; TMOD |= 0x01; // Set timer counter 0 as mode 1 16 bit timer.
    // Preload for 24Hz periodic interrupt.
    TH0 = ETH_T0_RELOAD >> 8;   TL0 = ETH_T0_RELOAD;
    TR0 = 1;           // Restart timer counter 0 running.
    ET0 = 1;           // Enable timer counter 0 overflow interrupt.
    EA = 1;            // Enable global interrupt.
}
//-----------------------------------------------------------------------------------------
// fuction  : 发送一个数据包
// parameter: TxdNetBuff 为发送数据的指针 length 为数据包的长度
// return   : none
//-----------------------------------------------------------------------------------------
void etherdev_send(void)
{   unsigned int i;
    unsigned char *ptr;
    ptr = uip_buf;
    page(0);                    //切换至第 0 页
    Tx_Buff_Sel=!Tx_Buff_Sel;   //设置两个发缓区,提高发送效率
    if(Tx_Buff_Sel)
    {   Reg09=0x40 ;}           //RSAR1 设置发送页地址
    else
    {   Reg09=0x46 ;}           //RSAR1 设置发送页地址
    Reg08=0x00;                     //RSAR0 read page address low
    Reg0b=uip_len>>8;       //RBCR1 read count high
    Reg0a=uip_len&0xff;     //RBCR0 read count low;
    Reg00=0x12;             //write dma, page0
    // DMA transfer packet from uip_buf & uip_appdata to RTL8019AS local transmit buffer memory.
    for(i = 0; i < uip_len; i++)
    {   if(i == 40 + UIP_LLH_LEN)
        {  ptr = (unsigned char *)uip_appdata;
        }
        Reg10=*ptr++;
    }
    while(Reg00 & 0x40);    //wait for the last transmit complete
    Reg07=0xff;
    if(Tx_Buff_Sel)
```

Frank Luo long.luo@outlook.com

```
    {   Reg04=0x40;   //TPSR   txd packet start;
    }else
    {   Reg04=0x46;  //TPSR         txd packet start;
    }
    // Ethernet packets must be > 60 bytes, otherwise are rejected as runts.
    if(uip_len < ETH_MIN_PACKET_LEN)
    {   uip_len = ETH_MIN_PACKET_LEN;
    }
    Reg06=uip_len>>8;           //high byte counter
    Reg05=uip_len&0xff;         //low byte counter
    Reg07=0xff;   Reg00=0x3e;   //to sendpacket;
    while(Reg04&0x01==0);
}
//-----------------------------------------------------------------------------
// fuction  : 查询是否有新的数据包
// parameter: none
// return   : 若有数据，则返回长度 否则返回 0
//-----------------------------------------------------------------------------
static unsigned int etherdev_poll(void)
{   unsigned char i;
    unsigned int ii,_len;
    unsigned char bnry,curr;
    unsigned char xdata rxdbuf[18];
   page(0);
   Reg07=0xFF;    bnry=Reg03;       //bnry page have read 读页指针
   page(1);
   curr=Reg07;                      //curr writepoint 8019 写页指针
   page(0);
   if(curr==0)
       return 0;                    //读的过程出错
   bnry=bnry++;
    if(bnry>0x5F) bnry=0x4C;
   if(bnry!=curr){                  //此时表示有新的数据包在缓冲区里
    //读取一包的前 18 个字节:4 字节的 8019 头部,6 字节目的地址,6 字节原地址,2 字节协议
    //在任何操作都最好返回 page0
       page(0);
       Reg09=bnry;           //read page address high
       Reg08=0x00;           //read page address low
       Reg0b=0x00;           //read count high
       Reg0a=18;             //read count low;
       Reg00=0x0A;           //read dma
       for(i=0;i<18;i++)
```

Frank Luo long.luo@outlook.com

```
            rxdbuf[i]=Reg10;
        i=rxdbuf[3];      //将长度字段的高低字节掉转
        rxdbuf[3]=rxdbuf[2];
        rxdbuf[2]=i;
        _len=rxdbuf[2]*256+rxdbuf[3]-4; //去掉 4 个字节的 CRC
        //表示读入的数据包有效
        if(((rxdbuf[0]&0x01)==0)||(rxdbuf[1]>0x5F)||(rxdbuf[1]<0x4C)||(rxdbuf[2]>0x06)){
            //接收状态错误,或者 next_page_start 错误或者长度错误,将丢弃所有数据包
            page(1);
            curr=Reg07;               //page1
            page(0);                  //切换回 page0
            bnry=curr-1;
            if(bnry<0x4C) bnry=0x5F;
            Reg03=bnry;               //write to bnry
            return 0;
        } else
        {//表示数据包是完好的.读取剩下的数据
            if((rxdbuf[16]*256+rxdbuf[17]==0x0800)||(rxdbuf[16]*256+rxdbuf[17]==0x0806))
            {    //协议为 IP 或 ARP 才接收
                Reg09=bnry;   //RSAR1 read page address high
                Reg08=4;      //RSAR0 不要 4byte 的 8019as 头部
                Reg0b=_len>>8;        //RBCR1  read count high
                Reg0a=_len&0xFF;      //RBCR0  read count low;
                Reg00=0x0A;           //re/ad dma
                for(ii=0;ii<_len;ii++)
                    *(uip_buf+ii)=Reg10;
                //终止 DMA 操作
                Reg0b=0x00;           //read count high
                Reg0a=0x00;           //read count low;
                Reg00=0x22;           //结束或放弃 DMA
            }
            bnry=rxdbuf[1]-1;//next page start-1
            if(bnry<0x4C) bnry=0x5F;
            Reg03=bnry;       //write to bnry
            return _len;          //返回包长
        }
    }
    return 0;
}
```

UIP_APPCALL 代码:

```
void appcall(void)
{  switch(uip_conn->lport) {
```

Frank Luo long.luo@outlook.com

```
case HTONS(80): /* http port */
    /* when syn_rcv the uip_connected() return not zero ,it means
       client connect to host*/
    if(uip_connected()) {
        /* init the byte count */
        hs.count = 0;
        return;
    }
    else if(uip_poll()) {
    /* If we are polled ten times, we abort the connection. This is
       because we don't want connections lingering indefinately in
       the system.  */
        if(hs.count++ >= 10) {
            uip_abort();
        }
        return;
    }
    else if(uip_newdata())
    {if(uip_appdata[0]!='G' || uip_appdata[1]!='E' || uip_appdata[2]!='T' || uip_appdata[3]!=' ')
        uip_abort();
        /* control the led on and off */
        /* 0123456789abcdef */
        /* GET /swc.html?sw7=1 */
        if(uip_appdata[4]=='/'&&uip_appdata[5]=='s'&&uip_appdata[6]=='w'&& uip_appdata[7]=='c')
        {   if(uip_appdata[16]=='7')
            {   if(uip_appdata[18]=='1')
                {   P1_7=0;
                    htmlpage[0x146]='0';
                }
                else
                {   P1_7=1;
                    htmlpage[0x146]='1';
                }
            }
            if(uip_appdata[16]=='6')
            {   if(uip_appdata[18]=='1')
                {   P1_6=0;
                    htmlpage[0x17d]='0';
                }
                else
                {   P1_6=1;
                    htmlpage[0x17d]='1';
```

Frank Luo long.luo@outlook.com

```
                }
            }
        }
        /* change the value of the temperature */
        tt=get_t();          //两位整数两位小数       采用10进制，没采用bcd码
        htmlpage[0x1ae]=(unsigned char)(tt>>8)/10+'0'; //整数
        htmlpage[0x1af]=(unsigned char)(tt>>8)%10+'0';
        htmlpage[0x1af+2]=(tt & 0x00ff)/10+'0';              //小数
        htmlpage[0x1af+3]=(tt & 0x00ff)%10+'0';
        hs.dataptr=htmlpage;
        hs.count=sizeof(htmlpage)-1;
    }
//若收到client的ack包，说明先前的数据已成功发送，进行剩下的数据处理，若没数据要发，就关闭连接
    if(uip_acked()) {//若没发送完，长度和开始指针还是存在hs->count 和hs->dataptr中
        if(hs.count >= uip_conn->len) {   /* uip_conn.len contain the bytes sent just now */
            hs.count -= uip_conn->len;
            hs.dataptr += uip_conn->len;
        }else
        {   hs.count = 0;
        }
        if(hs.count == 0)
        {   uip_close();   /* 数据发送完成，关闭连接 */
        }
    }
    /* 在没有重发数据且在ESTABLISHED状态时，才会出现uip_flags = UIP_POLL; */
    if(!uip_poll()) {
    /* Send a piece of data, but not more than the MSS of the    connection.    */
        uip_send(hs.dataptr, hs.count);
    }
    return;
case HTONS(8079):       //*串口参数配置
    if (uip_newdata()) //*UIP_CONNECTED UIP_NEWDATA UIP_ACKDATA
    {   if(uip_appdata[0]=='s' && uip_appdata[1]=='e' && uip_appdata[2]=='t')
        if(uip_appdata[3]=='c' && uip_appdata[4]=='o' && uip_appdata[5]=='m')
        {   switch(uip_appdata[6])
            {   case '0':
                    RCAP2H=0xf7;
                    RCAP2L=0x00;
                    puts("300bps\r\n");
                    break;
                case '1':
                    RCAP2H=0xfb;
```

Frank_Luo_long.luo@out-look.com

```
        RCAP2L=0x80;
        puts("600bps\r\n");
        break;
    case '2':
        RCAP2H=0xfd;
        RCAP2L=0xc0;
        puts("1200bps\r\n");
        break;
    case '3':
        RCAP2H=0xfe;
        RCAP2L=0xe0;
        puts("2400bps\r\n");
        break;
    case '4':
        RCAP2H=0xff;
        RCAP2L=0x70;
        puts("4800bps\r\n");
        break;
    case '5':
        RCAP2H=0xff;
        RCAP2L=0xb8;
        puts("9600bps\r\n");
        break;
    case '6':
        RCAP2H=0xff;
        RCAP2L=0xdc;
        puts("19200bps\r\n");
        break;
    case '7':
        RCAP2H=0xff;
        RCAP2L=0xee;
        puts("38400bps\r\n");
        break;
    case '8':
        RCAP2H=0xff;
        RCAP2L=0xf4;
        puts("57600bps\r\n");
        break;
    case '9':
        RCAP2H=0xff;
        RCAP2L=0xfa;
        puts("115200bps\r\n");
```

Frank Luo long.luo@outlook.com

```
                        break;
                    default:
                        RCAP2H=0xff;
                        RCAP2L=0xb8;
                        puts("9600bps\r\n");
                        break;
                }
            }
        }
    return;
    //用于串口到网络接口的数据传送
    case HTONS(8080):
        if (uip_newdata()) //UIP_CONNECTED UIP_NEWDATA UIP_ACKDATA
        {   unsigned int i;
            for(i=0;i<uip_len;i++)
            {   put(*(uip_appdata+i));}
            puts("\r");
        }
        if(uip_poll())
            if(uip_comrd!=uip_comwr)    //有数据
            {   unsigned int temp;
                temp=uip_comwr;
                if(temp>uip_comrd)      //wr 在 rd 后
                {   uip_slen=temp-uip_comrd;
                    uip_send(&uip_combuf[uip_comrd],uip_slen);
                }
                else                    //wr 在 rd 前   分两次发送
                {   uip_slen=UIP_COM_RX_BUF_SIZE-uip_comrd;
                    uip_send(&uip_combuf[uip_comrd],uip_slen);
                    uip_send(uip_combuf,temp);
                }
                uip_comrd=uip_comwr;
            }
    return;
    default:
    uip_abort();
    break;
  }
}
```

Frank Luo long.luo@outlook.com

uIP 协议栈框架代码

```
void main(void)
{    u8_t xdata i, arptimer;
     uip_init();        /* Initialise the uIP TCP/IP stack. */
     etherdev_init();   /* Initialise the device driver. */
     uip_arp_init();    /* Initialise the ARP cache. */
     Uart_Init ();      /*Initialise uart*/
     uisten_init();     /*Initialise listen port */
     DS18B20_RST();     /*reset the temperature sensor */
     EA=1;
    arptimer = 0;
    while(1)
    {   uip_len = etherdev_read();
        if(uip_len == 0)
        {    for(i = 0; i < UIP_CONNS; i++)
             {   uip_periodic(i);
                 if(uip_len > 0)
                 {   uip_arp_out();
                     etherdev_send();
                 }
             }
             if(++arptimer == 20)
             {   uip_arp_timer();
                 arptimer = 0;
             }
        }
        else /* (uip_len != 0) Process incoming */
        {  if(BUF->type == htons(UIP_ETHTYPE_IP))
             {  uip_arp_ipin();
                uip_input();
                /* If the above function invocation resulted in data that should be sent out on the network,
                   the global variable uip_len is set to a value > 0. */
                if(uip_len > 0)
                {  uip_arp_out();
                   etherdev_send();
                }
             }
             else if(BUF->type == htons(UIP_ETHTYPE_ARP))
             {   uip_arp_arpin();
                 /* If the above function invocation resulted in data that
                    should be sent out on the network, the global variable
                    uip_len is set to a value > 0. */
```

Frank luo long.luo@outlook.com

```
                if(uip_len > 0)
                {   etherdev_send();
                }
            }
        }
    }
    return;
}
```

Frank Luo long.luo@outlook.com

附录四:外文翻译

译文：

RFC793 传输控制协议

1.介绍

传输控制协议在包交换计算机通讯网络和这些网络的互联系统中作为一个高可靠性的主机到主机协议使用。

本文档描述了传输控制协议执行的功能，实现的程序，和程序接口或者要求它的服务的用户。

1.1 动机

计算机通信系统在军事、政府和民用环境中起着越来越重要的作用。本文档主要将注意力集中在军用计算机通讯要求，尤其是在不可靠通信中的坚固和在拥塞情况下的可用，但是很多这样的问题在民用和政府用途环境中也会碰到。随着战略上和战术上的计算机通讯网络的发展和应用，提供一种方法来连接这些网络，以及提供可用的支持大量应用程序的标准进程间通信协议是必要的。预料到该标准的需要，国防研究和工程副部长宣告了这里描述的 TCP 协议，来作为 DoD 范围的互联协议标准的基础。

TCP 是一个基于连接的、端到端的可靠协议，该协议设计以用来适应一个支持多个网络应用程序的层间协议结构。TCP 提供了在属于不同的但是是互联的计算机通信网络的宿主主机中的进程对间的可靠进程间通讯。在 TCP 层之下，很少考虑到通信协议的可靠性。TCP 假定它可以从底层协议获得一个简单的、潜在的不可靠数据报。原理上，TCP 必须能够在一个从有线连接到包交换或者回路交换网络的比较大范围的通讯系统上工作。

TCP 基于 Cerf 和 Kahn 在[1]中第一次描述的概念。TCP 适用于一个层间协议架构，该架构在一个基本的 Internet 协议之上，Internet 协议为 TCP 提供了一种发送和接收封装在 internet 数据报中的可变长度分片的方法。internet 数据报提供了在不同网络中寻址源和目的 TCPs 的方法。Internet 协议也处理为在多个网络和互联网关上取得传输和投递所进行的 TCP 分片的分片和重组。internet 协议也携带优先级、安全分类和 TCP 分片的分隔，因此这些信息可用通过多个网络进行端到端传输。本文档大部分是在宿主计算机中位于高层协议下的 TCP 实现上下文中写的。有些计算机网络会通过承载有 TCP 和 ip 层以及网络特定软件前端计算机连接到网络。TCP 规范描述了到高层协议的接口，这些接口即使对于前端情况也是可实现的，同时一个适合的主机到前台（host-to-front）端协议也被实现。

Frank Luo long.luo@outlook.com

```
Protocol Layering
+---------------------+
|     higher-level    |
+---------------------+
|         TCP         |
+---------------------+
|  internet protocol  |
+---------------------+
|communication network|
+---------------------+
```

Figure 1

1.2 范围（Scope）

TCP 用来在多网络环境下提供一个可靠的进程到进程通讯服务。TCP 用来作为在多网络条件下的通用的主机到主机协议。

1.3 关于这个文档

本文档提出了任何 TCP 实现所要求的行为的规范，包括同高层协议的交互以及同其它 TCPs 的交互。文档的其它部分提高了一个协议接口和操作的简要视图。第二部分概要描述了 TCP 设计的理论基础，第三部分提供了不同事件发生情况下（新分片到达，用户调用，错误等）的 TCP 要求的行为和 TCP 分片的详细描述。

1.4 接口

TCP 接口一端是用户或者应用程序，另一端是底层协议如 IP 协议。应用程序和 TCP 间的接口将详细阐述。该接口包含一套调用，类似于操作系统提供给应用程序操作文件的接口。比如，有打开和关闭连接以及在已经建立的连接上发送数据报的接口。同时希望 TCP 可以异步地同几个应用程序通信。虽然 TCP 的实现者有一定的自由设计适合于特定操作系统环境的接口，对任何一个合法的实现来说，都需要实现一个 TCP/用户接口的最小接口集合。除了认为存在一个机制可以互相异步地传递信息，TCP 和底层协议的接口没有特别指定。通常希望底层协议来指定该接口。TCP 设计来在一个非常通用的互联网络上工作。本文档假定的底层协议是 Internet 协议[2]。

1.5 操作

如上所述，TCP 的一个主要目的是在进程间提供可靠的，安全的逻辑回路或者连接服务。为了在一个比较不可靠的相互通信上提供这个服务，系统要求提供如下功能：基本的数据传输（Basic Data Transfer）、可靠性（Reliability）、流控（Flow Control）多路(Multiplexing)、连接(Connections)、优先级和安全性(Precedence and Security) TCP 在这些领域的基本操作在下面的段落中描述。

Frank_Luo_long_luo@outlook.com

基本数据传输：

TCP 通过将一定量的字节打包成在 internet 系统上传输的分片，能够在用户之间在两个方向上传输连续的字节流。通常情况下，TCPs 决定什么时候阻塞以及前推数据。有时候用户需要确保他们所提交给 TCP 的所有数据都被传输。基于这个目的，定义了 push 功能。为了确认提交给 TCP 的数据报确实被传送了，发送用户指示数据必须被推给接收用户。Push 导致了 TCPs 立即前推和投递数据给接收者。确切的 push 点对接收用户可能不可见，且 push 功能不提供一个记录边界标识。

可靠性（Reliability）

TCP 必须从被 internet 通信系统破坏的，丢失的，复制的或者非正确顺序投递的数据中恢复。这是通过给每个传输的字节流分片一个系列号实现的，且要求从接收 TCP 有肯定的确认（ACK）。如果 ACK 在一个间隔内（timeout interval）没有收到，数据就重传。在接收端，系列号用来正确对分片排序（分片可能次序颠倒）以及排除重复分片。通过给每个传输分片增加一个校验和来处理损坏，在接收端进行检查，且丢弃损坏的分片。

流量控制（Flow Control）

TCP 为接收者提供一个办法让其控制发送者发送的数据的数量。这是通过在每个 ACK 中返回一个窗口（“window”）来指示超过最后成功接收的一个分片的可接受的系列号码的范围。窗口指示了发送者在接收到进一步的允许前可以传输的字节的数量。

多路技术（Multiplexing）

为了允许在一个单独的主机里多个进程同时使用 TCP 通信机制，TCP 提供了一套地址和端口。从 internet 通信层同网络和宿主地址连接，这形成了一个 socket。一对 socket 标识了一个连接。也就是说，一个 socket 可能同时被使用在多个连接中。绑定端口到进程被每个主机单独处理。但是，将常用的进程（如“logger”或者时间服务）隶属于众所皆知的 socket 被证明是有用的。这些服务就可以通过已知的地址获取到。建立和学习其它进程的端口地址可能包括更加动态的机制。

连接（Connections）

上面描述的可靠性和流量控制机制要求所有的 TCPs 为每个数据流发起和维护某些状态信息。这些信息的结合体，包括 sockets，系列号，和窗口大小，被称为一个连接。每个连接被一套指定两端的 sockets 唯一指定。当两个进程需要通信的时候，他们的 TCPs 必须首先建立一个连接（在每一端初始化状态信息）。当通信完成的时候，连接终止或者关闭以释放资源用于其它用途。由于连接必须在不可靠的主机和不可靠的 internet 通信系统上建立，一个带有基于时钟的系列号的握手机制被用来避免连接的错误初始化。

优先级和安全（Precedence and Security）

Frank Luo Long.luo@outlook.com

TCP 用户可以指示通信的安全性和优先级。当这些特性不需要的时候，规定采用缺省值。

2. 体系（PHILOSOPHY）

2.1 Internetwork 系统的元素

InternetWork 环境由连接到依次通过网关互联的网络的主机组成。这里的网络有可能是局域网络（比如 ETHERNET）或者大网络（比如 ARPANET），但是任何一种情况都是基于包交换技术。产生和消费消息的活动代理是进程。在网络，网关和主机上不同层次的协议支持一个通信系统，该系统提供了进程端口之间在逻辑连接上的双向数据流。术语包（packet）用在这里表示一个主机和它的网络之间的一次事务的数据。我们通常不用关心一个网络内交换的数据块的格式。主机是附着在一个网络上的计算机，从通信网络的观点来看，是包的源和目的地。进程在主机里被视为活动的元素（根据相对普遍的定义，进程是在执行的程序）。终端和文件或者其它 I/O 设备都被视为通过进程的使用彼此通信的东东。因此，所有的通信被视为进程间通信。既然一个进程可能需要在它自己和其它进程间区分通信流，我们想像每个进程可以有几个端口，通过这些端口，进程同其它进程的端口进行通信。

2.2 操作模型

进程通过调用 TCP 传输数据，调用时传递数据的缓冲。TCP 将这些缓冲的数据打包成分片，然后调用 internet 模块来传输每个分片到目的 TCP。接收 TCP 将一个分片中的数据放置到接收用户缓冲，然后通知接收用户。TCPs 在分片里面包含了控制信息来确保可靠有序的数据传输。Internet 通信模型是由一个同每个 TCP 有联系的 internet 协议模块提供了到局域网络的接口，这个 internet 模块组装 TCP 分片到 internet 数据报，然后为这些数据报选路到一个目的 internet 模块或者中继网关。为了在局域网络上传输数据报，它被包含在一个局域网络桢中。包交换可以执行进一步的包装，分片或者其它操作以投递局域网包到目的 internet 模块。

在一个网络间的网关上，从它的局域网包上来的 internet 数据报是未经包装的（unwrapped），这些数据报被检查以确定数据报下一个要经过的网络。然后 Internet 数据报被包装成适合于下一个网络的局域包，然后被选路到下一个网关或者最后的目的地。允许网关将一个 internet 数据报分解成更小的 internet 数据报分片，如果这在下一个网络中传输是必要的话。要这样做，网关要产生一套 internet 数据报，每个携带一个分片。分片在接下去的网关中可能会进一步分解成更小的分片。Internet 分片设计成目的 internet 模块可以组装分片成一个 internet 数据报的格式。

目的 internet 模块从数据报解开分片后（如果需要，经过重组数据报）传给目的 TCP。这个简单的操作被多个细节注释。一个重要的特性是服务类型。这给网关（或 internet 模块）提供了指导它选择在下一个网络上传输的服务参数的信息。包含在服务

Frank Luo Long.luo@outlook.com

类型信息里面的是数据报的优先级信息。数据报也可以携带安全信息来允许主机和操作在不同安全级别环境上的网关来基于安全考虑正确隔离数据报。

2.3 主机环境

TCP 被当作操作系统中的一个模块。用户就像他们访问文件系统那样访问 TCP。TCP 可以调用其它操作系统接口，比如，管理数据结构的接口。假设真正的对网络的接口由设备驱动模块控制。TCP 不直接调用网络设备驱动，而是调用 internet 数据报协议模块，在 internet 数据报协议模块中调用设备驱动。

TCP 机制并不排除在一个前端(台)处理器（front-end processor）上的 TCP 实现。但是，在这样一个实现中，一个主机到前台（host-to-front-end）的协议必须提供功能来支持在本文档中描述的 TCP 用户接口的类型。

2.4 接口

TCP/用户接口提供了用户可以调用的 TCP 的接口，这些接口用来打开或者关闭一个连接，发送或者接收数据，或者获取一个连接的状态信息。这些接口同操作系统中其它用户的接口相似，比如，打开，读取或者关闭一个文件的接口。

TCP/internet 接口提供了在 internet 系统中发送和接收寻址到主机上的 TCP 模块的数据报。这些接口中可以传递地址（address），服务类型（type of service），优先级（precedence），安全（security）和其它控制信息。

2.5 同其它协议的关系

下图展示了 TCP 在协议层次结构中的地位：

```
+------+ +-----+ +-----+    +-----+
|Telnet| | FTP | |Voice|    ...   |  |   Application Level
+------+ +-----+ +-----+    +-----+
      | |       |         |
      +-----+    +-----+    +-----+
      | TCP |    | RTP |    ...   |  |   Host Level
      +-----+    +-----+    +-----+
        |          |          |
      +--------------------------------+
      | Internet Protocol & ICMP       |   Gateway Level
      +--------------------------------+
                  |
         +---------------------------+
         | Local Network Protocol    |   Network Level
         +---------------------------+
```

Frank Luo long.luo@outlook.com

Protocol Relationships

Figure 2.

TCP 被期望能够有效支持高层协议。必须容易确定高层协议如 ARPANET Telnet 或者 AUTODIN II THP 和 TCP 的接口。

2.6 可靠通信（Reliable Communication）

在一个 TCP 连接上的数据流可以可靠的顺序地投递到目的地。传输通过系列号（sequence numbers）和确认（acknowledements）来确保可靠。概念上，每八位字节数据分配一个系列号码。一个分片里面的第一个八位字节数据同分片一块传输，被称为分片系列号码。分片也携带一个确认号码，该确认号码是在相反方向的下一个期望的传输八位字节数据的系列号码。当 TCP 传输一个包含数据的分片的时候，他将该数据分片的拷贝放在重传队列中，然后开始一个定时器，当数据的确认收到的时候，该分片拷贝从队列中删除，如果在定时到达之前没有收到确认，分片被重传。

TCP 的确认并不保证数据被投递到最后的用户，只是保证接收 TCP 将负起该责任。为了控制 TCPs 之间的数据流量，采用了一个流量控制机制。接收 TCP 报告一个“窗口”（window）来发送 TCP。这个窗口指定了接收 TCP 当前准备接收的数据的字节数目，该数据字节从确认号码开始。

2.7 连接建立和清除（Connection Establishment and Clearing）

为了鉴别 TCP 可能要处理的分散的数据流，TCP 提供了一个端口标识。因为端口标识是每个 TCP 单独选择了，他们可能不是唯一的。为了在每个 TCP 内提供一个唯一地址，我们将标识 TCP 的一个 internet 地址和一个端口标识结合起来创建一个 socket，这在所有互联的网络上是唯一的。

一个连接由两端的 socket 对完全指定。一个本地的 socket 可以参与到多个外部 socket 的多个连接。一个连接可以用来携带两个方向的数据，这就是“全双工”（full duplex）。

TCP 可以自由地将端口和他们选择的进程联系起来。但是，在任何实现中，有些基本概念是必须的。必须有些众所皆知的 sockets，在这些 socket 上 TCP 仅通过一些方法同合适的进程联系在一起。我们想像进程可以“拥有”端口，且这些进程仅可以在他们拥有的端口上发起连接。（实现拥有权的方式是一个本地问题，但我们想像一个请求端口用户命令，或者唯一分配一组端口给一个指定进程的方法，比如，将端口名字的高位同一个指定的进程联系在一起）

连接由带有本地端口和外部 socket 参数的 OPEN 调用被指定。反过来，TCP 提供了一个（短的）本地连接名字，通过这个名字，用户可以在接下来的调用中索引到这个连接。这里有些关于一个连接的信息必须被记住。为了存储这些信息，我们假定有个称为传输控制块（Transmission Control Block ：TCB）的数据结构。一个实现策略是将一

Frank_Luo_Long_Luo@hotmail.com

个指向该连接的 TCB 的指针作为本地连接名字。OPEN 调用也指定了连接的建立是主动要求的，还是被动接受的。

被动 OPEN 请求意味着进程要接受到来的连接请求而不是尝试初始化一个连接。通常请求一个被动 OPEN 的进程将接受任何调用者的连接请求。在这种情况下，全 0 的外部 socket 用来指示一个未指定的 socket。未指定的外部 socket 仅在被动 OPENs 被允许。

一个服务进程如果希望为其它未知进程提供服务，要以未指定的外部 socket 发起一个被动的 OPEN 请求。然后连接可以通过请求连接到该本地 socket 的任何进程建立。如果这个本地 socket 已知用来同这个服务联系，这将是有帮助的。

将一个 socket 地址同标准服务联系起来的这种众所皆知的 socket 是一个便利的机制。比如，“Telenet－Server”进程永久地分配给一个特定的 socket，其它的 soket 保留给文件传输（File Transfer），远程工作入口（Remote Job Entry），文本产生器（Text Generator），回声（Echoer）和 Sink 进程（后面三个进程测试用）。一个 socket 地址可能被保留来访问一个“Look-Up”服务，该服务将返回指定的 socket，在该 socket 上一个新创建的服务将被提供。众所皆知的 socket 是 TCP 规范的一部分，但是分配 socket 给服务超出了这个规范。（参见[4]）

进程可以创建（issue）被动 OPENs，然后等待其它进程的匹配的主动 OPENs，当连接建立时，TCP 就会通知它。在同一时间彼此向对方发起主动 OPENs 的两个进程将被正确连接。这个弹性对支持分部式计算是至关重要的，在分部式计算中，各个部分异步行动，不考虑对方。

在本地被动 OPENs 和一个外部主动 OPENs 时有两种匹配 socket 的基本情况。第一种情况，本地被动 OPENs 完全指定了外部 socket，在这种情况中，匹配是精确的。第二种情况，本地被动 OPENs 未指定外部 socket，在这种情况下，任何外部的 socket 只要本地 socket 匹配就被接受。其它可能的情况包括部分严格匹配。

如果有几个具有相同本地 socket 的悬而未决的被动 OPENs（记录在 TCBs），如果存在一个指定了外部 socket 的 TCB，则在选择一个未指定外部 socket 的 TCB 之前，外部主动 OPEN 将被匹配到该 TCB。

建立连接的工程利用了同步控制标志（SYN），包含了三条消息的交换。这个交换被称为三步握手[3]。通过包含一个 SYN 的到达分片和由用户 OPEN 命令创建的一个等待 TCB 条目的集合点，连接被发起。当一个连接被发起时，判定本地和外部 socket 的匹配。当系列号（sequence numbers）在两个方向被同步的时候，连接创建。连接的清除也包括分片的交换，此时，携带的是 FIN 控制标志。

2.8 数据通信（Data Communication）

在一个连接上的数据流可以视为字节流。发送用户在每个 SEND 调用中指示数据是否要通过 PUSH 标志的设置被立即推给接收用户。

Frank Luo Long.luo@outlook.com

发送 TCP 允许集结从发送用户过来的数据，以他们便利的方式以分片发送那些数据，直到 push 函数被标记，然后它发送所有的未发送的数据。当接收 TCP 看到 PUSH 标志，它必须立即将数据传递给接收进程而不是等待发送 TCP 的其它数据。

Push 功能和分片边界之间没有必然联系。在任何特定分片的数据可以是一个单独的 SEND 调用的结果，全部或者部分，或者是多个 SEND 调用。PUSH 功能和 PUSH 标志的目的是把数据从发送用户推给接收用户。它不提供一个记录服务（record service）。Push 功能和 TCP/用户接口的数据的缓冲的使用有个结合点。每次，一个 PUSH 标志同放到接收用户缓冲的数据联系在一起，即使缓冲没有满也会返回给用户处理。如果数据到达，且在发现 PUSH 标志前充满了整个缓冲，数据以缓冲大小为单元传给用户。

TCP 还定义了一种同数据接收者通信的方法，就是在某些点上，有紧急数据，接收者正在读取。TCP 并不尝试定义用户在收到悬而未决的紧急数据的通知时确定的行为，但是通用的观念是接收进程将很快采取行动处理紧急数据。

2.9 优先级和安全性

TCP 使用 internet 协议的服务类型头部和安全选项来在每个连接上提供优先级和安全性。不是所有的 TCP 模块必须在多级别安全环境下起作用。有些可能仅限于未分类使用，其它可能仅在一个安全级别和隔离上操作。从而，有些 TCP 实现和对于用户的服务可以限制在多级别安全场合的一个子集上。

运行在多级别安全环境的 TCP 模块必须正确标记外出的分片的安全性，分隔和优先级。这样的 TCP 模块必须同时提供他们的用户或者高层协议入 Telnet 或者 THP 一个接口以允许他们指定要求的安全级别，分隔和连接优先级。

2.10 坚固性原则（Robustness Principle）

TCP 实现遵循了一个通用的坚固性原则：对自己所做的保守，对从其它地方接受的自由。

Frank Luo Long.luo@outlook.com

原文：

RFC793 TRANSMISSION CONTROL PROTOCOL

1. INTRODUCTION

The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks.

This document describes the functions to be performed by theTransmission Control Protocol, the program that implements it, and itsinterface to programs or users that require its services.

1.1. Motivation

Computer communication systems are playing an increasingly importantrole in military, government, and civilian environments. Thisdocument focuses its attention primarily on military computercommunication requirements, especially robustness in the presence ofcommunication unreliability and availability in the presence ofcongestion, but many of these problems are found in the civilian andgovernment sector as well.As strategic and tactical computer communication networks aredeveloped and deployed, it is essential to provide means ofinterconnecting them and to provide standard interprocess communication protocols which can support a broad range ofapplications. In anticipation of the need for such standards, theDeputy Undersecretary of Defense for Research and Engineering has declared the Transmission Control Protocol (TCP) described herein tobe a basis for DoD-wide inter-process communication protocolstandardization.

TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. The TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. Very few assumptions are made as to the reliability of the communication protocols below the TCP layer. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, the TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit-switched networks.

TCP is based on concepts first described by Cerf and Kahn in [1]. The TCP fits into a layered protocol architecture just above a basic Internet Protocol [2] which provides a way for the TCP to send and receive variable-length segments of information enclosed in internet

Frank Luo Long. luoout1ook.com

datagram "envelopes". The internet datagram provides a means for addressing source and destination TCPs in different networks. The internet protocol also deals with any fragmentation or reassembly of the TCP segments required to achieve transport and delivery through multiple networks and interconnecting gateways. The internet protocol also carries information on the precedence, security classification and compartmentation of the TCP segments, so this information can be communicated end-to-end across multiple networks. Much of this document is written in the context of TCP implementations which are co-resident with higher level protocols in the host computer. Some computer systems will be connected to networks via front-end computers which house the TCP and internet protocol layers, as well as network specific software. The TCP specification describes an interface to the higher level protocols which appears to be implementable even for the front-end case, as long as a suitable host-to-front end protocol is implemented.

Protocol Layering

```
+---------------------+
|     higher-level    |
+---------------------+
|          TCP        |
+---------------------+
| internet protocol   |
+---------------------+
|communication network|
+---------------------+
```

Figure 1

1.2. Scope

The TCP is intended to provide a reliable process-to-process communication service in a multinetwork environment. The TCP is intended to be a host-to-host protocol in common use in multiple networks.

1.3. About this Document

This document represents a specification of the behavior required of any TCP implementation, both in its interactions with higher level protocols and in its interactions with other TCPs. The rest of this section offers a very brief view of the protocol interfaces and operation. Section 2 summarizes the philosophical basis for the TCP design. Section 3 offers both a detailed description of the actions required of TCP when various events occur (arrival of new segments, user calls, errors, etc.) and the details of the formats of TCP segments.

Frank Luo long.luo@outlook.com

1.4. Interfaces

The TCP interfaces on one side to user or application processes and on the other side to a lower level protocol such as Internet Protocol. The interface between an application process and the TCP is illustrated in reasonable detail. This interface consists of a set of calls much like the calls an operating system provides to an application process for manipulating files. For example, there are calls to open and close connections and to send and receive data on established connections. It is also expected that the TCP can asynchronously communicate with application programs. Although considerable freedom is permitted to TCP implementors to design interfaces which are appropriate to a particular operating system environment, a minimum functionality is required at the TCP/user interface for any valid implementation. The interface between TCP and lower level protocol is essentially unspecified except that it is assumed there is a mechanism whereby the two levels can asynchronously pass information to each other. Typically, one expects the lower level protocol to specify this interface. TCP is designed to work in a very general environment of interconnected networks. The lower level protocol which is assumed throughout this document is the Internet Protocol [2].

1.5. Operation

As noted above, the primary purpose of the TCP is to provide reliable, securable logical circuit or connection service between pairs of processes. To provide this service on top of a less reliable internet communication system requires facilities in the following areas:

Basic Data Transfer

The TCP is able to transfer a continuous stream of octets in each direction between its users by packaging some number of octets into segments for transmission through the internet system. In general, the TCPs decide when to block and forward data at their own convenience. Sometimes users need to be sure that all the data they have submitted to the TCP has been transmitted. For this purpose a push function is defined. To assure that data submitted to a TCP is actually transmitted the sending user indicates that it should be pushed through to the receiving user. A push causes the TCPs to promptly forward and deliver data up to that point to the receiver. The exact push point might not be visible to the receiving user and the push function does not supply a record boundary marker.

Reliability:

The TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the internet communication system. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted.

Frank Luo Long.Luo@outlook.com

At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments.

As long as the TCPs continue to function properly and the internet system does not become completely partitioned, no transmission errors will affect the correct delivery of data. TCP recovers from internet communication system errors.

Flow Control:

TCP provides a means for the receiver to govern the amount of data sent by the sender. This is achieved by returning a "window" with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission.

Multiplexing:

To allow for many processes within a single Host to use TCP communication facilities simultaneously, the TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket. A pair of sockets uniquely identifies each connection. That is, a socket may be simultaneously used in multiple connections. The binding of ports to processes is handled independently by each Host. However, it proves useful to attach frequently used processes (e.g., a "logger" or timesharing service) to fixed sockets which are made known to the public. These services can then be accessed through the known addresses. Establishing and learning the port addresses of other processes may involve more dynamic mechanisms.

Connections:

The reliability and flow control mechanisms described above require that TCPs initialize and maintain certain status information for each data stream. The combination of this information, including sockets, sequence numbers, and window sizes, is called a connection. Each connection is uniquely specified by a pair of sockets identifying its two sides. When two processes wish to communicate, their TCP's must first establish a connection (initialize the status information on each side). When their communication is complete, the connection is terminated or closed to free the resources for other uses. Since connections must be established between unreliable hosts and over the unreliable internet communication system, a handshake mechanism with clock-based sequence numbers is used to avoid erroneous initialization of connections.

Precedence and Security:

The users of TCP may indicate the security and precedence of their communication.

Frank Luo Long.luo@outlook.com

Provision is made for default values to be used when these features are not needed.

2. PHILOSOPHY

2.1. Elements of the Internetwork System

The internetwork environment consists of hosts connected to networks which are in turn interconnected via gateways. It is assumed here that the networks may be either local networks (e.g., the ETHERNET) or large networks (e.g., the ARPANET), but in any case are based on packet switching technology. The active agents that produce and consume messages are processes. Various levels of protocols in the networks, the gateways, and the hosts support an interprocess communication system that provides two-way data flow on logical connections between process ports. The term packet is used generically here to mean the data of one transaction between a host and its network. The format of data blocks exchanged within the a network will generally not be of concern to us.

Hosts are computers attached to a network, and from the communication network's point of view, are the sources and destinations of packets. Processes are viewed as the active elements in host computers (in accordance with the fairly common definition of a process as a program in execution). Even terminals and files or other I/O devices are viewed as communicating with each other through the use of processes. Thus, all communication is viewed as inter-process communication.

Since a process may need to distinguish among several communication streams between itself and another process (or processes), we imagine that each process may have a number of ports through which it communicates with the ports of other processes.

2.2. Model of Operation

Processes transmit data by calling on the TCP and passing buffers of data as arguments. The TCP packages the data from these buffers into segments and calls on the internet module to transmit each segment to the destination TCP. The receiving TCP places the data from a segment into the receiving user's buffer and notifies the receiving user. The TCPs include control information in the segments which they use to ensure reliable ordered data transmission. The model of internet communication is that there is an internet protocol module associated with each TCP which provides an interface to the local network. This internet module packages TCP segments inside internet datagrams and routes these datagrams to a destination internet module or intermediate gateway. To transmit the datagram through the local network, it is embedded in a local network packet.

The packet switches may perform further packaging, fragmentation, or other operations to achieve the delivery of the local packet to the destination internet module.

Frank Luo Long.luo@outlook.com

At a gateway between networks, the internet datagram is "unwrapped" from its local packet and examined to determine through which network the internet datagram should travel next. The internet datagram is then "wrapped" in a local packet suitable to the next network and routed to the next gateway, or to the final destination.

A gateway is permitted to break up an internet datagram into smaller internet datagram fragments if this is necessary for transmission through the next network. To do this, the gateway produces a set of internet datagrams; each carrying a fragment. Fragments may be further broken into smaller fragments at subsequent gateways. The internet datagram fragment format is designed so that the destination internet module can reassemble fragments into internet datagrams.

A destination internet module unwraps the segment from the datagram (after reassembling the datagram, if necessary) and passes it to the destination TCP.

This simple model of the operation glosses over many details. One important feature is the type of service. This provides information to the gateway (or internet module) to guide it in selecting the service parameters to be used in traversing the next network. Included in the type of service information is the precedence of the datagram. Datagrams may also carry security information to permit host and gateways that operate in multilevel secure environments to properly segregate datagrams for security considerations.

2.3. The Host Environment

The TCP is assumed to be a module in an operating system. The users access the TCP much like they would access the file system. The TCP may call on other operating system functions, for example, to manage data structures. The actual interface to the network is assumed to be controlled by a device driver module. The TCP does not call on the network device driver directly, but rather calls on the internet datagram protocol module which may in turn call on the device driver.

The mechanisms of TCP do not preclude implementation of the TCP in a front-end processor. However, in such an implementation, a host-to-front-end protocol must provide the functionality to support the type of TCP-user interface described in this document.

2.4. Interfaces

The TCP/user interface provides for calls made by the user on the TCP to OPEN or CLOSE a connection, to SEND or RECEIVE data, or to obtain STATUS about a connection. These calls are like other calls from user programs on the operating system, for example, the calls to open, read from, and close a file.

Frank Luo Long.luo@outlook.com

The TCP/internet interface provides calls to send and receive datagrams addressed to TCP modules in hosts anywhere in the internet system. These calls have parameters for passing the address, type of service, precedence, security, and other control information.

2.5. Relation to Other Protocols

The following diagram illustrates the place of the TCP in the protocol hierarchy:

```
+------+ +-----+ +-----+ +-----+
|Telnet| | FTP | |Voice| ... | | Application Level
+------+ +-----+ +-----+ +-----+
| | | |
+-----+ +-----+ +-----+
| TCP | | RTP | ... | | Host Level
+-----+ +-----+ +-----+
| | |
+-------------------------------+
| Internet Protocol & ICMP | Gateway Level
+-------------------------------+
|
+---------------------------+
| Local Network Protocol | Network Level
+---------------------------+
```

Protocol Relationships

Figure 2.

It is expected that the TCP will be able to support higher level protocols efficiently. It should be easy to interface higher level protocols like the ARPANET Telnet or AUTODIN II THP to the TCP.

2.6. Reliable Communication

A stream of data sent on a TCP connection is delivered reliably and in order at the destination.Transmission is made reliable via the use of sequence numbers and acknowledgments. Conceptually, each octet of data is assigned a sequence number. The sequence number of the first octet of data in a segment is transmitted with that segment and is called the segment sequence number. Segments also carry an acknowledgment number which

is the sequence number of the next expected data octet of transmissions in the reverse direction. When the TCP transmits a segment containing data, it puts a copy on a retransmission queue and starts a timer; when the acknowledgment for that data is received,

Frank Luo long.luo@outlook.com

the segment is deleted from the queue. If the acknowledgment is not received before the timer runs out, the segment is retransmitted.

An acknowledgment by TCP does not guarantee that the data has been delivered to the end user, but only that the receiving TCP has taken the responsibility to do so. To govern the flow of data between TCPs, a flow control mechanism is employed. The receiving TCP reports a "window" to the sending TCP. This window specifies the number of octets, starting with the acknowledgment number, that the receiving TCP is currently prepared to receive.

2.7. Connection Establishment and Clearing

To identify the separate data streams that a TCP may handle, the TCP provides a port identifier. Since port identifiers are selected independently by each TCP they might not be unique. To provide for unique addresses within each TCP, we concatenate an internet address identifying the TCP with a port identifier to create a socket which will be unique throughout all networks connected together.

A connection is fully specified by the pair of sockets at the ends. A local socket may participate in many connections to different foreign sockets. A connection can be used to carry data in both directions, that is, it is "full duplex".

TCPs are free to associate ports with processes however they choose. However, several basic concepts are necessary in any implementation. There must be well-known sockets which the TCP associates only with the "appropriate" processes by some means. We envision that processes may "own" ports, and that processes can initiate connections only on the ports they own. (Means for implementing ownership is a local issue, but we envision a Request Port user command, or a method of uniquely allocating a group of ports to a given process, e.g., by associating the high order bits of a port name with a given process.)

A connection is specified in the OPEN call by the local port and foreign socket arguments. In return, the TCP supplies a (short) local connection name by which the user refers to the connection in subsequent calls. There are several things that must be remembered about a connection. To store this information we imagine that there is a data structure called a Transmission Control Block (TCB). One implementation strategy would have the local connection name be a pointer to the TCB for this connection. The OPEN call also specifies whether the connection establishment is to be actively pursued, or to be passively waited for.

A passive OPEN request means that the process wants to accept incoming connection requests rather than attempting to initiate a connection. Often the process requesting a passive OPEN will accept a connection request from any caller. In this case a foreign socket of all zeros is used to denote an unspecified socket. Unspecified foreign sockets are allowed only on

Frank Luo Long.luo@outlook.com

passive OPENs.

A service process that wished to provide services for unknown other processes would issue a passive OPEN request with an unspecified foreign socket. Then a connection could be made with any process that requested a connection to this local socket. It would help if this local socket were known to be associated with this service.

Well-known sockets are a convenient mechanism for a priori associating a socket address with a standard service. For instance, the "Telnet-Server" process is permanently assigned to a particular socket, and other sockets are reserved for File Transfer, Remote Job Entry, Text Generator, Echoer, and Sink processes (the last three being for test purposes). A socket address might be reserved for access to a "Look-Up" service which would return the specific socket at which a newly created service would be provided. The concept of a well-known socket is part of the TCP specification, but the assignment of sockets to services is outside this specification. (See [4].)

Processes can issue passive OPENs and wait for matching active OPENs from other processes and be informed by the TCP when connections have been established. Two processes which issue active OPENs to each other at the same time will be correctly connected. This flexibility is critical for the support of distributed computing in which components act asynchronously with respect to each other.

There are two principal cases for matching the sockets in the local passive OPENs and an foreign active OPENs. In the first case, the local passive OPENs has fully specified the foreign socket. In this case, the match must be exact. In the second case, the local passive OPENs has left the foreign socket unspecified. In this case, any foreign socket is acceptable as long as the local sockets match.Other possibilities include partially restricted matches.

If there are several pending passive OPENs (recorded in TCBs) with the same local socket, an foreign active OPEN will be matched to a TCB with the specific foreign socket in the foreign active OPEN, if such a TCB exists, before selecting a TCB with an unspecified foreign socket. The procedures to establish connections utilize the synchronize (SYN) control flag and involves an exchange of three messages. This exchange has been termed a three-way hand shake [3].

A connection is initiated by the rendezvous of an arriving segment containing a SYN and a waiting TCB entry each created by a user OPEN command. The matching of local and foreign sockets determines when a connection has been initiated. The connection becomes "established" when sequence numbers have been synchronized in both directions.

The clearing of a connection also involves the exchange of segments, in this case

Frank Luo Long.Luo@outlook.com

carrying the FIN control flag.

2.8. Data Communication

The data that flows on a connection may be thought of as a stream of octets. The sending user indicates in each SEND call whether the data in that call (and any preceeding calls) should be immediately pushed through to the receiving user by the setting of the PUSH flag.

A sending TCP is allowed to collect data from the sending user and to send that data in segments at its own convenience, until the push function is signaled, then it must send all unsent data. When a receiving TCP sees the PUSH flag, it must not wait for more data from the sending TCP before passing the data to the receiving process.

There is no necessary relationship between push functions and segment boundaries. The data in any particular segment may be the result of a single SEND call, in whole or part, or of multiple SEND calls.

The purpose of push function and the PUSH flag is to push data through from the sending user to the receiving user. It does not provide a record service.

There is a coupling between the push function and the use of buffers of data that cross the TCP/user interface. Each time a PUSH flag is associated with data placed into the receiving user's buffer, the buffer is returned to the user for processing even if the buffer is not filled. If data arrives that fills the user's buffer before a PUSH is seen, the data is passed to the user in buffer size units.

TCP also provides a means to communicate to the receiver of data that at some point further along in the data stream than the receiver is currently reading there is urgent data. TCP does not attempt to define what the user specifically does upon being notified of pending urgent data, but the general notion is that the receiving process will take action to process the urgent data quickly.

2.9. Precedence and Security

The TCP makes use of the internet protocol type of service field and security option to provide precedence and security on a per connection basis to TCP users. Not all TCP modules will necessarily function in a multilevel secure environment; some may be limited to unclassified use only, and others may operate at only one security level and compartment. Consequently, some TCP implementations and services to users may be limited to a subset of the multilevel secure case.

TCP modules which operate in a multilevel secure environment must properly mark outgoing segments with the security, compartment, and precedence. Such TCP modules must also provide to their users or higher level protocols such as Telnet or THP an interface to

Frank Luo Long.luo@outlook.com

allow them to specify the desired security level, compartment, and precedence of connections.

2.10. Robustness Principle

TCP implementations will follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.

Frank Luo long.luo@outlook.com