凌 阳 大 学 计 划
Sunplus University Program

# unIP 用户手册

发布日：2004-3-5

凌阳大学计划推广中心
北京市海淀区上地信息产业基地中黎科技园 1 号楼 6 层 C 段　邮编：100085
TEL : 86-10-62981668　FAX : 86-10-62985972　E-mail:unsp@sunplus.com.c　http://www:unsp.com.c

# 版权声明

凌阳科技股份有限公司保留对此文件修改之权利且不另行通知。凌阳科技股份有限公司所提供之信息相信为正确且可靠之信息，但并不保证本文件中绝无错误。请于向凌阳科技股份有限公司提出订单前，自行确定所使用之相关技术文件及规格为最新之版本。若因贵公司使用本公司之文件或产品，而涉及第三人之专利或著作权等智能财产权之应用及配合时，则应由贵公司负责取得同意及授权，本公司仅单纯贩售产品，上述关于同意及授权，非属本公司应为保证之责任。又未经凌阳科技股份有限公司之正式书面许可，本公司之所有产品不得使用于医疗器材，维持生命系统及飞航等相关设备。

# 目　　录

# unIP 用户手册

## 1 unIP 简介

unIP 是运行在凌阳公司 U'nSP 系列单片机上的一个精简 TCP/IP 协议栈。协议栈的初始版本由 LwIP 移植而来，之所以不沿用 LwIP 的名字是因为移植工作不仅仅只是 LwIP 说明的 arch 目录下的改动，core 部分也做了不少的修改以适应 U'nSP 的 16bit 的特性（U'nSP 并不具有 8bit 的数据类型，地址也是以 16bit 为单位），因此 unIP 与 LwIP 并不兼容，此外，增加了 DNS Client（域名解析客户端）到协议栈中，以及部分应用实例，例如 WEB SERVER 等等，综合以上原因，给本协议栈重新命名为 unIP，特指是运行于 U'nSP 系列单片机之上的网络协议栈。

### 1.1 unIP Stack 的特性

unIP Stack 从资源消耗来说是一个精简的协议栈，而协议栈的实现却是是非常的完整，表 1 所示就是本协议栈所具有的所有特性：

表1 unIP Stack 特性表

| 特性 | unIP Stack |
|---|---|
| 多网络接口（ethernet，slip） | 支持 |
| ARP | 支持 |
| IP (不支持 IP 分片与重组) | 支持 |
| ICMP（包括 ECHO，和 destination unreach） | 支持 |
| UDP（包括 UDP Checksum 的计算） | 支持，可选 |
| TCP 选项（只支持最大报文段长度 MSS） | 支持 |
| TCP 滑动窗口 | 支持 |
| TCP 慢启动、拥塞避免 | 支持 |
| TCP 快速重传、快速恢复 | 支持 |
| TCP 错序数据重组 | 支持，可选 |
| TCP 紧急数据（urgent data） | 支持 |
| 往返时间估计（RTT） | 支持 |
| DHCP 客户端（可自动获取网络设置） | 支持，可选 |
| DNS 客户端（可解析普通域名以及邮件地址 mx 记录） | 支持，可选 |
| 模式化以及非模式化缓冲区存管理 | 支持 |
| 动态内存管理 | 支持，可选 |
| 原始 API（网络编程） | 支持 |

为了方便用户的使用，本手册也一并详细介绍利用本协议栈在凌阳的 SPCE061 芯片上实现的一个 WEB SERVER 的全过程，并配套有硬件开发板，可供用户实验之用。此开发实例以及源程序可以用来进行二次开发，用户可以很方便的开发自己的网页。

### 1.2 unIP Stack 的硬件需求

unIP Stack 运行需要如下的硬件配置：

- U'nSP 系列 16 位单片机（目前支持凌阳的 SPCE061，SPL16256）
- 需要以太网控制器（目前支持 Davicom 公司的 Dm9000），或者利用串口直接通过 Slip 与 PC 机联网（串口会被占用）。
- 协议栈本身需求 ROM 空间为 21K Word，RAM 消耗可配置参见第 3 节，RAM 的消耗依据配置的不同可以为 400Word 至 600Word 左右。

## 2 协议栈的 API 说明

目前协议栈是以库的形式提供给用户，对协议栈的使用也就是对协议栈所提供的各种 API 的调用，掌握了这些 API 便是掌握了协议栈，也就能自己开发新的应用层协议了。下面是本协议栈所有的 API 函数的说明表，API 总共分 7 部分，包括网络接口层、动态内存管理模块、缓冲区管理模块（pbuf）、UDP 层、TCP 层、DHCP 模块和 DNS 模块。

### 2.1 网络接口层 API 函数列表

网络接口层的功能是建立、配置修改网络接口，本层的 API 均以 netif 作为前缀，后缀说明其功能。本层的所有 API 函数为：netif_init()、netif_add()、netif_set_addr()、netif_remove()、netif_find()、netif_set_default()、netif_set_ipaddr()、netif_set_netmask()和 netif_set_gw()。它们的详细说明见下表.

netif_init ()

| | |
|---|---|
| 函数原型 | void netif_init (voidp) |
| 参数 | 无 |
| 返回值 | 无 |
| 功能说明 | 网络接口部分初始化。 |

netif_add ()

| | |
|---|---|
| 函数原型 | struct netif *netif_add(struct ip_addr *ipaddr, struct ip_addr *netmask,<br>struct ip_addr *gw,<br>void *state,<br>err_t (* init)(struct netif *netif),<br>err_t (* input)(struct pbuf *p, struct netif *netif)) |
| 参数 | ipaddr ip 地址指针 指定新接口 ip 地址<br>netmask ip 地址指针 指定新接口子网掩码 |

| | |
|---|---|
| | gw ip 地址指针 指定新接口网关<br>state 无类型指针 用于传递参数至接口处理函数<br>init 回调函数指针 用于初始化本网络接口包括网络接口的名称例如 eth0，lo0 等等。还要指定新接口最大传输单元，MTU，注册新接口的数据发送回调函数。<br>input 回调函数指针 当网络接口层收到一个完整的链路层数据之后，会将链路信息去掉，然后把 packet 交由此回调函数处理。在 tcp/ip 协议中，此函数应该为 ip 层的 input 函数指针。 |
| 返回值 | 成功，则返回网络接口类型指针，失败则返回 NULL。<br>失败原因： 内存不够。<br>链路初始化出错。 |
| 功能说明 | init 与 input 函数指针均是此网络接口的硬件驱动层的函数（由用户根据实际使用的硬件而编写），init 函数在此函数调用中会立即被调用，以初始化链路层硬件。input 函数会在链路层有数据到达的时候被调用。此函数创建一个可用的网络接口。并返回一个 netif 指针。如果内存不够的话会返回一个 NULL，用户也应该在 init 函数中加以判断，以便当有错误发生的时候返回一个 NULL 指针以通知应用层链路不可用。 |

netif_set_addr ()

| | |
|---|---|
| 函数原型 | void netif_set_addr(struct netif *netif,<br>struct ip_addr *ipaddr,<br>struct ip_addr *netmask,<br>struct ip_addr *gw); |
| 参数 | netif 网络接口指针<br>ipaddr 新的 ip 地址指针<br>netmask 新的子网掩码指针<br>gw 新的网关指针 |
| 返回值 | 无 |
| 功能说明 | 设定网络接口的 ip 地址，子网掩码和网关为给定值。 |

netif_remove ()

| | |
|---|---|
| 函数原型 | void netif_remove(struct netif * netif) |
| 参数 | netif 网络接口指针 |
| 返回值 | 无 |
| 功能说明 | 删除指定网络接口，并回收该接口所占用所有资源。 |

netif_find ()

| | |
|---|---|
| 函数原型 | struct netif *netif_find(char *name) |
| 参数 | name 字符串 网络接口的名字（例如 eth0，sl0 等） |
| 返回值 | 网络接口指针 |
| 功能说明 | 寻找指定名字的网络接口，并返回其地址。若没找到，则返回 0。<br>网络接口的名字是在函数 netif_add 中设定的 init 函数中赋值的。 |

netif_set_default ()

| | |
|---|---|
| 函数原型 | void netif_set_default(struct netif *netif) |
| 参数 | netif 网络接口指针 |
| 返回值 | 无 |
| 功能说明 | 指定系统默认网络接口，当没有匹配的接口时，packet 均通过此接口发送至 internet。<br>对于单网络接口也应该执行此函数，设定默认接口。 |

netif_set_ipaddr ()

| | |
|---|---|
| 函数原型 | void netif_set_ipaddr(struct netif *netif, struct ip_addr *ipaddr); |
| 参数 | netif 网络接口指针<br>ipaddr 新的 ip 地址指针 |
| 返回值 | 无 |
| 功能说明 | 设定指定接口的 ip 地址，可以用本函数实现运行期改变某接口的 ip 地址。 |

netif_set_netmask ()

| | |
|---|---|
| 函数原型 | void netif_set_netmask(struct netif *netif, struct ip_addr *netmast); |
| 参数 | netif 网络接口指针<br>netmask 新的子网掩码指针 |
| 返回值 | 无 |
| 功能说明 | 设定指定接口的子网掩码。 |

netif_set_gw ()

| | |
|---|---|
| 函数原型 | void netif_set_gw(struct netif *netif, struct ip_addr *gw); |
| 参数 | netif 网络接口指针<br>gw 新的网关的 ip 地址指针 |
| 返回值 | 无 |
| 功能说明 | 设定指定接口的网关。 |

## 2.2 动态内存管理模块 API 函数说明

本模块的功能是能够动态的分配和释放一段内存区域，内存分配算法有两种可以选择，一种是源自 LwIP 自带的内存管理模块；另一种是凌阳的 Buddy System 算法。前者相对简单，但是随着管理的内存区域变大，性能会显著下降，后者较为复杂，但是当管理大片内存的时候，效率并不会下降很多，但是在管理小片内存时，可能会存在比较大的浪费。根据需要可以选择两种内存管理算法中的一种。根据 061 的平台考虑，本协议栈目前选择的是第一种分配算法。无论选择何种算法，应用程序的 API 都是一致的，本层的 API：mem_init ()、mem_malloc()、mem_free()、mem_realloc()和 mem_reallocm ()。详细说明如下：

mem_init ()

| | |
|---|---|
| 函数原型 | void mem_init (void) |
| 参数 | 无 |
| 返回值 | 无 |
| 功能说明 | 初始化动态内存区域，是以 ram 为起始地址，长度为 MEMSIZE words 的区域。由于协议栈的其它模块大多使用了内存管理模块,所以此函数应该在所有模块初始化之前初始化。 |

mem_malloc ()

| | |
|---|---|
| 函数原型 | void *mem_malloc(mem_size_t size) |
| 参数 | size 类型为无符号 16 位整形数，值为需要申请的内存块的以字为单位的大小 |
| 返回值 | 返回一个无类型指针。若分配成功则指向分配内存块的起始地址，分配失败则为一个 NULL 指针。 |
| 功能说明 | 在动态内存区域中寻找长度大于或等于 size 的内存块，若存在则将该内存块的前 size 个单元划分出来，交由调用函数使用。 |

mem_free ()

| | |
|---|---|
| 函数原型 | void mem_free(void *mem) |
| 参数 | mem 为无类型指针，值为需要释放内存块的首地址。 |
| 返回值 | 无 |
| 功能说明 | 释放指定内存区域，将标志位置为未使用，若前后有相邻的未使用内存区域，则执行内存空洞合并操作。 |

mem_realloc ()

| | |
|---|---|
| 函数原型 | void *mem_realloc(void *mem, mem_size_t size) |
| 参数 | mem 为无类型指针，值为需要调整大小的内存块的首地址。 |

| | |
|---|---|
| | size 为 16 位整型数，值为需要调整到的大小（以字为单位） |
| 返回值 | 无类型指针，值为调整完毕以后的内存区域的首地址 |
| 功能说明 | 将 mem 的长度缩小为 size 指定的大小，mem 的首地址维持不变，也不进行拷贝操作。此函数只能将指定内存区域缩小，而无法增大。 |

mem_reallocm ()

| | |
|---|---|
| 函数原型 | void *mem_reallocm(void *mem, mem_size_t size) |
| 参数 | mem 为无类型指针，值为需要释放内存块的首地址。<br>size 为 16 位整型数，值为需要调整到的大小（以字为单位） |
| 返回值 | 无类型指针，值为调整完毕以后的内存区域的首地址 |
| 功能说明 | 本函数优先另外申请一段区域大小为 size *Word，然后拷贝源区域数据前 size 个 Word 的数据，最后释放源区域。若在第一步分配时失败则调用 mem_realloc()函数来处理，并返回 mem_realloc()的返回值给调用者。 |

### 2.3 模式化内存管理模块

由于协议栈的特殊性，有些数据结构具有固定的大小，而且会在协议栈的运行过程中频繁使用，对于这一部分数据结构，协议栈采用了固定的内存分配模式。采用这种分配模式的一共有一下几种数据结构：ROM 模式的 pbuf，UDP 协议控制块，TCP 协议控制块，监听状态的 TCP 协议控制块（用来实现服务器程序），TCP 收发 Packet 的队列。用户根据实际应用需要，可以配置这些固定结构数组的大小。具体的配置方法将会在本手册后续的 Lib 配置中叙述。本模块的 API 均为内部 API，在协议栈内部实现之用，用户不需要了解，在此就不细述了。

### 2.4 缓冲区（pbuf）管理模块

unIP 中，所有的 Packet 均用同一种结构来封装，那就是 pbuf。pbuf 是一个链表，用来链接一个 packet 的各个组成部分，类似于 Linux 下的 sk_buff 一样的结构，只是 pbuf 的设计更加节省 RAM。pbuf 结构的声明如下：

```
struct pbuf {
        struct pbuf *next;
        void *payload;          //有效数据的指针
        u16_t tot_len;          //整个 pbuf 链表的有效数据长度总和，以 byte 为单位
        u16_t len;              //本 pbuf 中的有效数据的长度，以 byte 为单位
        u16_t flags;            //本 pbuf 的类型
        u16_t ref;              //本 pbuf 的引用次数
 };
```

具体的 unIP 下的 pbuf 一共有以下三种（根据 flags 来区分）：

- PBUF_ROM： 对应 flag 为 PBUF_FLAG_ROM，此类 pbuf 在分配的时候不会分配任何 RAM 区域给 payload 指针，而 payload 一般应该由用户设定指向一片 ROM 区域中。这种数据结构多半用来保存不变的数据，例如不需要拷贝到协议栈内部的 TCP 数据。PBUF_REF 也属于此类 pbuf，对应的 flag 为 PBUF_FLAG_REF，同的是它指向的数据区域未必是 ROM 区域，也可能是 RAM 中，一般用来引用不同的 pbuf 中的数据。由于并不会分配额外的内存给 payload，因此此类 PBUF 有一个特点就是 pbuf 的大小固定，因此这类 pbuf 可以以模式化内存来分配。用户可以自行配置系统中所需要的此类 pbuf 的数量。一般来说如果有大量的数据位于 ROM 中需要传送的话，这类 pbuf 的数量就应该设置得适当多一些。
- PBUF_RAM：对应于 flag 为 PBUF_FLAG_RAM，此类 pbuf 在分配的时候会一并分配指定大小的 RAM 区域给 payload 指针。因此 payload 一定是指向 RAM 区域。这类 pbuf 用来存放动态生成的 packet，例如动态网页数据，等等。由于大小都是由调用者指定，因此这类 pbuf 不能以模式化来处理，只能完全靠动态内存分配来得到。
- PBUF_POOL :对应于 flag 为 PBUF_FLAG_ POOL，此类 pbuf 的数据指针也是指向一片 RAM 区域，不过所指向区域的大小是预先固定的。此类 pbuf 很显然也可以用模式化内存来分配，pool 的大小以及 pool 的数量都可以由用户在编译期配置好。此类 pbuf 的用途一般是用于存放链路层接收到的 packet，由于链路层的 packet 的长度会有一个不太大的上限值，例如以太网链路层的 Packet 就不会大于 1500Byte（除去以太包头和校验和），因此此 pool 的长度不用超过 750 个 word。当然如果主要是用 TCP 层来做应用，那么 TCP 层的 MSS 设定好以后，此处的 pool size 救只用设置为（MSS＋40）/2 个 word 就可以了。

在清楚了以上三类 pbuf 以后就可以来看看本层提供的 API 函数了，要编写好应用层的协议，这些 API 是需要好好运用的。本层 API 一共有以下这些：pbuf_init()、pbuf_alloc ()、pbuf_realloc()、pbuf_header()、pbuf_ref()、pbuf_free()、pbuf_clen ()、pbuf_chain()和 pbuf_cat()。下面是所有函数的详细说明表。

pbuf_init ()

| | |
|---|---|
| 函数原型 | void pbuf_init(void) |
| 参数 | 无 |
| 返回值 | 无 |
| 功能说明 | 初始化 pbuf 区域<br>必须在所有其它 pbuf 类函数运行前被调用 |

pbuf_alloc ()

| | |
|---|---|
| 函数原型 | struct pbuf *pbuf_alloc(pbuf_layer l, u16_t size, pbuf_flag flag) |
| 参数 | l 枚举类型 pbuf_layer，包括以下数值<br>PBUF_TRANSPORT, 传输层的 packet<br>PBUF_IP, IP 层的 packet<br>PBUF_LINK, 链路层的 packet<br>PBUF_RAW 原始 packet<br>size 申请的 pbuf 管理的数据区域的长度（以字节为单位！）<br>flag 枚举类型 pbuf_flag，包括以下数值<br>PBUF_RAM, 数据区域在 RAM 中 |

| | |
|---|---|
| | PBUF_ROM, 数据区域在 ROM 中（不修改）<br>PBUF_REF, 用来引用其它 packet 的 pbuf<br>PBUF_POOL 在 pbuf_pool 中，为静态内存区域 |
| 返回值 | pbuf 类型指针，或空指针 |
| 功能说明 | 四种枚举类型的唯一区别在于在 payload 指针前预留了不同大小的空闲空间：<br>传输层 20＋20＋14<br>网络层 20＋14<br>链路层 14<br>原始型 0<br>用户使用的是 payload 之后的区域，大小依然是 size。这里的单位均为字节。在应用层编程的时候，基本用不着网络层与链路层的 pbuf，一般纯用户数据就用原始型，而需要利用 UDP 的时候，就需要选择传输层的 pbuf，用户使用的是 payload 后的数据，而 payload 前的数据就预留给封装 packet 之用。 |

pbuf_realloc ()

| | |
|---|---|
| 函数原型 | void pbuf_realloc(struct pbuf *p, u16_t size) |
| 参数 | p 准备调整的 pbuf 类型指针<br>size p 准备调整到的新长度（指数据长度） |
| 返回值 | 无 |
| 功能说明 | 主要是用来去除在分配了不必要的 pbuf 链的没有用到的单元<br>例如，事先分配了一个 pbuf 链<br>P1 -> P2 -> P3<br>后来调整了数据的长度，使得 P3 中的数据没有用到，那么通过调整 P1 的 total_len 域，然后调用本函数，就可以回收 P3 了。 |

pbuf_header ()

| | |
|---|---|
| 函数原型 | u8_t pbuf_header(struct pbuf *p, s16_t header_size) |
| 参数 | p 目标 pbuf 指针<br>header_size 有符号数，header 头的大小。（以字节为单位）。 |
| 返回值 | 成功返回 0，失败返回 1 |
| 功能说明 | 调整 pbuf 的 payload 指针，使 payload 指向原来 payload＋header_size 的位置。<br>当 header_size 为正数的时候，是将 payload 增加，从 packet 中去除一部分数据，例如 tcp 头等等，在拆包的时候运用。<br>当 header_size 为负数的时候，实际上是将 payload 减少，以向 packet 增加一些内容，例如 ip 头等，在封包的时候运用。 |

| | |
|---|---|
| | 当给出非法的 header_size 的时候（例如超出范围或者为奇数等），会返回 0 并保持原来的 payload 指针不变。 |

pbuf_ref ()

| | |
|---|---|
| 函数原型 | void pbuf_ref(struct pbuf *p) |
| 参数 | p 目标 pbuf 指针 |
| 返回值 | 无 |
| 功能说明 | 将目标 pbuf 域的 ref 域加 1<br>意味着被引用次数加一，只有关当 ref 域为 0 的时候，该 pbuf 才可能被释放。 |

pbuf_ref_chain ()

| | |
|---|---|
| 函数原型 | void pbuf_ref_chain (struct pbuf *p) |
| 参数 | p 目标 pbuf 链的首指针 |
| 返回值 | 无 |
| 功能说明 | 将 p 指向的 pbuf 链中所有的 pbuf 的域 ref 加 1<br>意味着该链表下所有单元被引用次数加一 |

pbuf_free ()

| | |
|---|---|
| 函数原型 | u8_t pbuf_free (struct pbuf *p) |
| 参数 | p 目标 pbuf 指针 |
| 返回值 | 8 位整型数 |
| 功能说明 | 将 p 指向的 pbuf 链表中的所有单元的 ref 域减一，若减至零，则释放该单元。<br>返回值为改链表的单元总数。 |

pbuf_cat ()

| | |
|---|---|
| 函数原型 | void pbuf_cat(struct pbuf *h, struct pbuf *t) |
| 参数 | h pbuf 指针<br>t pbuf 指针 |
| 返回值 | 无 |
| 功能说明 | 将两个 pbuf 链表连接起来，成为一个新的 pbuf。新的 pbuf 所包含的数据是两者的顺序联合。<br><br>多用于同层的数据的联合。按照 pbuf_free 的算法，t 与 h 会一同被释放掉。 |

pbuf_chain ()

| 函数原型 | void pbuf_chain(struct pbuf *h, struct pbuf *t) |
|---|---|
| 参数 | h pbuf 指针<br>t pbuf 指针 |
| 返回值 | 无 |
| 功能说明 | 将两个 pbuf 链表连接起来，成为一个新的 pbuf。新的 pbuf 所包含的数据是两者的顺序联合。此外 t 的 ref 域会被加一。<br><br>多用于不同层的数据的联合。h 为低层的 pbuf，而 t 为高层的 pbuf。因为 t 的 ref 值会加一，按照 pbuf_free 的算法，它不会与 h 一同被释放掉，这正好适合分层处理的原则。 |

pbuf_clen ()

| 函数原型 | u8_t pbuf_clen (struct pbuf *p) |
|---|---|
| 参数 | p 目标 pbuf 链的首指针 |
| 返回值 | 无 |
| 功能说明 | 返回值为该链表的单元总数。 |

## 2.5 UDP 层 API 函数说明

unIP 提供了 UDP 层的 API 函数，用户可以用其来编写客户端和服务器端的应用程序，例如 TFTP 以及 DNS Client 和 DHCP Client。下面是本层 API 的详细说明：

udp_init ()

| 函数原型 | udp_init () |
|---|---|
| 参数 | 无 |
| 返回值 | 无 |
| 功能说明 | 本函数初始化 udp 层相关的数据结构，必须在所有 udp 层应用之前被调用。 |

udp_new ()

| 函数原型 | struct udp_pcb * udp_new(void) |
|---|---|
| 参数 | 无 |
| 返回值 | UDP 协议控制块指针，若无空闲协议控制块，则返回 NULL。<br>UDP 协议控制块的个数，可由 UDP_NUM 设定。 |
| 功能说明 | 本函数初始化 udp 层相关的数据结构，必须在所有 udp 层应用之前被调用。 |

udp_remove ()

| 函数原型 | void udp_remove(struct udp_pcb *pcb) |
|---|---|

| 参数 | UDP 协议控制块指针 |
|---|---|
| 返回值 | 无 |
| 功能说明 | 删除并释放该 udp 控制块 |

udp_bind ()

| 函数原型 | err_t udp_bind(struct udp_pcb *pcb, struct ip_addr *ipaddr, u16_t port) |
|---|---|
| 参数 | pcb<br>UDP 协议控制块指针 需要绑定的协议控制块<br>ipaddr<br>绑定到本地的 ip 地址，与某个网络接口匹配，若为 IP_ADDR_ANY，则是绑定到本地的所有网络接口<br>port<br>绑定到本地的端口，若为 0 则表示可随机选取一个端口 |
| 返回值 | 成功则返回 ERR_OK<br>出错可能返回 ERR_USE，表示无可用端口。 |
| 功能说明 | 将 udp 块绑定到本地 ip 地址以及 port 端口号。 |

udp_connect ()

| 函数原型 | err_t udp_connect (struct udp_pcb *pcb, struct ip_addr *ipaddr, u16_t port); |
|---|---|
| 参数 | pcb<br>UDP 协议控制块指针，需要连接的协议控制块<br>ipaddr<br>remote host 的 ip 地址<br>port<br>remote host 的端口号 |
| 返回值 | 目前一定返回 ERR_OK |
| 功能说明 | 本函数并不发起实际的网络通信，只是将 pcb 的 remote ip 和 remote port 设定为参数中的值。 |

udp_send ()

| 函数原型 | err_t udp_send (struct udp_pcb *pcb, struct pbuf *p); |
|---|---|
| 参数 | pcb<br>UDP 协议控制块指针，需要连接的协议控制块<br>p<br>存放需要发送的应用层数据的 pbuf 指针，此 pbuf 应该为传输层 pbuf。 |
| 返回值 | 目前一定返回 ERR_OK |

| | |
|---|---|
| 功能说明 | 发送 p 中所包含的数据到 UDP 层，最终由链路层送出到网路上。UDP 层发送数据就是通过本函数进行的，入口参数 p 必须申请为传输层 pbuf，否则会出错。 |
| 示例 | ……　//一系列初始化过程<br>p = pbuf_alloc(PBUF_TRANSPORT, sizeof(struct nserver), PBUF_RAM);<br>……..　//填充 p 的 payload 域的数据<br>upcb = udp_new();<br>udp_connect(upcb, (struct ip_addr *)&nserv, NS_PORT);//连接 dns 服务器<br>udp_send(upcb, p);<br>pbuf_free(p);<br>…… |

udp_recv ()

| | |
|---|---|
| 函数原型 | void　udp_recv　　(struct udp_pcb *pcb,<br>void (* recv)　(void *arg, struct udp_pcb *upcb,<br>struct pbuf *p,<br>struct ip_addr *addr,<br>u16_t port),<br>void *recv_arg); |
| 参数 | pcb<br>UDP 协议控制块指针　　需要绑定的协议控制块<br>recv<br>回调函数指针。<br>recv_arg<br>回调参数 |
| 返回值 | 成功则返回 ERR_OK<br>出错可能返回 ERR_USE，表示无可用端口。 |
| 功能说明 | 设定回调函数指针 recv.以及回调参数 recv_arg<br>该函数指针所指向的函数会在有新的 udp 层数据到达的时候被调用。 |

recv ()

| | |
|---|---|
| 函数原型 | void (* recv)　(void *arg,<br>struct udp_pcb *upcb,<br>struct pbuf *p,<br>struct ip_addr *addr,<br>u16_t port) |

| | |
|---|---|
| 参数 | arg<br>回调参数<br>upcb<br>UDP 协议控制块指针<br>p<br>存放接收到的数据的 pbuf 指针<br>addr<br>remote host 的 ip 地址<br>port<br>remote host 的端口号 |
| 返回值 | 成功则返回 ERR_OK，继续这次 UDP 通信。<br>若出错则可以返回相应的错误编码，以中止 UDP 通信。 |
| 功能说明 | 本函数当 udp 层有新的数据到来的时候会被调用。新的数据就在 p 中存放，addr 和 port 为 remote ip addr 和 remote port。udp 层的协议的接收处理部分就在此函数中进行。利用 UDP 层实现的应用层协议的实现，主体部分一般在 recv 函数中进行。 |

2.6 TCP 层 API 函数说明

unIP 具有完整的 TCP 层的实现，利用本层提供的 API 可以实现各种基于 TCP 协议的应用层协议，例如 HTTP，POP3 和 SMTP 等等。下面是本层 API 函数的详细说明。

tcp_init ()

| | |
|---|---|
| 函数原型 | void tcp_init (void); |
| 参数 | 无 |
| 返回值 | 无 |
| 功能说明 | tcp 初始化函数，应该在所有的 tcp 应用之前被调用，初始化 tcp 模块。 |

tcp_new ()

| | |
|---|---|
| 函数原型 | struct tcp_pcb * tcp_new (void); |
| 参数 | 无 |
| 返回值 | 若成功则 TCP 协议控制块指针<br>若无空闲 TCP 协议控制块，则返回 NULL。系统可用 TCP 协议控制块的个数，可以通过宏 TCP_NUM 配置。 |
| 功能说明 | 新建 TCP 协议控制块；新建一个连接之前调用此函数获取一个可用的协议控制块。此函数给协议控制块默认优先级为 TCP_PRIO_NORMAL ＝ 64 |

tcp_alloc ()

| | |
|---|---|
| 函数原型 | struct tcp_pcb * tcp_alloc (u8_t prio); |
| 参数 | prio 协议控制块的优先级 |
| 返回值 | 若成功则 TCP 协议控制块指针<br>若无空闲 TCP 协议控制块，则返回 NULL。系统可用 TCP 协议控制块的个数，可以通过宏 TCP_NUM 配置。 |
| 功能说明 | 功能同 tcp_new()，额外的功能是可以指定 PCB 的优先级。 |

tcp_arg ()

| | |
|---|---|
| 函数原型 | void tcp_arg (struct tcp_pcb *pcb, void *arg); |
| 参数 | pcb<br>TCP 协议控制块指针<br>arg<br>回调参数 |
| 返回值 | 无 |
| 功能说明 | 设置有 PCB 所确定的 TCP 连接中使用的回调参数。该参数会在该 PCB 块对应的回调函数中被引用。 |

tcp_bind ()

| | |
|---|---|
| 函数原型 | err_t tcp_bind (struct tcp_pcb *pcb, struct ip_addr *ipaddr,<br>u16_t port); |
| 参数 | pcb<br>TCP 协议控制块指针<br>ipaddr<br>需要绑定的本地网络接口的 ip 地址<br>port 绑定的本地端口，若为 0 则由协议栈随机选取一个端口。 |
| 返回值 | 成功则返回 ERR_OK<br>出错可能返回 ERR_USE，表示端口已经被别的 pcb 绑定。 |
| 功能说明 | 将指定 pcb 绑定到本地的 ip 地址和 port 端口号；若 ipaddr 若为 IP_ADDR_ANY，则是绑定到本地所有 ip 地址（当有多网络接口时，就是绑定到所有的接口）。<br>若有其它 pcb 已经绑定该 ip 以及 port 口，则返回 ERR_USE，否则返回 ERR_OK 表示绑定成功 |

tcp_listen ()

| | |
|---|---|
| 函数原型 | struct tcp_pcb * tcp_listen (struct tcp_pcb *pcb); |
| 参数 | pcb |

| | |
|---|---|
| | TCP 协议控制块指针 |
| 返回值 | 成功则返回 pcb 指针（实际为 listen pcb *） |
| 功能说明 | 将 pcb 所指定的连接置于监听队列中。在此操作之前，必须已经执行完绑定动作。置于监听队列以后，若有客户端连接已经绑定的端口，则会调用接下来由函数 tcp_accept()所指定的回调函数，来服务该连接。此函数是用来实现 tcp 服务器程序的。<br><br>由于处于 listen 状态的 tcp 协议控制块所需的 ram 较小，因此本函数将释放入口的 pcb，并另外申请一个小一些的 pcb 以供监听状态之用。若 mem 过少，则可能返回 NULL。 |

tcp_accept ()

| | |
|---|---|
| 函数原型 | void tcp_accept (struct tcp_pcb *pcb,<br>err_t (* accept)(void *arg, struct tcp_pcb *newpcb,<br>err_t err)); |
| 参数 | pcb<br>TCP 协议控制块指针<br>accept<br>接受连接回调函数指针 |
| 返回值 | 无 |
| 功能说明 | 设定接受连接回调函数。<br>当有 client 连接到监听 pcb 中的相应端口时，accept 所指向的函数将被调用。 |

accept ()

| | |
|---|---|
| 函数原型 | err_t (* accept)(void *arg, struct tcp_pcb *newpcb,<br>err_t err)); |
| 参数 | arg<br>回调参数<br>newpcb<br>维护此连接的协议控制块<br>err<br>连接过程中的错误码 |
| 返回值 | 无 |
| 功能说明 | 回调函数（函数名称可以由程序员自己指定）：当客户端发起连接请求时会被调用。<br><br>此函数在 tcp 连接由 SYN_RECV 状态转换到 ESTABLISED 之间被调用。<br>应用层可以在此函数中做一些初始化动作以及 filter 的动作，若通过所有检查则返回 ERR_OK，通知协议栈接受此连接。否则返回其它错误码，通知协议栈重置客户端连 |

| | |
|---|---|
| | 接（RST）。 |

tcp_connect ()

| | |
|---|---|
| 函数原型 | err_t tcp_connect (struct tcp_pcb *pcb, struct ip_addr *ipaddr,<br>u16_t port, err_t (* connected)(void *arg,<br>struct tcp_pcb *tpcb,<br>err_t err)); |
| 参数 | pcb<br>TCP 协议控制块指针<br>ipaddr<br>remote host 的 ip 地址<br>port<br>remote host 的端口号<br>connected<br>连接建立回调函数指针。 |
| 返回值 | 操作成功返回 ERR_OK，否则返回错误码 |
| 功能说明 | 建立到服务器的一个连接。并指定一个回调函数指针 connected，当连接被服务器接受，连接最终建立的时候，协议栈会自动调用该指针所指向的函数。<br>本函数执行中会送出一个 SYN，并不会等服务器回应就立刻返回，后续的动作是在协议栈中完成的，直到连接建立的时候，应用层才有机会在 connected 函数中做一些操作。<br>本函数执行成功后会返回 ERR_OK，若返回 ERR_MEM 则表示没有 RAM 来发送 SYN。<br>本函数用来实现客户端对服务器端发起一次连接。 |

connected ()

| | |
|---|---|
| 函数原型 | err_t (* connected)(void *arg,<br>struct tcp_pcb *tpcb,<br>err_t err)); |
| 参数 | arg<br>回调函数指针<br>tpcb<br>TCP 协议控制块指针<br>err<br>连接建立过程中的错误码 |
| 返回值 | 操作成功返回 ERR_OK，否则返回错误码 |

| | |
|---|---|
| 功能说明 | 回调函数（函数名称可以由程序员自己指定）：当收到服务器的 SYN ACK 后会调用此函数。<br>此函数在 tcp 由 SYN Sent 状态转换到 ESTABLISED 之间被调用，让应用层可以处理一个刚被服务器接受的一个连接。入口参数 err 代表了是否是一次正确地连接建立。 |

tcp_write ()

| | |
|---|---|
| 函数原型 | err_t tcp_write (struct tcp_pcb *pcb, const void *dataptr, u16_t len,<br>u8_t copy); |
| 参数 | pcb<br>TCP 协议控制块指针<br>dataptr<br>待发送数据首地址<br>len<br>待发送数据的长度（以字节为单位）<br>copy<br>0 不拷贝到协议栈内部发送<br>1 拷贝到协议栈内部发送 |
| 返回值 | 操作成功返回 ERR_OK，否则返回错误码 |
| 功能说明 | 应用层传输 TCP 数据函数。<br>此函数一般在 ESTABLISED 状态下调用。用来发送以 dataptr 为首地址，长度为 len(byte length)的数据。copy 为 0 表示不需要在协议栈内部拷贝一份该数据，多用于发送 rom 或不会变动的静态数据。若是动态数据则 copy 应该置为 1，让协议栈拷贝一份该数据发送。<br>本函数返回 ERR_OK 则表示数据成功插入到发送队列之中，否则表示插入失败，一般会返回 ERR_MEM。 |

tcp_sent ()

| | |
|---|---|
| 函数原型 | void tcp_sent (struct tcp_pcb *pcb,<br>err_t (* sent)(void *arg, struct tcp_pcb *tpcb,<br>u16_t len)); |
| 参数 | pcb<br>TCP 协议控制块指针<br>sent<br>发送数据回调函数 |
| 返回值 | 无 |
| 功能说明 | 设定回调函数指针 sent<br>此函数设定回调函数 sent。 |

| | |
|---|---|
| | 函数指针 sent 指向的函数将在 tcp 层得知有新的数据被 remote 端收到的时候被调用。用来继续发送应用层未发送完毕的数据。 |

sent ()

| | |
|---|---|
| 函数原型 | err_t (* sent)(void *arg, struct tcp_pcb *tpcb,<br>u16_t len)； |
| 参数 | arg<br>回调参数<br>tpcb<br>TCP 协议控制块指针<br>len<br>remote host 新收到的数据长度，以字节为单位 |
| 返回值 | 操作成功返回 ERR_OK，让连接继续。<br>否则返回错误码，通知协议栈中止该连接，并释放资源。 |
| 功能说明 | 此回调函数在 remote host 收到新的数据后会被调用。len 表示新被收到的字节数。此函数用来续发应用层未发送完的数据。 |

tcp_recv ()

| | |
|---|---|
| 函数原型 | void tcp_recv (struct tcp_pcb *pcb,<br>err_t (* recv)(void *arg, struct tcp_pcb *tpcb,<br>struct pbuf *p, err_t err)); |
| 参数 | pcb<br>TCP 协议控制块指针<br>recv<br>接受数据回调函数指针 |
| 返回值 | 无 |
| 功能说明 | 设定回调函数 recv。该函数在收到新的数据后会被调用。 |

recv ()

| | |
|---|---|
| 函数原型 | err_t (* recv)(void *arg, struct tcp_pcb *tpcb,<br>struct pbuf *p, err_t err); |
| 参数 | arg<br>回调参数<br>tpcb<br>TCP 协议控制块指针<br>p<br>存放收到的数据的 pbuf |

| | |
|---|---|
| | err<br>协议栈内部抛出的错误编码 |
| 返回值 | 一切操作成功则返回 ERR_OK，让连接继续进行。<br>否则返回错误码通知协议栈中止连接。 |
| 功能说明 | 本回调函数用来接收新的数据。应用层协议主体在本函数中实现，所有的应用层输入数据均来源于此函数中 pbuf 指针 p。处理完毕后返回 ERR_OK 表示继续连接，否则协议栈会终止连接。 |

tcp_recved ()

| | |
|---|---|
| 函数原型 | void tcp_recved (struct tcp_pcb *pcb, u16_t len); |
| 参数 | pcb<br>TCP 协议控制块指针<br>len<br>应用层收到的数据长度，以字节为单位 |
| 返回值 | 无 |
| 功能说明 | 应用层用来通知底层协议成功收到的字节数，在接收到新的数据后必须调用此函数，否则窗口将不会更新，窗口将很快塞满而无法接收新的数据。 |

tcp_poll ()

| | |
|---|---|
| 函数原型 | void tcp_poll (struct tcp_pcb *pcb,<br>err_t (* poll)(void *arg, struct tcp_pcb *tpcb),<br>u8_t interval); |
| 参数 | pcb<br>TCP 协议控制块指针<br>poll<br>轮询回调函数指针<br>interval<br>轮询周期 |
| 返回值 | 无 |
| 功能说明 | 设定一个周期调用的函数指针 poll。周期典型值为 interval *0.5secs。 |

poll ()

| | |
|---|---|
| 函数原型 | err_t (* poll)(void *arg, struct tcp_pcb *tpcb),<br>u8_t interval); |
| 参数 | arg<br>回调参数 |

| | |
|---|---|
| | tpcb<br>TCP 协议控制块指针<br>poll<br>轮询回调函数指针 |
| 返回值 | 无 |
| 功能说明 | 位于应用层的周期调用的函数，当一次连接长期处于 idle 状态时，此函数可以用来处理一些事务，例如超时断线回收资源等等。 |

tcp_err ()

| | |
|---|---|
| 函数原型 | void tcp_err (struct tcp_pcb *pcb,<br>void (* err)(void *arg, err_t err)); |
| 参数 | pcb<br>TCP 协议控制块指针<br>err<br>错误处理函数指针 |
| 返回值 | 无 |
| 功能说明 | 设定错误处理回调函数，当连接发生某些错误的时候会调用，例如接收到 RST 或者发出 RST 等，用户应该在此函数中进行一些资源回收的处理，因为此函数被调用意味着该连接将马上被删除，若此时不回收一些资源将会造成资源泄漏。 |

err ()

| | |
|---|---|
| 函数原型 | void (* err)(void *arg, err_t err)); |
| 参数 | arg<br>回调参数<br>err<br>协议栈扔出的错误编码 |
| 返回值 | 无 |
| 功能说明 | 错误处理回调函数，当连接发生某些错误的时候会调用，例如接收到 RST 或者发出 RST 等，用户应该在此函数中进行一些资源回收的处理，因为此函数被调用意味着该连接将马上被删除，若此时不回收一些资源将会造成资源泄漏。 |

tcp_close ()

| | |
|---|---|
| 函数原型 | err_t tcp_close (struct tcp_pcb *pcb); |
| 参数 | pcb<br>TCP 协议控制块 |
| 返回值 | 操作成功返回 ERR_OK |

| | |
|---|---|
| 功能说明 | 关闭连接。该函数执行成功后会发出 FIN 到 remote host，并释放 pcb，返回 ERR_OK。否则会返回 ERR_MEM，程序员要负责处理这种情形，例如可以轮询一直到返回成功，否则就没有成功关闭该连接。 |

tcp_abort ()

| | |
|---|---|
| 函数原型 | void tcp_abort (struct tcp_pcb *pcb); |
| 参数 | pcb<br>TCP 协议控制块 |
| 返回值 | 无 |
| 功能说明 | 中止连接。当发现状态出错的时候，或者重试一定次数都无法恢复的时候，就执行次函数中止连接。此函数执行成功会发出 RST 到 remote host。本函数执行到最后会调用错误处理回调函数。 |

tcp_tmr ()

| | |
|---|---|
| 函数原型 | void tcp_tmr (void); |
| 参数 | 无 |
| 返回值 | 无 |
| 功能说明 | tcp 节拍函数，应该保证每 200ms 调用一次本函数。 |

## 2.7 网络字节辅助函数 API

由于网络字节序是 big endian，而 UnSP 单片机为 little endian，所以在从网络字节流中获取数据的以及送数据到网络链路中去的时候都应该做一下转换。还有，UnSP 单片机的一个字节在物理上是 16bit 的数据，而网络字节流的单位为字节，因此也不能直接利用数组下标的模式来取数，而需要用别的方法来处理。本节就是介绍用于处理以上两类情形的 API 函数。下面就是详细说明：

htons()

| | |
|---|---|
| 函数原型 | u16_t htons(u16_t n) |
| 参数 | n 16bit 数据 |
| 返回值 | 16bit 数据 |
| 功能说明 | 将本地数据转换为网络字节序，在发送 16bit 数据之前应该做此转换。<br>高低 8bit 数据互换了<br><br>例如 n ＝ 0x1234 |

| | |
|---|---|
| | m ＝ htons(n)<br><br>执行完毕后 m 为 0x3412 |

ntohs()

| | |
|---|---|
| 函数原型 | u16_t ntohs(u16_t n) |
| 参数 | n 16bit 数据 |
| 返回值 | 16bit 数据 |
| 功能说明 | 将网络字节序转换为本地字节序，在接收 16bit 数据之前应该做此转换。<br>高低 8bit 数据互换了<br><br>例如 n ＝ 0x1234<br>m ＝ htons(n)<br><br>执行完毕后 m 为 0x3412 |

htonl ()

| | |
|---|---|
| 函数原型 | u32_t htonl (u32_t n) |
| 参数 | n 32bit 数据 |
| 返回值 | 32bit 数据 |
| 功能说明 | 将本地数据转换为网络字节序，在发送 32bit 数据之前应该做此转换。<br>高低 8bit 数据互换了<br><br>例如 n ＝ 0x12 34 56 78<br>m ＝ htons(n)<br><br>执行完毕后 m 为 0x78 56 34 12 |

ntohl ()

| | |
|---|---|
| 函数原型 | u32_t htonl (u32_t n) |
| 参数 | n 32bit 数据 |
| 返回值 | 32bit 数据 |
| 功能说明 | 将网络字节序转换为本地字节序，在接收 32bit 数据之前应该做此转换。<br>高低 8bit 数据互换了<br><br>例如 n ＝ 0x12 34 56 78<br>m ＝ htons(n) |

| | |
|---|---|
| | 执行完毕后 m 为 0x78 56 34 12 |

getbyte ()

| | |
|---|---|
| 函数原型 | u8_t getbyte(u16_t *data, u16_t offset) |
| 参数 | data<br>缓冲区首地址指针<br>offset<br>需要抽取的字节对应的偏移（以字节为单位） |
| 返回值 | 相应位置处的 8bit 数据 |
| 功能说明 | 从以缓冲区为首地址，偏移量为 offset 个字节的位置处，取出该 byte 数据。<br>若 offset 为偶数，则 offset 处的数据为 data [offset/2]的低 8 位数据<br>若 offset 为奇数，则 offset 处的数据为 data [offset/2]的高 8 位数据<br><br>例子：<br>addr ： content<br>0x0000 ： 0x1234 0x5678 0xabcd 0xcdef 0x 1374 0x1567 0x3478<br>offet 为 0 对应的数据为 0x34，<br>offset 为 5 对应的数据为 0xab<br>offset 为 10 对应的数据为 0x67 |

putbyte ()

| | |
|---|---|
| 函数原型 | void putbyte(u16_t *data, u8_t ch, u16_t offset) |
| 参数 | data<br>缓冲区首地址指针<br>ch<br>待写入的数据<br>offset<br>需要抽取的字节对应的偏移（以字节为单位） |
| 返回值 | 无 |
| 功能说明 | 从以缓冲区为首地址，偏移量为 offset 个字节的位置处，写入数据 ch。<br>offset 与数据的对应关系同 getbyte |

packstrncat ()

| | |
|---|---|
| 函数原型 | u16_t packstrncat(u16_t *data, char *str, u16_t offset) |
| 参数 | data<br>缓冲区首地址指针 |

| | |
|---|---|
| | str<br>待加入的非压缩型字符串首地址<br>offset<br>待加入的 data 缓冲区的末端偏移量 |
| 返回值 | 操作完成后新的 data 的末端偏移量 |
| 功能说明 | 从以缓冲区 data 为首地址，偏移量为 offset 个字节的位置处，将非压缩型字符串 str 中的字符全部存入 data 中，得到一个新的压缩型字符串。str 必须以’\0’结束，’\0’并不会被写入 data 缓冲区的末尾。 |

unIP_timeout()

| | |
|---|---|
| 函数原型 | u16_t unIP_timeout(void) |
| 参数 | 无 |
| 返回值 | unIP 超时事件标志<br>bit0 tcp 超时<br>bit1 ARP 超时<br>bit2 DHCP FINE 时钟超时<br>bit3 DHCP COARSE 时钟超时 |
| 功能说明 | 通过调用 unIP，得知某超时事件发生，然后执行相应操作 |
| 例子 | u16_t iTemp = lwip_timeout();<br>if (iTemp & DHCP_COARSE_TIMEOUT) dhcp_coarse_tmr();<br>if (iTemp & DHCP_FINE_TIMEOUT) dhcp_fine_tmr();<br>if(iTemp & TCP_TIMEOUT) tcp_tmr();<br><br>本例假设已经启动 DHCP 程序。 |

## 3 unIP 参数配置说明

在第 2 节中已经有许多 API 函数的说明中提到了协议栈的配置事项，在这里我们将这些配置整理一下，一方便用户定制自己的协议栈。目前版本 unIP 的配置功能基本上集中在配置不同的对 RAM 消耗的选项，而不能配置 CODE 不同的选项，今后应该会突破此限制。下面是所有的可配置参数，均在 config.h 中配置。

表2 unIP 库的配置说明表

| 宏名 | 最小值 | 典型值 | 最大值 | 说明 |
|---|---|---|---|---|
| MEM_SIZE | 600 | 800 | | 动态内存的总数，如果系统有许多的动态数据，则此项应该设置得大一些，但是最大的大小受限于总得 RAM 大小 |
| POOL_NUM | 1 | 1 | | POOL 型 Pbuf 得数量，因为系统一般只会一次处理一个输入得 packet 因此，此处设置为 1 就可以了。 |
| POOL_SIZE | (TCP_MSS+54)/2 | 300 | | POOL 的大小，以字为单位，此 POOL 的大小应该是能容纳链路层最大的 packet，由于 unIP 暂时不支持 IP 分片，因此为了能正确接收以太网上所有的 packet，此大小理论上应该设置为 1500/2。但是由于 SPCE061 资源过小，而 WEB SERVER 又是利用的 TCP 层，TCP 层有自己的 mss，因此可以设置此处大小为 (TCP_MSS+54)/2 即可接收到所有的 TCP packet。 |
| ARP_NUM | 1 | 3 | | 存放 ARP 记录的表的大小 |
| DNS_NUM | 1 | 3 | | 存放 DNS 记录的表的大小（若没有使用 DNS 功能则此项无效） |
| TCP_MSS | 128 | 128 | | TCP 层 MSS 值 |
| TCP_WINDOW | TCP_MSS | TCP_MSS*2 | | TCP 窗口大小，因为 MSS 的整数倍 |
| TCP_MAXRTX | 4 | 4 | | TCP 普通 packet 最大重传次数 |
| TCP_SYNMAXRTX | 4 | 12 | | TCP SYN 最大重传次数 |
| PBUF_NUM | 5 | 5 | | PBUF_ROM 的个数，如果需要发送大量的静态区域的数据，此 pbuf 应该设置得大一些 |
| UDP_NUM | 1 | 3 | | 能同时存在 UDP 通信个数（若库是没有不带 UDP 的库，则此项无效） |
| TCP_NUM | 1 | 2 | | 能同时建立得 TCP 连接数（若 |

| | | | | |
|---|---|---|---|---|
| | | | | 库是不带 TCP 的库则此项无效） |
| LTCP_NUM | 1 | 1 | | 能同时处于监听状态得 TCP 个数，一个服务器应用对应一个 LTCP。(若库是不带 TCP 的库则此项无效） |
| TCP_SEG_NUM | 5 | 5 | | TCP 层发送缓冲区的 packet 队列的最大长度。传送过程中存在大量的小的 packet，则此项应该设置得大一些。（若库是不带 TCP 的库则此项无效） |

# 4 协议栈应用举例

在上一节中，本手册列举出了 unIP 所有的 API 接口函数，以及它们的用法，在本节中，将会以在 SPCE061 上一个 Web Server 程序的实现，来介绍部分 API 的组合运用。本节可以作为利用 unIP 编写 TCP 服务器程序的一个范例。为了先建立一个感性认识，在本节的第一小节中，我们会引导用户进行一步步的创建一次自己的 WEB SERVER 的全过程，通过编译连接运行，最后可以看到可以从 IE 中访问到自己存放在 SPCE061 上的网页，自制不同的网页，可以实现通过 IO 口控制一个 LED 灯的亮灭（需要用户自己动手去实现）。在第二小节中，我会详细叙述 HTTP 协议实现的细节，用户可以改写部分程序以支持更多的 HTTP 请求，例如 POST 等。

## 4.1 Step by Step 建立自己的 WEB SERVER。

### 4.1.1 网页的制作

用户可以自己制作网页（利用各种网页制作工具），也可以拿本手册提供的一个例子作为模板进行编辑即可。网页制作好以后，例如本手册的例子中在 webfile 目录下存放着所有的网页文件，用户可以直接通过 IE 浏览这些网页，我们接下来的工作，是要把它们塞进 SPCE061 中去，并且，也要在 IE 上访问到它们。

第一步就是将网页数据转换为 c 语言文件，运行本手册附带的工具 NetPageToC，选中我们需要转换的所有网页文件（webfile 目录下），然后点击 Convert，生成文件 fsdata.c，用这个 c 文件覆盖原目录下的 fsdata.c。至此，我们的网页制作工作就算大功告成。

### 4.1.2 HTTP 请求的处理以及动态效果的实现

打开文件

4.1.3 运行 WEB SERVER

打开工程文件，选择 rebuild all，然后点击下载，运行。

回头来配置一下网络参数，WEB SERVER 的默认配置为

IP 地址 ：172.20.8.114

子网掩码：255.255.0.0

网关： 172.20.10.254

与 PC 机的最方便的连接方式是双机对连，需要一根对连的网线，将 DM9000 上的 RJ45 接口与 PC 上的网卡连接起来。将 PC 机的 IP 地址设为 172.20.10.254，子网掩码设为 255.255.0.0，网关设为 172.20.10.254。打开 IE，在 IE 中选择不用代理服务器。然后在 IE 地址栏输入：

HTTP://172.20.8.114

回车即可访问到运行于 SPCE061 上的网页了。

4.2 Web Server 的实现

本 Web Server 的实现的硬件平台是 SPCE061 ＋ DM9000，本节的重点也将不放在硬件电路的实现以及 MAC 驱动层的实现之上，而是放在如何利用 TCP 层 API 以及 unIP 的其它 API 来实现一个 Web Server。对硬件的细节以及驱动的细节想要进一步了解的可以参考我们提供的硬件原理图以及驱动程序的源代码：ethernet.c。

Web Server 的主程序流程如图 1，2 所示：

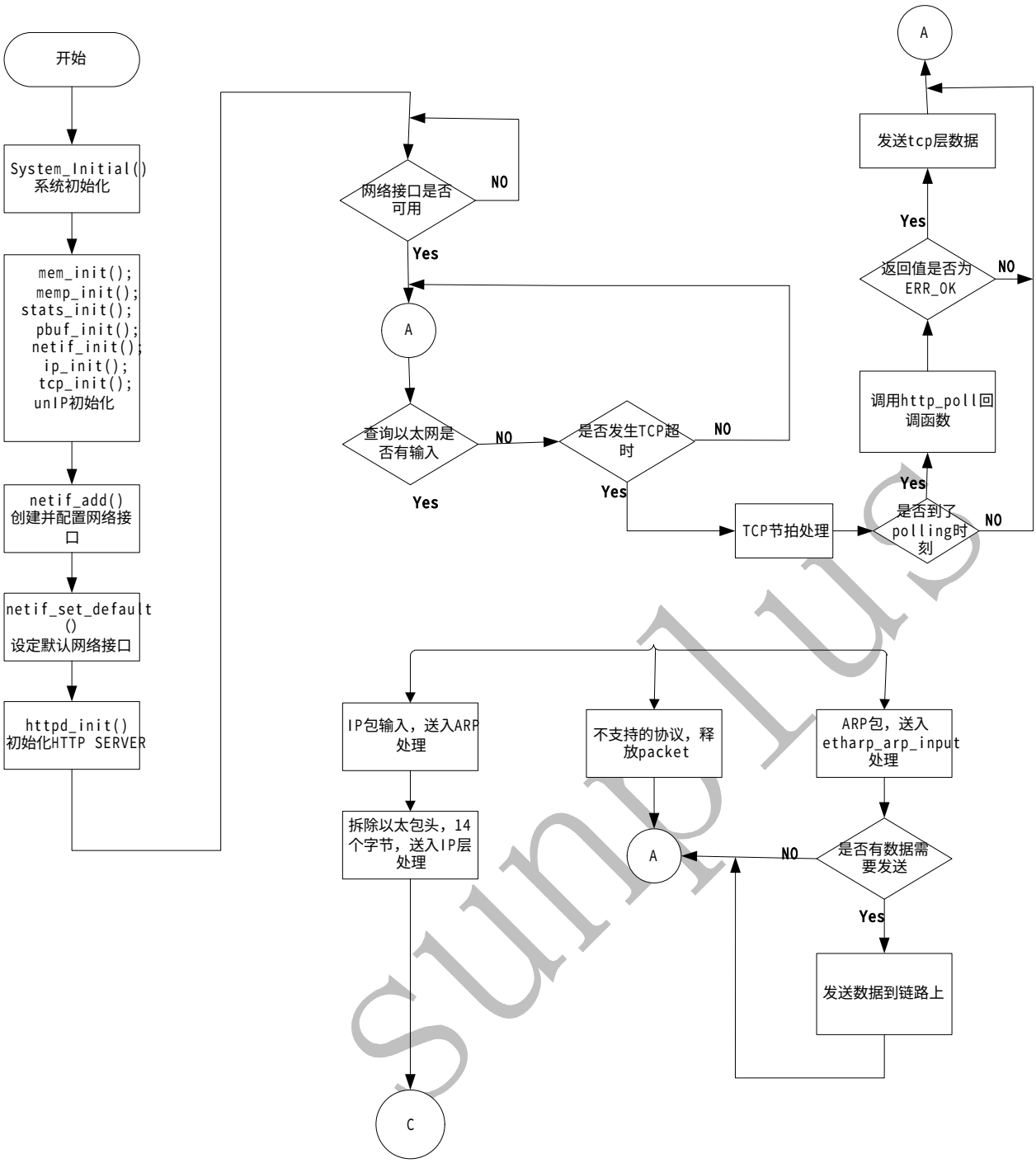


**图 1　Web Server 主程序流程图**

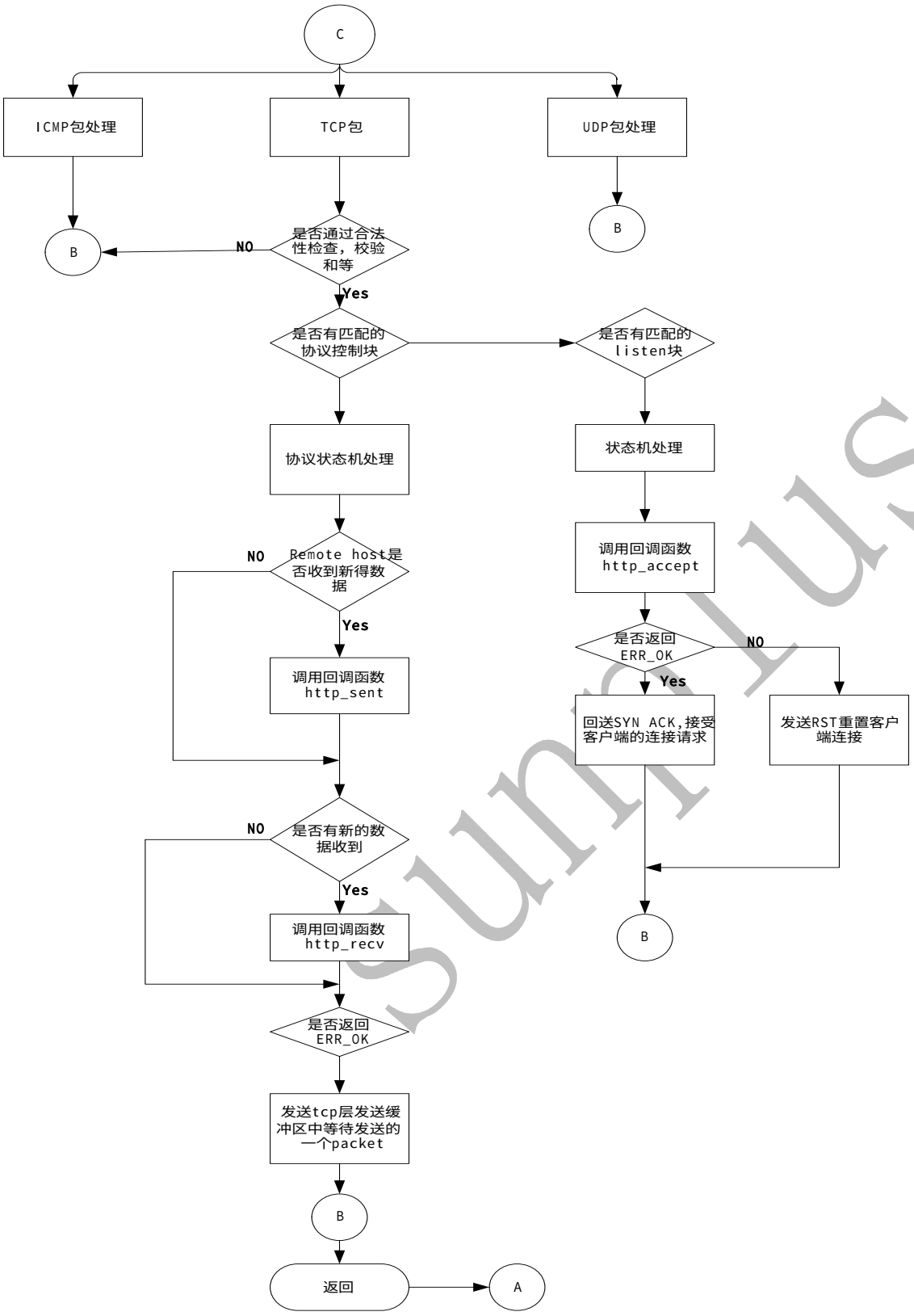


**图 2　Web Server 协议栈处理流程图**

以上两副流程图可以看出整个 Web Server 工作的一个大概流程，用户应该可以从中找出几个回调函数被调用的时机，这一点对于理解 Web Server 的具体实现很重要。下面详细介绍 Web Server 各个部分的具体实现。

### 4.2.1 Web Server 的初始化

在图 1 中所示的 httpd_init()函数完成 Web Server 的初始化动作。下面是该函数的源代码：

```
void
httpd_init(void)
{
   struct tcp_pcb *pcb;
   pcb = tcp_new();
   if (pcb==NULL) return;
   tcp_bind(pcb, IP_ADDR_ANY, 80);
   pcb = tcp_listen(pcb);
//注意，必须用 tcp_listen(pcb)的返回值替换之前的 pcb
//因为在函数 tcp_listen 中已经释放了原 pcb。

   tcp_accept(pcb, http_accept);   //设定 accept 回调函数指针

}
```

执行完这里的初始化过程后，就已经有一个 tcp 控制块被放入监听队列中，只要有客户端的连接请求到来，就会调用最后设定的回调函数 http_accept，用户也可一对应图 2 找到 http_accept 的位置，思考一下它的调用时机为何。

### 4.2.2 接受客户端的请求

当客户端对 80 端口发起连接请求后，http_accept 就会被调用来服务这次连接，下面就是 http_accept 的程序清单：

```
err_t
http_accept(void *arg, struct tcp_pcb *pcb, err_t err)
{
   struct http_state *hs;                          //HTTP 协议控制块
   hs = mem_malloc(sizeof(struct http_state));
   if(hs == NULL) {
      return ERR_MEM;
   }
   hs->retries = 0;                                //
```

```
    hs->web = NULL;
    tcp_arg(pcb, hs);                    //将 HTTP 协议控制块作为回调参数
    tcp_recv(pcb, http_recv);            //设置 http_recv 为 recv 回调函数指针
    tcp_err(pcb, conn_err);              //设置 conn_err 为错误处理回调函数
    tcp_poll(pcb, http_poll, 10);        //设置 http_poll 为轮询回调函数
    return ERR_OK;                       //返回 ERR_OK，接受此连接请求
}
```

此处要说明一下 HTTP 协议控制块的结构，struct http_state 声明如下：

```
struct http_state {
    struct webdata *web;     //此结构中存放 Web page 的所有数据
    u8_t    retries;         //表示重传次数
};
```

其中存放网页的数据结构声明如下：

```
struct webdata {
        struct webdata *next; //形成链表
        u16_t *payload; //本单元网页数据指针
        u16_t offset;     //已经发送除去的数据长度（以字节为单位）
        u16_t len;        //本单元总长度
        u16_t attr;       //属性，DYNAMIC  表示网页数据位于 RAM 中 \
                                STATIC  表示网页数据位于 ROM 中
        };
```

http_accept 函数执行完毕以后，连接就顺利的建立起来了，并且也设定了三个回调函数 http_recv、http_poll 和 conn_err。前两个函数的调用时机在图 1 与图 2 中可以看出来，conn_err 的调用时机是在接收到 RST 后或者发出 RST 时会被调用。正常情况下，http_accept 执行完毕系统就开始等候后续的客户端的 HTTP 请求，类似于 GET HTTP：//www.test.com 这样的数据。一旦有这样的数据送达服务器，则函数 http_recv 会被调用，用来分析客户端的页面请求，并给出正确的回应。

### 4.2.3 页面请求的处理（接收客户端的数据）以及页面数据的传送

本小节是分析客户端的页面请求，并回送相应的页面数据给客户端（假设为 IE）。这个过程是 HTTP 协议处理的主要过程，正确完成后，在 IE 上就会出现我们期望的页面。http_accep 函数分析客户的请求，并且准备好相应的页面数据到 HTTP 协议控制块的 web 域，然后通过函数 send_data 来发送一次 web 域中包含的网页数据。由于 send_data 一次最多可以发送系统设置的 TCP 发送缓冲区大小的内容，而网页数据的长度很可能会大于这个缓冲区的大小，因此网页数据需要续传，而续传的最佳时机就是发送缓冲区有空余的时候。由 unIP 协议栈的处理过程可以知道，当有新的数据被 remote host 确认的时候，就会回收一部分发送缓冲区，因此这个时候进行续传最为合适。根据图 2 可以看到，这个时机，协议栈会调用回调函数 http_sent，因此续传的动作应该在 http_sent 中完成。

下面是 http_recv 函数的清单：

```
err_t
http_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err)
{
   int i;
   u16_t *data;
   struct fs_file file;
   struct http_state *hs;
    u16_t * tempdata;
   hs = arg;
   if(err == ERR_OK && p != NULL) {
      if(hs->web == NULL) {                    //请求页面的开始
       tempdata = mem_malloc(58);
        data = p->payload;
        for(i=0;i<50;i++)                      //拷贝请求字符串的前 50 个字符到非压缩区域
        tempdata[i] = getbyte(data, i);
        pbuf_free(p);                          //释放接收的数据 pbuf
        if(strncmp(tempdata, "GET ", 4) == 0) {//判断是否 GET  请求
      for(i = 0; i < 50; i++) {                //对请求字符串进行处理
        if(((char *)tempdata + 4)[i] == ' ' ||
            ((char *)tempdata + 4)[i] == '\r' ||
            ((char *)tempdata + 4)[i] == '\n' ||
            ((char *)tempdata + 4)[i] == '=' ||
            ((char *)tempdata + 4)[i] == '&' ||
            ((char *)tempdata + 4)[i] == '?' ) {
          ((char *)tempdata + 4)[i] = 0;
        }
      }
      tcp_recved(pcb, p->tot_len);             //通知协议栈，成功收到客户端数据长度，更新接\收窗口
      hs->web = WEB_Callback((char *)&tempdata[4]);
                                               //根据请求获取 web 数据，存放于 web 域中
      send_data(pcb, hs);                      //调用此函数，发送一次页面数据
      tcp_sent(pcb, http_sent);                //设定续传页面数据的回调函数
        }
      mem_free(tempdata);
      }
      pbuf_free(p);
   }
```

```
else if(err == ERR_OK && p == NULL) {
        close_conn(pcb, hs);                    //remote host  关闭连接
    }
    else
    {
    tcp_abort(pcb);                             //协议栈内部出错，调用 tcp_abort 中止连接
    return ERR_ABRT;
    }
    return ERR_OK;                              //一切 ok，继续本连接
}
```

接下来是 send_data 函数的清单。