# LwIP协议栈的学习与应用

## 前言

LWIP(Light Weight Internet Protoco1)是瑞士计算机科学院(Swedish Institute of Computer Science)AdamDunkels等人开发的一套用于嵌入式系统的开放源代码TCP／IP协议栈。LWIP的含义是Light Weight(轻型)IP协议。LWIP可以移植到操作系统上，也可以在无操作系统的情况下独立运行。LWIP TCP／IP实现的重点是在保持TCP协议主要功能的基础上减少对RAM的占用。一般它只需要几十KB的RAM和40 KB左右的ROM就可以运行，这使LWIP协议栈适合在小型嵌入式系统中使用。

LwIP的主要特性如下：

(1)支持多网络接口下的IP转发；

(2) 支持ICMP协议；

(3) 包括实验性扩展的的UDP（用户数据报协议）；

(4)包括阻塞控制，RTT估算和快速恢复和快速转发的TCP（传输控制协议）；

(5)提供专门的内部回调接口（Raw API）用于提高应用程序性能；

(6)可选择的Berkeley接口API（多线程情况下）；

(7)在最新的版本中支持ppp；

(8)新版本中增加了的IP fragment的支持；

(9) 支持DHCP协议,动态分配ip地址。

## 第一部分 协议栈的移植

### 准备工作

本文的硬件环境采用LPC2468作为主控MCU，LPC24xx是NXP半导体公司(由Philips公司创建) 推出的基于ARM7TDMI-S内核的微控制器，它在片上集成了10 Mbps／100 Mbps以太网控制器。PHY芯片采用DM9161AEP。 操作系统方面，我们选用的μC/OS II 2.88。开发环境采用KEIL MDK3.8。

### 第二章 操作系统适配层

为了使lwIP便于移植，与操作系统有关的功能函数调用和数据结构没有在代码中直接使用。而是当需要这样的函数时，操作系统适配层将加以使用。操作系统适配层向诸如定时器、处理同步、消息传送机制等的操作系统服务提供一套统一的接口。原则上，移植lwIP到其他操作系统时，仅仅需要实现适合该操作系统的操作系统模拟层。

操作系统适配层提供了由TCP使用的定时器功能。操作系统适配层提供的定时器是一次性的定时器，当超时发生时，调用一个已注册函数至少要200ms的间隔。

进程同步机制仅提供了信号量。即使在操作系统底层中信号量不可用，也可以通过其他信号原语像条件变量或互锁来模拟。

信息传递的实现使用一种简单机制，用一种称为“邮箱”的抽象方法。邮箱做两种操作：邮寄和提取。邮寄操作不会阻塞进程；邮寄到邮箱的消息由操作系统模拟层排入队列直到另一个进程来提取它们。即使操作系统底层对邮箱机制不支持，也容易用信号量实现。

信号量多用于任务间同步和互斥。 邮箱用于大数据的传送。 队列多用于处理有序的事件。 做比较“粗俗”的比喻，信号量就是中央政府发给官人做一方大员的官印，有很多种官印但是不能一印多发，得到官印者才能掌权鱼肉一方百姓（任务得到信号量才能运行），否则你就只要等官跑官。邮箱，就好给比当差的下达的抄家、拆房、收监等红头文件，拿到啥样的文件就干啥。消息队列，就是给任务发了一连串的邮件，官员（任务）拿到这一大摞文件，可以从底部或者顶部（LIFO or FIFO)一个一个拆开处理。（注：来自www.ouravr.com）

操作系统适配层的移植主要是在sys\_arch.c里面,主要有以下几部分:

**信号量相关:**

sys\_sem\_t sys\_sem\_new(u8\_t count)

创建一个新的信号量,并给信号量赋予初值 count。

void sys\_sem\_signal(sys\_sem\_t sem)

向指定的信号量发送信号。

void sys\_sem\_free(sys\_sem\_t sem)

释放指定的信号量

u32\_t sys\_arch\_sem\_wait(sys\_sem\_t sem, u32\_t timeout)

**邮箱(MailBox)相关:**

sys\_mbox\_t sys\_mbox\_new(int size) 函数

建立一个空的邮箱，如果创建成功，则返回邮箱的地址，如果创建失败则返回为空。

void sys\_mbox\_free(sys\_mbox\_t mbox) 函数

void sys\_mbox\_post(sys\_mbox\_t mbox, void \*msg) 函数

err\_t sys\_mbox\_trypost(sys\_mbox\_t mbox, void \*msg) 函数

u32\_t sys\_arch\_mbox\_fetch(sys\_mbox\_t mbox, void \*\*msg, u32\_t timeout)

u32\_t sys\_arch\_mbox\_tryfetch(sys\_mbox\_t mbox, void \*\*msg) 函数

这个函数是1.3后新有的,

### 网卡驱动层

网卡的驱动层主要分为2个方面：MAC和PHY的初始化，数据的收发控制。下面先介绍MAC和PHY的初始化：

以太网接口的自适应能力由DM9161AEP的自动协商功能体现出来。自动协商功能提供了一种在网络连接的两端之间交换配置信息的机制，在该机制下，这两端将自动选择最优的配置， DM9161AEP支持4种不同的以太网工作方式(10 Mbps半双工、10 Mbps全双工、100 Mbps半双工和100 Mbps全双工)，自动协商功能在芯片配置的基础上自动选择性能最高的工作方式。

为了进行数据高效率的收发，我们设计了接收和发送两个线程进行并发处理。数据接收线程采用信号量机制，一直在等待ISR的数据接收信号。

### 第三章 应用示例

**LwIP的应用程序接口**

通常情况下TCP/IP协议栈的数据处理流程一般有几种方式：

(1) TCP/IP协议的每一层是一个单独进程。链路层是一个进程，IP层是一个进程，TCP层是一个进程。这样的好处是网络协议的每一层都非常清晰，代码的调试和理解都非常容易。但是最大的坏处数据跨层传递时会引起上下文切换(context switch)。对于接收一个TCP segment要引起3次context switch(从网卡驱动程序到链路层进程，从链路层进程到ip层进程，从ip层进程到TCP进程)。通常对于操作系统来说，任务切换是要浪费时间的。过频的context swich是不可取的。

(2) TCP/IP协议栈在操作系统内核当中。应用程序通过操作系统的系统调用(system call)和协议栈来进行通讯。这样TCP/IP的协议栈就限定于特定的操作系统内核了。如windows就是这种方式。

(3) TCP/IP协议栈都在一个进程当中。这样TCP/IP协议栈就和操作系统内核分开了。而应用层程序既可以是单独的进程也可以驻留在TCP/IP进程中。如果应用程序是单独的进程可以通过操作系统的邮箱，消息队列等和TCP/IP进程进行通讯。

LwIP采用的是第三种的实现方式，更具体的，LwIP提供以下三种应用程序接口：

(1)RAW API。应用程序直接调用TCP/IP协议栈中的回调函数，应用程序和协议栈代码集成在同一个任务中，这样相对于普通的BSD API来说，速度更快，内存消耗更少。LwIP的后两种API的实现也是基于RAW API。RAW API的缺点是编程较为复杂；

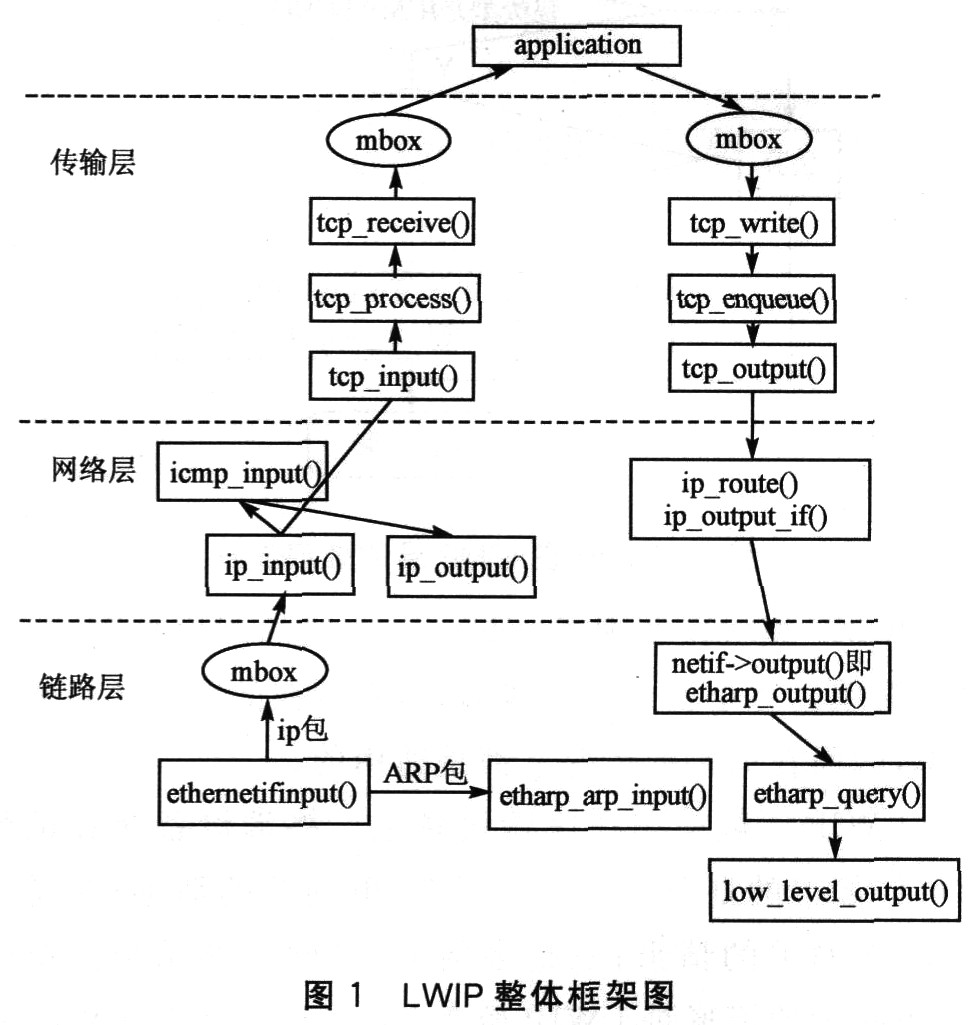
(2)正式的API。这种实现方式是在系统中单独建立了一个TCP/IP任务，由这个任务调用RAW API来处理网络通信，其它的网络任务都是利用消息机制与这个任务通信来收发数据。这也是本文中使用的方法；

(3)BSD API。这种API非常像BSD标准UNIX中的socket API，和普通的socket API一样是基于open-read-write-close模型的，这种API是对正式的API又一层的封装，效率较低，资源消耗较多，但是使用BSD API的应用程序有较好的移植性。

### 第四章 性能调优

## 第二部分 代码剖析

### 第一章 总揽



### 第二章 基础组件

**内存管理**

LwIP内存管理部分**（mem.h mem.c）**比较灵活，支持多种分配策略，有运行时库自带的内存分配(MEM\_LIBC\_MALLOC)，有内存池分配(MEM\_USE\_POOLS)，有动态内存堆分配，这些分配策略可以通过宏定义来更改。在嵌入式系统里面，C运行时库自带的内存分配一般情况下很少用，更多的是后面二者，下面就这两种分配策略进行简单的分析：

**动态内存堆分配**

其原理就是在一个事先定义好大小的内存块中进行管理，其内存分配的策略是采用最快合适（First Fit）方式，只要找到一个比所请求的内存大的空闲块，就从中切割出合适的块，并把剩余的部分返回到动态内存堆中。在分配的内存块前大约有12字节会存放内存分配器管理用的私有数据，该数据区不能被用户程序修改，否则导致致命问题。

内存释放的过程是相反的过程，但分配器会查看该节点前后相邻的内存块是否空闲，如果空闲则合并成一个大的内存空闲块。

采用这种分配策略，其优点就是内存浪费小，比较简单，适合用于小内存的管理，其缺点就是如果频繁的动态分配和释放，可能会造成严重的内存碎片，如果在碎片情况严重的话，可能会导致内存分配不成功。对于动态内存的使用，比较推荐的方法就是分配->释放->分配->释放，这种使用方法能够减少内存碎片。

**mem\_init( )** 内存堆的初始化，主要是告知内存堆的起止地址，以及初始化空闲列表，由lwip初始化时自己调用，该接口为内部私有接口，不对用户层开放。

**mem\_malloc( )** 申请分配内存。将总共需要的字节数作为参数传递给该函数，返回值是指向最新分配的内存的指针，而如果内存没有分配好，则返回值是NULL，分配的空间大小会收到内存对齐的影响，可能会比申请的略大。返回的内存是“没有“初始化的。这块内存可能包含任何随机的垃圾，你可以马上用有效数据或者至少是用零来初始化这块内存。内存的分配和释放，不能在中断函数里面进行。内存堆是全局变量，因此内存的申请、释放操作做了线程安全保护，如果有多个线程在同时进行内存申请和释放，那么可能会因为信号量的等待而导致申请耗时较长。

**mem\_calloc( )** 是对mem\_malloc( )函数的简单包装，他有两个参数，分别为元素的数目和每个元素的大小，这两个参数的乘积就是要分配的内存空间的大小，与mem\_malloc()不同的是它会把动态分配的内存清零。有经验的程序员更喜欢使用mem\_malloc ()，因为这样的话新分配内存的内容就不会有什么问题，调用mem\_malloc ()肯定会清0，并且可以避免调用memset()。

**mem\_realloc( )** 函数

Opt.h文件的宏MEM\_SIZE 是表示初始内存堆的大小。

**动态内存池分配**

说实话，我也不知道这个说法对不对，反正从源代码里面看，前者叫heap，后者叫pool。欢迎专家指正。

动态内存分配方式只能在内存池与内存堆中二选一。他们对外部的接口都是一样，只不过内部工作原理不太一样。动态内存池分配部分底层实现是在memp.c，memp.h 文件里面实现。

采用内存池进行内存管理可以有效防止内存碎片的产生，而且相比之下内存的分配、释放效率更高，不过，他会浪费部分内存。

需要启用宏MEM\_USE\_POOLS 和 MEM\_USE\_CUSTOM\_POOLS，另外还要做类似如下定义：

LWIP\_MALLOC\_MEMPOOL\_START

LWIP\_MALLOC\_MEMPOOL(20, 256)

LWIP\_MALLOC\_MEMPOOL(10, 512)

LWIP\_MALLOC\_MEMPOOL(5, 1512)

LWIP\_MALLOC\_MEMPOOL\_END

上面的意思就是分配20个256字节长度的内存块，10个512字节的内存块，5个1512字节的内存块。内存池管理会根据以上的宏自动在内存中静态定义一个大片内存用于内存池。在内存分配申请的时候，自动根据所请求的大小，选择最适合他长度的池里面去申请，如果启用宏 MEM\_USE\_POOLS\_TRY\_BIGGER\_POOL，那么，如果上述的最适合长度的池中没有空间可以用了，分配器将从更大长度的池中去申请，不过这样会浪费更多的内存。

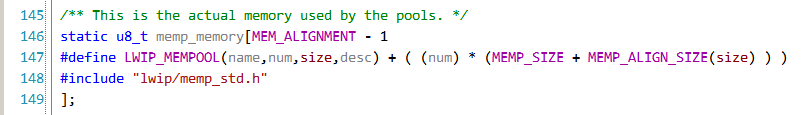
此外，LwIP为内部的一些结构设计了专用的内存池，比如netconn，协议控制块，数据包等，这些都是在memp.c/memp.h里用内存池进行管理的。

这个模块里面LwIP把C语言的宏用到了极致，它大量采用了C语言的宏特性，设计上面也非常的精妙，看上去也很优雅，不过对于初学者来说猛的看上去很头大，下面就且听我给你介绍，我们先看几个静态变量数组：

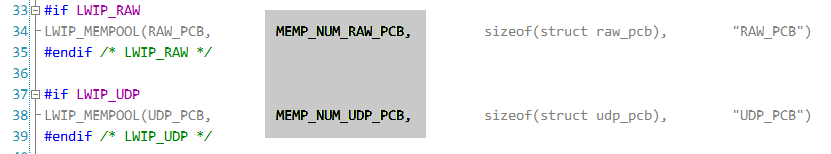
memp\_memory[]

这是内存池容器，他的大小由编译期决定，他是各个组件的结构用量的累加。

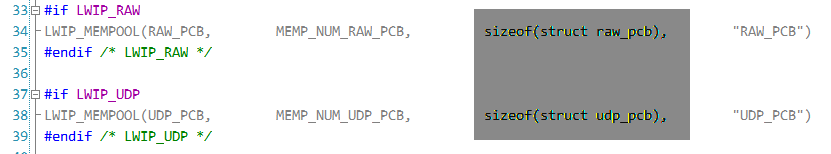
我们来看代码：



这是内存池的具体定义，通过147行，我们可以看出内存池的大小由各个组件的 num\*size的累加。如下图所示，每个组件的num 就是下列阴影区的宏定义(在memp\_std.h文件)



size的大小就是各个结构的大小，如下图阴影区所示。



整个内存池的大小又可以根据组件的需要而调整，比如，如果你不需要UDP，那么只要把LWIP\_UDP定义为0。用宏定义来实现用起来方便，改起来容易，就是看起来头大。

memp\_num

这个静态数组用于保存各个组件的成员数目，与memp\_memory类似也是用宏实现的。

memp\_sizes

这个静态数组用于保存各个组件的结构大小，与memp\_memory类似也是用宏实现的。

memp\_init

内存池的初始化，主要是为每种内存池建立链表memp\_tab，其链表是逆序的，此外，如果有统计功能使能的话，也把记录了各种内存池的数目。

memp\_malloc

如果相应的memp\_tab链表还有空闲的节点，则从中切出一个节点返回，否则返回空。

memp\_free

把释放的节点添加到相应的链表memp\_tab头上。

**pbuf**

pbuf是lwIP包的内部表示，被设计为最小化栈的特殊需要。pbufs类似于BSD实现中的mbufs。pbuf结构支持为包内容动态分配内存和让包数据驻留在静态内存中。pbufs能被一个称为pbuf链的链接到一个链表中，以至一个包能跨越多个pbufs。

pbufs有三种类型:PBUF\_RAM,PBUF\_ROM和PBUF\_POOL。图1表示PBUF\_RAM类型，包含有存在内存中由pbuf子系统管理的包数据。图2显示了一个pbuf链表，第1个是PBUF\_RAM类型，第2个是PBUF\_ROM类型，意味着它包含有不被pubf子系统管理的内存数据。图3描述了PBUF\_POOL，其包含有从固定大小pbuf池中分配来的pbuf。一个pbuf链可以包含多个不同类型的pbuf。

这三种类型有不同的用处。PBUF\_POOL类型主要由网络设备驱动使用，因为分配单个pbuf快速且适合中断句柄使用。PBUF\_ROM类型由应用程序发送那些在应用程序内存空间中的数据时使用。这些数据不会在pbuf递交给TCP/IP栈后被修改，因此这个类型主要用于当数据在ROM中时。PBUF\_ROM中指向数据的头部被存在链表中其前一个PUBF\_RAM类型的pbuf中，如图2所示。

PBUF\_RAM类型也用于应用程序发送动态产生的数据。这情况下，pbuf系统不仅为应用程序数据分配内存，也为将指向（prepend）数据的头部分配内存。如图1所示。pbuf系统不能预知哪种头部将指向（prepend）那些数据，只假定最坏的情况。头部的大小在编译时确定。

本质上，进来的pbuf是PBUF\_POOL类型，而出去的pbuf是PBUF\_ROM或PBUF\_RAM类型。

从图1，图2可以看出pbuf的内部结构。pbuf结构包含有两个指针，两个长度字段，一个标志字段，和一个参考计数。next字段指向统一链表中的下一个pbuf。有效载荷指针指向该pbuf中数据的起始点。Len字段包含有该pbuf数据内同的长度。tot\_len字段是当前pbuf和所有链表接下来中的len字段值的总和。**简单说，tot\_len字段是len字段及下一个pbuf中tot\_len字段值的总和。f**lags字段表示pbuf类型而ref字段包含一个参考计数。next和payload字段是本地指针，其大小由处理器体系结构决定。两个长度字段是16位无符号整数，而flags和ref字段都是4比特大小。pbuf的总大小决定于使用的处理器体系结构。在32位指针和4字节校正的体系结构上，总大小是16字节，而在16位指针和1自己校正的体系结构上，总大小是9字节。

pbuf模块提供了操作pbuf的函数：

pbuf\_alloc()可以分配前面提到的三种类型的pbuf。

pbuf\_ref()增加引用计数,

pbuf\_free()释放分配的空间，它先减少引用计数，当引用计数为0时就释放pbuf。

pbuf\_realloc()收缩空间以使pbuf只占用刚好的空间保存数据。

pbuf\_header()调整payload指针和长度字段，以使一个头部指向pbuf中的数据。pbuf\_chain()和pbuf\_dechain()用于链表化pbuf。

### 第二章 ARP

**ARP协议**（**A**ddress **R**esolution **P**rotocol），或称**地址解析协议**。ARP协议的基本功能就是通过目标设备的IP地址，查询目标设备的MAC地址，以保证通信的顺利进行。他是IPv4中网络层必不可少的协议，不过在IPv6中已不再适用，并被icmp v6所替代。

**功能**

在局域网中，网络中实际传输的是“帧”（frame），帧里面是有目标主机的MAC地址的。在以太网中，一个主机要和另一个主机进行直接通信，必须要知道目标主机的MAC地址，但这个目标MAC地址是通过地址解析协议获得的。所谓“地址解析”就是主机在发送帧前将目标IP地址转换成目标MAC地址的过程。

**原理**

在每台安装有TCP/IP协议的电脑或 route 里都有一个ARP缓存表，表里的IP地址与MAC地址是一对应的，如表甲所示。

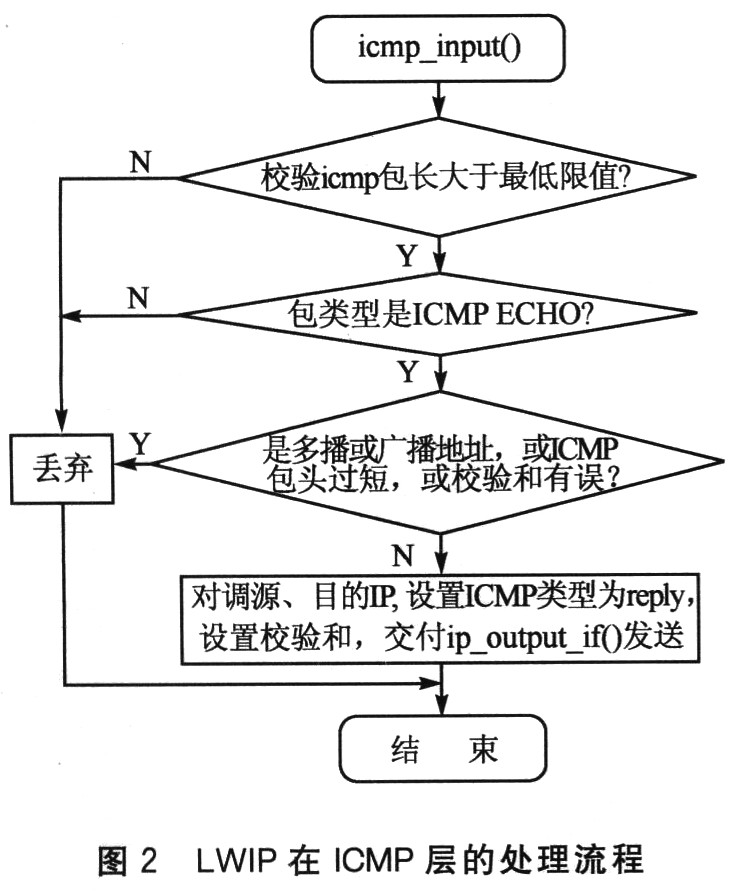
|  |  |  |
| --- | --- | --- |
| **主机名称** | **IP地址** | **MAC地址** |
| A | 192.168.38.10 | 00-AA-00-62-D2-02 |
| B | 192.168.38.11 | 00-BB-00-62-C2-02 |
| C | 192.168.38.12 | 00-CC-00-62-C2-02 |
| D | 192.168.38.13 | 00-DD-00-62-C2-02 |
| E | 192.168.38.14 | 00-EE-00-62-C2-02 |
| ... | ... |  |

以主机A（192.168.38.10）向主机B（192.168.38.11）发送数据为例。当发送数据时，主机A会在自己的ARP缓存表中寻找是否有目标IP地址。如果找到了，也就知道了目标MAC地址为(00-BB-00-62-C2-02)，直接把目标MAC地址写入帧里面发送就可以了；如果在ARP缓存表中没有找到相对应的IP地址，主机A就会在网络上发送一个广播(ARP request)，目标MAC地址是“FF.FF.FF.FF.FF.FF”，这表示向同一网段内的所有主机发出这样的询问：“192.168.38.11的MAC地址是什么？”网络上其他主机并不响应ARP询问，只有主机B接收到这个帧时，才向主机A做出这样的回应(ARP response)：“192.168.38.11的MAC地址是(00-BB-00-62-C2-02)”。这样，主机A就知道了主机B的MAC地址，它就可以向主机B发送信息了。同时它还更新了自己的ARP缓存表，下次再向主机B发送信息时，直接从ARP缓存表里查找就可以了。ARP缓存表采用了老化机制，在一段时间内如果表中的某一行没有使用，就会被删除，这样可以大大减少ARP缓存表的长度，加快查询速度。

ARP协议是一个网络层的协议，实现的功能是网络设备的MAC地址到IP地址的映射。在以太网中每个网络设备都有一个唯一的48位（6字节）MAC地址，数据报都是按照MAC地址发送的，其地址范围是由相关组织按照不同设备制造商统一分配的，这样保证了网络上设备地址不会冲突。但是TCP/IP协议是以32位（4字节）IP地址作为通讯地址的，怎样使MAC地址和IP地址对应上呢，这里就用到了ARP协议。

ARP的工作过程大致是这样的：比如网络中的一台主机想要知道MAC地址为01：02：03：04：05：06的机器的IP地址，于是它就向网上发送一个ARP查询数据报（目标MAC全为FF的广播报文），网络上的所有机器收到这个广播后将查询的MAC与自己的MAC比对，如果不一致，则不回应该报文。若一致则向该主机发出ARP回复数据报（这时就是只针对发送方的单播报文了），告诉主机自己的IP（比如192.9.200.128）。这样主机就会在ARP映射表中记录这一项192.9.200.128-----〉01：02：03：04：05：06。以后，发往这个IP地址的IP/TCP/UDP等数据报就会对应到它的MAC地址。在Windows命令提示符窗口输入arp -a查询ARP表项可以看到 MAC-〉IP的映射。

### 第三章 ICMP



### 第四章 IP分片

以太网对数据帧长度有一个最大限制，通常是1500字节，这个特性称作MTU，最大传输单元。如果IP层有一个数据包的长度比MTU还大，那么IP层就需要进行分包，有时也称之为分片，把数据包分成若干片，这样每一片都小于MTU。

分片和重新组装的过程对传输层是透明的，其原因是当IP数据报进行分片之后，只有当它到达目的站时，才可进行重新组装，且它是由目的端的IP层来完成的。分片之后的数据报根据需要也可以再次进行分片。

IP分片和完整IP报文差不多拥有相同的IP头，ID域对于每个分片都是一致的，这样才能在重新组装的时候识别出来自同一个IP报文的分片。在IP头里面，16位识别号唯一记录了一个IP包的ID，具有同一个ID的IP分片将会重新组装；而13位片偏移则记录了某IP片相对整个包的位置；而这两个表中间的3位标志则标志着该分片后面是否还有新的分片。这三个标志就组成了IP分片的所有信息(将在后面介绍)，接受方就可以利用这些信息对IP数据进行重新组织。

### 第五章 协议栈分析

#### SOCKET的实现

Lwip协议栈的实现目的，无非是要上层用来实现app的socket编程。好，我们就从socket开始。为了兼容性，lwip的socket应该也是提供标准的socket接口函数，恩，没错，在src\include\lwip\socket.h文件中可以看到下面的宏定义：

#if LWIP\_COMPAT\_SOCKETS

#define accept(a,b,c) lwip\_accept(a,b,c)

#define bind(a,b,c) lwip\_bind(a,b,c)

#define shutdown(a,b) lwip\_shutdown(a,b)

#define closesocket(s) lwip\_close(s)

#define connect(a,b,c) lwip\_connect(a,b,c)

#define getsockname(a,b,c) lwip\_getsockname(a,b,c)

#define getpeername(a,b,c) lwip\_getpeername(a,b,c)

#define setsockopt(a,b,c,d,e) lwip\_setsockopt(a,b,c,d,e)

#define getsockopt(a,b,c,d,e) lwip\_getsockopt(a,b,c,d,e)

#define listen(a,b) lwip\_listen(a,b)

#define recv(a,b,c,d) lwip\_recv(a,b,c,d)

#define recvfrom(a,b,c,d,e,f) lwip\_recvfrom(a,b,c,d,e,f)

#define send(a,b,c,d) lwip\_send(a,b,c,d)

#define sendto(a,b,c,d,e,f) lwip\_sendto(a,b,c,d,e,f)

#define socket(a,b,c) lwip\_socket(a,b,c)

#define select(a,b,c,d,e) lwip\_select(a,b,c,d,e)

#define ioctlsocket(a,b,c) lwip\_ioctl(a,b,c)

#if LWIP\_POSIX\_SOCKETS\_IO\_NAMES

#define read(a,b,c) lwip\_read(a,b,c)

#define write(a,b,c) lwip\_write(a,b,c)

#define close(s) lwip\_close(s)

先不说实际的实现函数，光看这些定义的宏，就是标准socket所必须有的接口。

接着看这些实际的函数实现。这些函数实现在src\api\socket.c中。先看下接受连接的函数，这个是tcp的

原型：int lwip\_accept(int s, struct sockaddr \*addr, socklen\_t \*addrlen)

可以看到这里的socket类型参数 s，实际上是个int型

在这个函数中的第一个函数调用是sock = get\_socket(s);

这里的sock变量类型是lwip\_socket，定义如下：

/\*\* Contains all internal pointers and states used for a socket \*/

struct lwip\_socket {

/\*\* sockets currently are built on netconns, each socket has one netconn \*/

struct netconn \*conn;

/\*\* data that was left from the previous read \*/

struct netbuf \*lastdata;

/\*\* offset in the data that was left from the previous read \*/

u16\_t lastoffset;

/\*\* number of times data was received, set by event\_callback(),

tested by the receive and select functions \*/

u16\_t rcvevent;

/\*\* number of times data was received, set by event\_callback(),

tested by select \*/

u16\_t sendevent;

/\*\* socket flags (currently, only used for O\_NONBLOCK) \*/

u16\_t flags;

/\*\* last error that occurred on this socket \*/

int err;

};

好，这个结构先不管它，接着看下get\_socket函数的实现【也是在src\api\socket.c文件中】，在这里我们看到这样一条语句sock = &sockets[s];很明显，返回值也是这个sock，它是根据传进来的序列号在sockets数组中找到对应的元素并返回该元素的地址。好了，那么这个sockets数组是在哪里被赋值了这些元素的呢？

进行到这里似乎应该从标准的socket编程的开始，也就是socket函数讲起，那我们就顺便看一下。它对应的实际实现是下面这个函数

Int lwip\_socket(int domain, int type, int protocol)【src\api\socket.c】

这个函数根据不同的协议类型，也就是函数中的type参数，创建了一个netconn结构体的指针，接着就是用这个指针作为参数调用了alloc\_socket函数，下面具体看下这个函数的实现

static int alloc\_socket(struct netconn \*newconn)

{

int i;

/\* Protect socket array \*/

sys\_sem\_wait(socksem);

/\* allocate a new socket identifier \*/

for (i = 0; i < NUM\_SOCKETS; ++i) {

if (!sockets[i].conn) {

sockets[i].conn = newconn;

sockets[i].lastdata = NULL;

sockets[i].lastoffset = 0;

sockets[i].rcvevent = 0;

sockets[i].sendevent = 1; /\* TCP send buf is empty \*/

sockets[i].flags = 0;

sockets[i].err = 0;

sys\_sem\_signal(socksem);

return i;

}

}

sys\_sem\_signal(socksem);

return -1;

}

对了，就是这个时候对全局变量sockets数组的元素赋值的。

既然都来到这里了，那就顺便看下netconn结构的情况吧。它的学名叫netconn descriptor

/\*\* A netconn descriptor \*/

struct netconn

{

/\*\* type of the netconn (TCP, UDP or RAW) \*/

enum netconn\_type type;

/\*\* current state of the netconn \*/

enum netconn\_state state;

/\*\* the lwIP internal protocol control block \*/

union {

struct ip\_pcb \*ip;

struct tcp\_pcb \*tcp;

struct udp\_pcb \*udp;

struct raw\_pcb \*raw;

} pcb;

/\*\* the last error this netconn had \*/

err\_t err;

/\*\* sem that is used to synchroneously execute functions in the core context \*/

sys\_sem\_t op\_completed;

/\*\* mbox where received packets are stored until they are fetched

by the netconn application thread (can grow quite big) \*/

sys\_mbox\_t recvmbox;

/\*\* mbox where new connections are stored until processed

by the application thread \*/

sys\_mbox\_t acceptmbox;

/\*\* only used for socket layer \*/

int socket;

#if LWIP\_SO\_RCVTIMEO

/\*\* timeout to wait for new data to be received

(or connections to arrive for listening netconns) \*/

int recv\_timeout;

#endif /\* LWIP\_SO\_RCVTIMEO \*/

#if LWIP\_SO\_RCVBUF

/\*\* maximum amount of bytes queued in recvmbox \*/

int recv\_bufsize;

#endif /\* LWIP\_SO\_RCVBUF \*/

u16\_t recv\_avail;

/\*\* TCP: when data passed to netconn\_write doesn't fit into the send buffer,

this temporarily stores the message. \*/

struct api\_msg\_msg \*write\_msg;

/\*\* TCP: when data passed to netconn\_write doesn't fit into the send buffer,

this temporarily stores how much is already sent. \*/

int write\_offset;

#if LWIP\_TCPIP\_CORE\_LOCKING

/\*\* TCP: when data passed to netconn\_write doesn't fit into the send buffer,

this temporarily stores whether to wake up the original application task

if data couldn't be sent in the first try. \*/

u8\_t write\_delayed;

#endif /\* LWIP\_TCPIP\_CORE\_LOCKING \*/

/\*\* A callback function that is informed about events for this netconn \*/

netconn\_callback callback;

};【src\include\lwip\api.h】

到此，对这个结构都有些什么，做了一个大概的了解。

下面以SOCK\_STREAM类型为例，看下netconn的new过程：

在lwip\_socket函数中有

case SOCK\_DGRAM:

conn = netconn\_new\_with\_callback( (protocol == IPPROTO\_UDPLITE) ?

NETCONN\_UDPLITE : NETCONN\_UDP, event\_callback);

#define netconn\_new\_with\_callback(t, c) netconn\_new\_with\_proto\_and\_callback(t, 0, c)

简略实现如下：

struct netconn\*

netconn\_new\_with\_proto\_and\_callback(enum netconn\_type t, u8\_t proto, netconn\_callback callback)

{

struct netconn \*conn;

struct api\_msg msg;

conn = netconn\_alloc(t, callback);

if (conn != NULL )

{

msg.function = do\_newconn;

msg.msg.msg.n.proto = proto;

msg.msg.conn = conn;

TCPIP\_APIMSG(&msg);

}

return conn;

}

主要就看TCPIP\_APIMSG了，这个宏有两个定义，一个是LWIP\_TCPIP\_CORE\_LOCKING的，一个非locking的。分别分析这两个不同类型的函数

\* Call the lower part of a netconn\_\* function

\* This function has exclusive access to lwIP core code by locking it

\* before the function is called.

err\_t tcpip\_apimsg\_lock(struct api\_msg \*apimsg)【这个是可以locking的】

{

LOCK\_TCPIP\_CORE();

apimsg->function(&(apimsg->msg));

UNLOCK\_TCPIP\_CORE();

return ERR\_OK;

}

\* Call the lower part of a netconn\_\* function

\* This function is then running in the thread context

\* of tcpip\_thread and has exclusive access to lwIP core code.

err\_t tcpip\_apimsg(struct api\_msg \*apimsg)【此为非locking的】

{

struct tcpip\_msg msg;

if (mbox != SYS\_MBOX\_NULL) {

msg.type = TCPIP\_MSG\_API;

msg.msg.apimsg = apimsg;

sys\_mbox\_post(mbox, &msg);

sys\_arch\_sem\_wait(apimsg->msg.conn->op\_completed, 0);

return ERR\_OK;

}

return ERR\_VAL;

}

其实，功能都是一样的，都是要对apimsg->function函数的调用。只是途径不一样而已。看看它们的功能说明就知道了。这么来说apimsg->function的调用很重要了。从netconn\_new\_with\_proto\_and\_callback函数的实现，可以知道这个function就是do\_newconn

Void do\_newconn(struct api\_msg\_msg \*msg)

{

if(msg->conn->pcb.tcp == NULL) {

pcb\_new(msg);

}

/\* Else? This "new" connection already has a PCB allocated. \*/

/\* Is this an error condition? Should it be deleted? \*/

/\* We currently just are happy and return. \*/

TCPIP\_APIMSG\_ACK(msg);

}

还是看TCP的，在pcb\_new函数中有如下代码：

case NETCONN\_TCP:

msg->conn->pcb.tcp = tcp\_new();

if(msg->conn->pcb.tcp == NULL) {

msg->conn->err = ERR\_MEM;

break;

}

setup\_tcp(msg->conn);

break;

我们知道在这里建立了这个tcp的连接。至于这个超级牛的函数，以后再做介绍。

嗯，还是回过头来接着看accept函数吧。

Sock获得了，接着就是newconn = netconn\_accept(sock->conn);通过mbox取得新的连接。粗略的估计了一下，这个新的连接应该和listen有关系。那就再次打断一下，看看那个listen操作。

lwip\_listen --🡪 netconn\_listen\_with\_backlog--🡪 do\_listen--🡪

tcp\_arg(msg->conn->pcb.tcp, msg->conn);

tcp\_accept(msg->conn->pcb.tcp, accept\_function);//注册了一个接受函数

\* Accept callback function for TCP netconns.

\* Allocates a new netconn and posts that to conn->acceptmbox.

static err\_t accept\_function(void \*arg, struct tcp\_pcb \*newpcb, err\_t err)

{

struct netconn \*newconn;

struct netconn \*conn;

conn = (struct netconn \*)arg;

/\* We have to set the callback here even though

\* the new socket is unknown. conn->socket is marked as -1. \*/

newconn = netconn\_alloc(conn->type, conn->callback);

if (newconn == NULL) {

return ERR\_MEM;

}

newconn->pcb.tcp = newpcb;

setup\_tcp(newconn);

newconn->err = err;

/\* Register event with callback \*/

API\_EVENT(conn, NETCONN\_EVT\_RCVPLUS, 0);

if (sys\_mbox\_trypost(conn->acceptmbox, newconn) != ERR\_OK)

{

/\* When returning != ERR\_OK, the connection is aborted in tcp\_process(),

so do nothing here! \*/

newconn->pcb.tcp = NULL;

netconn\_free(newconn);

return ERR\_MEM;

}

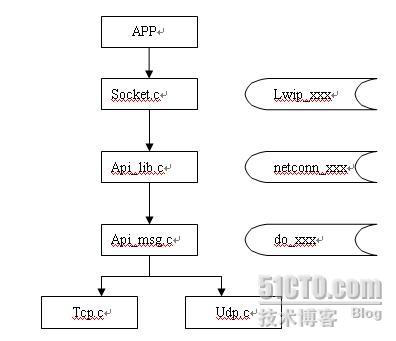
return ERR\_OK;

}

对了，accept函数中从mbox中获取的连接就是这里放进去的。

再回到accept中来，取得了新的连接，接下来就是分配sock了，再然后，再然后？再然后就等用户来使用接收、发送数据了。

到此整个APP层，也就是传输层以上对socket的封装讲完了。在最后再总结一些整个路径的调用情况吧



#### API\_MSG结构及其实现

如果要评起到最关键作用的一个结构体，那么struct api\_msg当之无愧。先看下它的定义：

/\*\* This struct contains a function to execute in another thread context and

a struct api\_msg\_msg that serves as an argument for this function.

This is passed to tcpip\_apimsg to execute functions in tcpip\_thread context. \*/

struct api\_msg

{

/\*\* function to execute in tcpip\_thread context \*/

void (\* function)(struct api\_msg\_msg \*msg);

/\*\* arguments for this function \*/

struct api\_msg\_msg msg;

};

功能说的很清楚。但是具体怎么个操作法还是不知道，没关系，接着看它的调用。

举一个例子，刚好是上一篇中调用，但是没有看具体实现的

err\_t netconn\_getaddr(struct netconn \*conn, struct ip\_addr \*addr, u16\_t \*port, u8\_t local)

{

struct api\_msg msg;

msg.function = do\_getaddr;

msg.msg.conn = conn;

msg.msg.msg.ad.ipaddr = addr;

msg.msg.msg.ad.port = port;

msg.msg.msg.ad.local = local;

TCPIP\_APIMSG(&msg);

return conn->err;

}

说明一下，api\_msg结构几乎都是在netconn\_xxx函数中被调用，方式千篇一律，除了msg.funcion的赋值不一样外。上面的调用很简单，对该结构体变量赋值，接着就是调用TCPIP\_APIMSG，这个函数上面讲过，可过去看下。既然如此，就不得不说mbox及其相关函数了。

static sys\_mbox\_t mbox = SYS\_MBOX\_NULL;【tcp.c】

再看sys\_mbox\_t的定义，在【src\include\lwip\sys.h】中

/\* For a totally minimal and standalone system, we provide null

definitions of the sys\_ functions. \*/

typedef u8\_t sys\_sem\_t;

typedef u8\_t sys\_mbox\_t;

typedef u8\_t sys\_prot\_t;

可以看到这里只是简单的定义成了u8类型，注意上面的红色字体的说明，很明显这个是可移植的一部分，需要根据不同的平台，不同的操作系统具体定义。可以借鉴**焦海波**大侠的关于ucos上对lwip的移植笔记来看。

我们可以看到在api\_msg结构的处理过程中，所有的信息都是包含在api\_msg\_msg结构体中的，api\_msg只是将其和function简单的组合了。下面看下这个牛结构的定义：

/\*\* This struct includes everything that is necessary to execute a function

for a netconn in another thread context (mainly used to process netconns

in the tcpip\_thread context to be thread safe). \*/

struct api\_msg\_msg

{

/\*\* The netconn which to process - always needed: it includes the semaphore

which is used to block the application thread until the function finished. \*/

struct netconn \*conn;

/\*\* Depending on the executed function, one of these union members is used \*/

union

{

/\*\* used for do\_send \*/

struct netbuf \*b;

/\*\* used for do\_newconn \*/

struct {

u8\_t proto;

} n;

/\*\* used for do\_bind and do\_connect \*/

struct {

struct ip\_addr \*ipaddr;

u16\_t port;

} bc;

/\*\* used for do\_getaddr \*/

struct {

struct ip\_addr \*ipaddr;

u16\_t \*port;

u8\_t local;

} ad;

/\*\* used for do\_write \*/

struct {

const void \*dataptr;

int len;

u8\_t apiflags;

} w;

/\*\* used ofr do\_recv \*/

struct {

u16\_t len;

} r;

#if LWIP\_IGMP

/\*\* used for do\_join\_leave\_group \*/

struct {

struct ip\_addr \*multiaddr;

struct ip\_addr \*interface;

enum netconn\_igmp join\_or\_leave;

} jl;

#endif /\* LWIP\_IGMP \*/

#if TCP\_LISTEN\_BACKLOG

struct {

u8\_t backlog;

} lb;

#endif /\* TCP\_LISTEN\_BACKLOG \*/

} msg;

};

一个很合理的设计，至少笔者是这么认为的。关键在于msg union的设计。

#### TCP层发送相关

现在我们正式开始进入对TCP的研究，它属于传输层协议，它为应用程序提供了可靠的字节流服务。在LWIP中基本的TCP处理过程被分割为六个功能函数的实现：tcp\_input(), tcp\_process(), tcp\_receive()【与TCP输入有关】, tcp\_write(), tcp\_enqueue(), tcp\_output()【用于TCP输出】。这些是从大的方面来划分的。

现在先从小部tcp.c文件来分析一下：我们知道这里的函数都是被socket那一层的最终调用的。为了利于分析，我选择lwip\_send函数来分析，具体不多说，最终调用到了

static err\_t do\_writemore(struct netconn \*conn)这个函数，当然这期间也做了不少工作，最主要的就是把发送数据的指针放到了msg的指定变量中

msg.msg.msg.w.dataptr = dataptr;//指针

msg.msg.msg.w.len = size; //长度

这些又经过转化放到了netconn的write\_msg中

最后就是对do\_writemore的调用了，下面详细分析这个函数。

这个函数的最直接调用有以下几个：

available = tcp\_sndbuf(conn->pcb.tcp);

err = tcp\_write(conn->pcb.tcp, dataptr, len, conn->write\_msg->msg.w.apiflags);

err = tcp\_output\_nagle(conn->pcb.tcp);

err = tcp\_output(conn->pcb.tcp);

好，先看tcp\_sndbuf这个。从其参数，我们应该想像pcb的重要性。

#define tcp\_sndbuf(pcb) ((pcb)->snd\_buf)//由下面分析得这里直接返回buf大小

那就继续跟踪这个pcb好了。是的，这还得从lwip\_socket开始，在创建连接netconn的时候调用了do\_newconn，它接着调用了pcb\_new，在这个函数中如果是tcp的话有以下代码

msg->conn->pcb.tcp = tcp\_new();

哈哈，还记得吧，在前面我讨论到了这里，就没有再讨论了。嗯，现在开始吧。

\* Creates a new TCP protocol control block but doesn't place it on

\* any of the TCP PCB lists.

\* The pcb is not put on any list until binding using tcp\_bind().

struct tcp\_pcb \* tcp\_new(void)

{

return tcp\_alloc(TCP\_PRIO\_NORMAL);

}

也许注释的意义更大一些吧，哈哈，至此，我们从注释可以知道，tcp\_bind的作用是把tcp的pcb放入list中，至少是某一个list

×××××××××××××××××××××××××××××××××××××××既然都提到了，似乎不说说有点过意不去啊。Lwid\_bind函数最终会调用到tcp\_bind【当然是tcp为例进行分析】。这个函数也比较有意思，在进入正题之前先来了下面这么个调用

if (port == 0)

{

port = tcp\_new\_port();

}

意思很明显，就是分配个新的端口号，有意思的就是这个端口号的分配函数tcp\_new\_port

static u16\_t

tcp\_new\_port(void)

{

struct tcp\_pcb \*pcb;

#ifndef TCP\_LOCAL\_PORT\_RANGE\_START

#define TCP\_LOCAL\_PORT\_RANGE\_START 4096

#define TCP\_LOCAL\_PORT\_RANGE\_END 0x7fff

#endif

static u16\_t port = TCP\_LOCAL\_PORT\_RANGE\_START;

again:

if (++port > TCP\_LOCAL\_PORT\_RANGE\_END) {

port = TCP\_LOCAL\_PORT\_RANGE\_START;

}

for(pcb = tcp\_active\_pcbs; pcb != NULL; pcb = pcb->next) {

if (pcb->local\_port == port) {

goto again;

}

}

for(pcb = tcp\_tw\_pcbs; pcb != NULL; pcb = pcb->next) {

if (pcb->local\_port == port) {

goto again;

}

}

for(pcb = (struct tcp\_pcb \*)tcp\_listen\_pcbs.pcbs; pcb != NULL; pcb = pcb->next) {

if (pcb->local\_port == port) {

goto again;

}

}

return port;

}分别检查了3个pcb链表，本来我不想把这个函数的实现列在这里的，但是这里告诉了我们一些东西，至少我们知道有3个pcb的链表，分别是tcp\_active\_pcbs【处于接受发送数据状态的pcbs】、tcp\_tw\_pcbs【处于时间等待状态的pcbs】、tcp\_listen\_pcbs【处于监听状态的pcbs】。

似乎tcp\_bind函数更有意思，它分别检查了4个pcbs链表，除了上面的三个还有一个tcp\_bound\_pcbs【处于已经绑定但还没有连接或者监听pcbs】。作用是是否有与目前pcb的ipaddr相同的pcb存在。不过这些都不是最重要的，TCP\_REG(&tcp\_bound\_pcbs, pcb);才是我们的目的，果然如此，直接加入到了tcp\_bound\_pcbs链表中了。

×××××××××××××××××××××××××××××××××××××××

struct tcp\_pcb \* tcp\_alloc(u8\_t prio)//是的，你没看错，这个参数是学名是叫优先级

{

pcb = memp\_malloc(MEMP\_TCP\_PCB);

if (pcb != NULL)

{

memset(pcb, 0, sizeof(struct tcp\_pcb));

pcb->prio = TCP\_PRIO\_NORMAL;

pcb->snd\_buf = TCP\_SND\_BUF;//对的，别的可以先不管，这就是我们要找的东西

pcb->snd\_queuelen = 0;

pcb->rcv\_wnd = TCP\_WND;

pcb->rcv\_ann\_wnd = TCP\_WND;

pcb->tos = 0;

pcb->ttl = TCP\_TTL;

/\* The send MSS is updated when an MSS option is received. \*/

pcb->mss = (TCP\_MSS > 536) ? 536 : TCP\_MSS;

pcb->rto = 3000 / TCP\_SLOW\_INTERVAL;

pcb->sa = 0;

pcb->sv = 3000 / TCP\_SLOW\_INTERVAL;

pcb->rtime = -1;

pcb->cwnd = 1;

iss = tcp\_next\_iss();

pcb->snd\_wl2 = iss;

pcb->snd\_nxt = iss;

pcb->snd\_max = iss;

pcb->lastack = iss;

pcb->snd\_lbb = iss;

pcb->tmr = tcp\_ticks;

pcb->polltmr = 0;

#if LWIP\_CALLBACK\_API

pcb->recv = tcp\_recv\_null;

#endif /\* LWIP\_CALLBACK\_API \*/

/\* Init KEEPALIVE timer \*/

pcb->keep\_idle = TCP\_KEEPIDLE\_DEFAULT;

#if LWIP\_TCP\_KEEPALIVE

pcb->keep\_intvl = TCP\_KEEPINTVL\_DEFAULT;

pcb->keep\_cnt = TCP\_KEEPCNT\_DEFAULT;

#endif /\* LWIP\_TCP\_KEEPALIVE \*/

pcb->keep\_cnt\_sent = 0;

}

return pcb;

}就是一个tcp\_pcb的结构体初始化过程，使用默认值填充该结构

好了，下面接着走，该轮到tcp\_write了吧

\* Write data for sending (but does not send it immediately).

\* It waits in the expectation of more data being sent soon (as

\* it can send them more efficiently by combining them together).

\* To prompt the system to send data now, call tcp\_output() after

\* calling tcp\_write().

【src\core\tcp\_out.c】

err\_t tcp\_write(struct tcp\_pcb \*pcb, const void \*data, u16\_t len, u8\_t apiflags)

{

/\* connection is in valid state for data transmission? \*/

if (pcb->state == ESTABLISHED ||

pcb->state == CLOSE\_WAIT ||

pcb->state == SYN\_SENT ||

pcb->state == SYN\_RCVD)

{

if (len > 0)

{

return tcp\_enqueue(pcb, (void \*)data, len, 0, apiflags, NULL, 0);

}

return ERR\_OK;

}

else

{

return ERR\_CONN;

}

}

这个函数确实够简单的了，检查pcb状态，直接入队列。嗯，还是觉得看注释比看函数实现过瘾。

下面的这个tcp\_enqueue才是个大头函数，lwip协议栈的设计与实现文档中是这么介绍的：应用层调用tcp\_write（）函数以实现发送数据，接着tcp\_write（）函数再将控制权交给tcp\_enqueue（），这个函数会在必要时将数据分割为适当大小的TCP段，然后再把这些TCP段放到所属连接的传输队列中【pcb->unsent】。函数的注释是这样写的：

\* Enqueue either data or TCP options (but not both) for tranmission

\* Called by tcp\_connect(), tcp\_listen\_input(), tcp\_send\_ctrl() and tcp\_write().

咱们接着上面往下看，tcp\_output\_nagle这个函数实际上是个宏，通过预判断以决定是否调用tcp\_output。也就是说最后的执行是tcp\_output来实现的。这个函数或许比tcp\_enqueue还要恐怖，从体积上来看。这里也只做一个简单介绍，具体请参阅tcp\_out.c文档。该函数的注释是这么说的：Find out what we can send and send it。文档是这么解释的：tcp\_output函数会检查现在是不是能够发送数据，也就是判断接收器窗口是否拥有足够大的空间，阻塞窗口是否也足够大，如果条件满足，它先填充未被tcp\_enqueue函数填充的tcp报头字段，接着就使用ip\_route或者ip\_output\_if函数发送数据。

#### TCP层接收相关

既然定了这么个标题，当然是要从socket的recv来讲了。这里主要涉及到lwip\_recvfrom这个函数。它的大致过程是，先通过netconn\_recv(sock->conn);从netconn的recvmbox中收取数据，在这里有个do\_recv的调用，而do\_recv又调用了tcp\_recved，关于这个函数的注释如下：

\* This function should be called by the application when it has

\* processed the data. The purpose is to advertise a larger window

\* when the data has been processed.

知道了，这里的do\_recv只是起到一个知会的作用，可能的话会对接收条件做一些调整。

回到主题，lwip\_recvfrom从netconn接收完数据就是要copy the contents of the received buffer into the supplied memory pointer mem，这一步是通过netbuf\_copy\_partial函数来完成的。

接着往下走，我们发现，数据的来源是在netconn的recvmbox，刚才提到过的。好了，那么是在什么地方，什么时候这个recvmbox被填充的呢？

在工程中搜索recvmbox，发现在recv\_tcp函数有这样一句：

sys\_mbox\_trypost(conn->recvmbox, p)

ok，就是它了。看看函数的原型：

\* Receive callback function for TCP netconns.

\* Posts the packet to conn->recvmbox, but doesn't delete it on errors.

static err\_t recv\_tcp(void \*arg, struct tcp\_pcb \*pcb, struct pbuf \*p, err\_t err)

好的，p是个输入参数，并且这是个静态函数，是作为tcp的接收回调用的。

嗯接着看recv\_tcp的caller：

\* Setup a tcp\_pcb with the correct callback function pointers

\* and their arguments.

\* @param conn the TCP netconn to setup

static void setup\_tcp(struct netconn \*conn)

{

struct tcp\_pcb \*pcb;

pcb = conn->pcb.tcp;

tcp\_arg(pcb, conn);

tcp\_recv(pcb, recv\_tcp);

tcp\_sent(pcb, sent\_tcp);

tcp\_poll(pcb, poll\_tcp, 4);

tcp\_err(pcb, err\_tcp);

}

哈哈，可谓是一网打尽啊。对于这个函数中的几个调用，我们看一个就好了，别的实现也差不多，就是个赋值的过程

Void tcp\_recv(struct tcp\_pcb \*pcb,

err\_t (\* recv)(void \*arg, struct tcp\_pcb \*tpcb, struct pbuf \*p, err\_t err))

{

pcb->recv = recv;

}

setup\_tcp上面也有讲过，是在newconn的时候被调用的，创建完pcb，就是调用的这个函数，以配置连接的各个回调函数。

然而到这里似乎走了一个死胡同里了，貌似没有什么地方对pcb->recv有调用的，而唯一有的就是接收TCP事件TCP\_EVENT\_RECV的宏定义中。同时，其他的几个函数也是类似的情况，例如send\_tcp函数。

真是“山穷水尽疑无路，柳暗花明又一村”啊。原来上面的也不只是死胡同。这里就要从tcp的三大接收处理函数说起了。

最底层的（在tcp层）就是tcp\_input，它是直接被ip层调用的，该函数的定义注释是这么写的：

\* The initial input processing of TCP. It verifies the TCP header, demultiplexes

\* the segment between the PCBs and passes it on to tcp\_process(), which implements

\* the TCP finite state machine. This function is called by the IP layer (in ip\_input()).

Tcp\_input又调用了tcp\_process函数做进一步的处理，它的定义注释如下：

\* Implements the TCP state machine. Called by tcp\_input. In some

\* states tcp\_receive() is called to receive data. The tcp\_seg

\* argument will be freed by the caller (tcp\_input()) unless the

\* recv\_data pointer in the pcb is set.

是的，下面是tcp\_receive函数，被tcp\_process调用

\* Called by tcp\_process. Checks if the given segment is an ACK for outstanding

\* data, and if so frees the memory of the buffered data. Next, is places the

\* segment on any of the receive queues (pcb->recved or pcb->ooseq). If the segment

\* is buffered, the pbuf is referenced by pbuf\_ref so that it will not be freed until

\* i it has been removed from the buffer.

然而光有这些调用顺序是不行的，最重要的是下面两个变量【都在tcp\_in.c中】

static u8\_t recv\_flags;

static struct pbuf \*recv\_data;

在tcp\_receive中有以下主要几句

if (inseg.p->tot\_len > 0)

{

recv\_data = inseg.p;

}

if (cseg->p->tot\_len > 0)

{

/\* Chain this pbuf onto the pbuf that we will pass to

the application. \*/

if (recv\_data)

{

pbuf\_cat(recv\_data, cseg->p);

}

else

{

recv\_data = cseg->p;

}

cseg->p = NULL;

}

下面的这个是tcp\_input中的，是在tcp\_process处理完之后的

if (recv\_data != NULL)

{

if(flags & TCP\_PSH)

{

recv\_data->flags |= PBUF\_FLAG\_PUSH;

}

/\* Notify application that data has been received. \*/

TCP\_EVENT\_RECV(pcb, recv\_data, ERR\_OK, err);

}

看最后一条语句就好了，很熟悉是吧，对了，就是上面传说中的死胡同，到此也解开了。

#### IP层实现

这一部分的实现都是在ip.c文件中【src\cor\ipv4】，可以看到在这个文件中主要实现了3个函数，ip\_input；ip\_route；ip\_output以及ip\_output\_if。下面分别来介绍它们。

这些函数可以分成两大类：接收和发送。下面就先从发送开始，首先要说的就是ip\_output函数，这个也是发送过程中最重要的一个，它是被tcp层调用的，详细可参见以上章节。

\* Simple interface to ip\_output\_if. It finds the outgoing network

\* interface and calls upon ip\_output\_if to do the actual work.

err\_t ip\_output(struct pbuf \*p, struct ip\_addr \*src, struct ip\_addr \*dest,

u8\_t ttl, u8\_t tos, u8\_t proto)

{

struct netif \*netif;

if ((netif = ip\_route(dest)) == NULL) {

return ERR\_RTE;

}

return ip\_output\_if(p, src, dest, ttl, tos, proto, netif);

}

可以看到该函数的实现就像注释所说的一样，直接调用了ip\_route和ip\_outputif两个函数。根据以往的经验，先看下netif这个结构的实现情况：

\* Generic data structure used for all lwIP network interfaces.

\* The following fields should be filled in by the initialization

\* function for the device driver: hwaddr\_len, hwaddr[], mtu, flags//这几个是要用驱动层填写的

struct netif

{

/\*\* pointer to next in linked list \*/

struct netif \*next;

/\*\* IP address configuration in network byte order \*/

struct ip\_addr ip\_addr;

struct ip\_addr netmask;

struct ip\_addr gw;

/\*\* This function is called by the network device driver

\* to pass a packet up the TCP/IP stack. \*/

err\_t (\* input)(struct pbuf \*p, struct netif \*inp);

/\*\* This function is called by the IP module when it wants

\* to send a packet on the interface. This function typically

\* first resolves the hardware address, then sends the packet. \*/

err\_t (\* output)(struct netif \*netif, struct pbuf \*p, struct ip\_addr \*ipaddr);

/\*\* This function is called by the ARP module when it wants

\* to send a packet on the interface. This function outputs

\* the pbuf as-is on the link medium. \*/

err\_t (\* linkoutput)(struct netif \*netif, struct pbuf \*p);

#if LWIP\_NETIF\_STATUS\_CALLBACK

/\*\* This function is called when the netif state is set to up or down

\*/

void (\* status\_callback)(struct netif \*netif);

#endif /\* LWIP\_NETIF\_STATUS\_CALLBACK \*/

#if LWIP\_NETIF\_LINK\_CALLBACK

/\*\* This function is called when the netif link is set to up or down

\*/

void (\* link\_callback)(struct netif \*netif);

#endif /\* LWIP\_NETIF\_LINK\_CALLBACK \*/

/\*\* This field can be set by the device driver and could point

\* to state information for the device. \*/

void \*state;

#if LWIP\_DHCP

/\*\* the DHCP client state information for this netif \*/

struct dhcp \*dhcp;

#endif /\* LWIP\_DHCP \*/

#if LWIP\_AUTOIP

/\*\* the AutoIP client state information for this netif \*/

struct autoip \*autoip;

#endif

#if LWIP\_NETIF\_HOSTNAME

/\* the hostname for this netif, NULL is a valid value \*/

char\* hostname;

#endif /\* LWIP\_NETIF\_HOSTNAME \*/

/\*\* number of bytes used in hwaddr \*/

u8\_t hwaddr\_len;

/\*\* link level hardware address of this interface \*/

u8\_t hwaddr[NETIF\_MAX\_HWADDR\_LEN];

/\*\* maximum transfer unit (in bytes) \*/

u16\_t mtu;

/\*\* flags (see NETIF\_FLAG\_ above) \*/

u8\_t flags;

/\*\* descriptive abbreviation \*/

char name[2];

/\*\* number of this interface \*/

u8\_t num;

#if LWIP\_SNMP

/\*\* link type (from "snmp\_ifType" enum from snmp.h) \*/

u8\_t link\_type;

/\*\* (estimate) link speed \*/

u32\_t link\_speed;

/\*\* timestamp at last change made (up/down) \*/

u32\_t ts;

/\*\* counters \*/

u32\_t ifinoctets;

u32\_t ifinucastpkts;

u32\_t ifinnucastpkts;

u32\_t ifindiscards;

u32\_t ifoutoctets;

u32\_t ifoutucastpkts;

u32\_t ifoutnucastpkts;

u32\_t ifoutdiscards;

#endif /\* LWIP\_SNMP \*/

#if LWIP\_IGMP

/\* This function could be called to add or delete a entry in the multicast filter table of the ethernet MAC.\*/

err\_t (\*igmp\_mac\_filter)( struct netif \*netif, struct ip\_addr \*group, u8\_t action);

#endif /\* LWIP\_IGMP \*/

#if LWIP\_NETIF\_HWADDRHINT

u8\_t \*addr\_hint;

#endif /\* LWIP\_NETIF\_HWADDRHINT \*/

};

该结构体实现在【src\include\lwip\netif.h】，注意到该结构体成员中有3个函数指针变量。好了，这个结构体先做一大体了解。用到的时候再详细讲。

接下来先看下ip\_route函数的实现：

\* Finds the appropriate network interface for a given IP address. It

\* searches the list of network interfaces linearly. A match is found

\* if the masked IP address of the network interface equals the masked

\* IP address given to the function.

struct netif \* ip\_route(struct ip\_addr \*dest)

{

struct netif \*netif;

/\* iterate through netifs \*/

for(netif = netif\_list; netif != NULL; netif = netif->next) {

/\* network mask matches? \*/

if (netif\_is\_up(netif)) {

if (ip\_addr\_netcmp(dest, &(netif->ip\_addr), &(netif->netmask))) {

/\* return netif on which to forward IP packet \*/

return netif;

}

}

}

if ((netif\_default == NULL) || (!netif\_is\_up(netif\_default)))

{

snmp\_inc\_ipoutnoroutes();

return NULL;

}

/\* no matching netif found, use default netif \*/

return netif\_default;

}

可以说这个函数的实现很简单，且作用也很容易看懂，就像其注释所说的一样。不过在这个函数中我们还是发现了一些什么，对了，就是struct netif \*netif\_list;【src\core\netif.c】的使用。既然这里都使用了这个网络接口链表，那它是在哪里被初始化的呢？

好了，首先我们发现在netif\_add函数中有对netif\_list的调用，netif\_add是被do\_netifapi\_netif\_add函数调用的，而do\_netifapi\_netif\_add是在netifapi\_netif\_add中通过netifapi\_msg被TCPIP\_NETIFAPI调用的。问题似乎很清楚，只要找到nnetifapi\_netif\_add是被谁调用的就好了，然而，搜遍整个工程也没有发现这个函数的影子，除了一个声明一个实现外。My god，又进入死胡同了？好吧，这里先标识一下，待解【见下面的解释】

我们接着看ip\_output\_if这个函数，具体函数可参考【src\core\ipv4\ip.c】。它的函数定义注释如下：

\* Sends an IP packet on a network interface. This function constructs

\* the IP header and calculates the IP header checksum. If the source

\* IP address is NULL, the IP address of the outgoing network

\* interface is filled in as source address.

\* If the destination IP address is IP\_HDRINCL, p is assumed to already

\* include an IP header and p->payload points to it instead of the data.

再看最后一句：return netif->output(netif, p, dest);

嗯，看来这个netif还是关键啊，如果估计不错的话，接收的时候也要用到这个结构的。那就看它在什么地方被赋值的吧。又经过一番搜索，看来在目前的代码中是找不到的了。查看lwip协议栈的设计与实现，特别是网络接口层的那一节，终于明白了，原来这些是要有设备驱动来参与的：【一下为引用】

当收到一个信息包时，设备驱动程序调用input指针指向的函数。网络接口通过output指针连接到设备驱动。这个指针指向设备驱动中一个向物理网络发送信息包的函数，当信息包包被发送时由IP层调用，这个字段由设备驱动的初始设置函数填充。

嗯，那就这样吧，到这里我们可以说IP层的发送流程已经走完了。

接下来就是ip层的接收过程了。刚才上面也有提到驱动设备收到包，丢给netif的input函数，这个input函数也是设备驱动层来设置的。无非有两个可能，一个是ip\_input，另外一个就是tcpip\_input。因为tcpip\_input函数的处理是最终调用到了ip\_input【在tcpip\_thread中】。按照正常情况下应该是ip\_input函数的，我们先来看下这个函数。

\* This function is called by the network interface device driver when

\* an IP packet is received. The function does the basic checks of the

\* IP header such as packet size being at least larger than the header

\* size etc. If the packet was not destined for us, the packet is

\* forwarded (using ip\_forward). The IP checksum is always checked.

原型：err\_t ip\_input(struct pbuf \*p, struct netif \*inp)

该函数大致的处理过程是：处理ip包头；找到对应的netif；检查如果是广播或多播包，则丢掉；如果是tcp协议的话就直接调用了tcp\_input函数处理数据。

到此，ip层的东西大致就说完了。最后，由于tcp和ip层的东西都说完了，所以此时我们顺便看下，tcpip的整体实现，这个主要是在src\api\tcpip.c文件中实现。我们知道发送过程是由socket直接调用的，所以这个文件中不涉及，说白了，这个文件主要是涉及到整个接收过程。这里实现的函数有tcpip\_input，和tcpip\_thread以及tcpip\_init函数。

Tcpip\_init函数很简单就是创建系统线程（sys\_thread\_new）tcpip\_thread。

Tcpip\_thread函数的注释如下：

\* The main lwIP thread. This thread has exclusive access to lwIP core functions

\* (unless access to them is not locked). Other threads communicate with this

\* thread using message boxes.

它的整个过程就是一直从mbox中取出msg，对各种msg的一个处理过程。

Tcpip\_input函数，是在tcpip\_thread中被调用的处理设备驱动接收到的信息包，并调用

ip\_input来进一步处理。

整个启动过程：

main---> vlwIPInit()

void vlwIPInit( void )

{

/\* Initialize lwIP and its interface layer. \*/

sys\_init();

mem\_init();

memp\_init();

pbuf\_init();

netif\_init();

ip\_init();

sys\_set\_state(( signed portCHAR \* ) "lwIP", lwipTCP\_STACK\_SIZE);

tcpip\_init( NULL, NULL );

sys\_set\_default\_state();

}

从上面我们知道，tcpip\_init创建tcpip\_thread

在tcpip\_thread的开始有如下代码：

(void)arg;

ip\_init();

#if LWIP\_UDP

udp\_init();

#endif

#if LWIP\_TCP

tcp\_init();

#endif

#if IP\_REASSEMBLY

sys\_timeout(1000, ip\_timer, NULL);

#endif

if (tcpip\_init\_done != NULL)

{

tcpip\_init\_done(tcpip\_init\_done\_arg);

}

下面是tcp\_init的实现

Void tcp\_init(void)

{

/\* Clear globals. \*/

tcp\_listen\_pcbs.listen\_pcbs = NULL;

tcp\_active\_pcbs = NULL;

tcp\_tw\_pcbs = NULL;

tcp\_tmp\_pcb = NULL;

/\* initialize timer \*/

tcp\_ticks = 0;

tcp\_timer = 0;

}

#### 底层结构（物理层）

们前面讲到说是ip层的发送和接收都是直接调用了底层，也就是设备驱动层的函数实现，在这里暂且称之为物理层吧。下面就接着ip层的讲，不过由于这里的设备驱动各平台的都不一样，为此，我们选择ARM9\_STR91X\_IAR这个Demo作为实例，该平台的网络设备驱动在\library\source\91x\_enet.c文件中。而ethernetif.c文件就是我们需要的，它是连接设备驱动程序与ip层的桥梁。

Ethernetif.c文件中提供的函数主要有以下这么几个：

(1)    low\_level\_init

(2)    low\_level\_input

(3)    low\_level\_output

(4)    ethernetif\_init

(5)    ethernetif\_input

(6)    ethernetif\_output

这里对外的接口只有ethernetif\_init函数，它是main函数中通过

netif\_add( &EMAC\_if, &xIpAddr, &xNetMast, &xGateway, NULL, ethernetif\_init, tcpip\_input );来被调用的。我们可以清楚的看到，tcpip\_input的使用，它就是被用来当有数据接收的时候被调用的以使接收到的数据进入tcpip协议栈。

在netif\_add函数中，我们可以看到

netif->input = input;

if (init(netif) != ERR\_OK)

{

return NULL;

}

Ok，从这里就进入到ethernetif\_init函数了，在这个函数中，我们主要看以下几句：

netif->output = ethernetif\_output;

netif->linkoutput = low\_level\_output;

low\_level\_init(netif);

etharp\_init();

可以看到，netif->output 和netif->linkoutput被赋值了，这个很重要的，等会再说。

好，再接着看low\_level\_init函数

s\_pxNetIf = netif;//对全局变量s\_pxNetIf 赋初值

ENET\_InitClocksGPIO();

ENET\_Init();

ENET\_Start();//这3句是对网络设备的寄存等的配置

xTaskCreate( ethernetif\_input, ( signed portCHAR \* ) "ETH\_INT", netifINTERFACE\_TASK\_STACK\_SIZE, NULL, netifINTERFACE\_TASK\_PRIORITY, NULL );

以ethernet\_input创建task，这个函数也很有意思，首先可以看到的是一个无限循环，在循环体中有以下调用：

p = low\_level\_input( s\_pxNetIf );

s\_pxNetIf->input(p, s\_pxNetIf);//tcpip\_input

虽然有了这两句，还不是很清楚，可以确定的是后一句是把接收到的数据送入tcpip协议栈处理，为此，我们想到上一句是从硬件读出数据。看下具体的low\_level\_input函数实现：

len = ENET\_HandleRxPkt(s\_rxBuff);

这个函数很好理解，主要的是上面的那一句。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Function Name : ENET\_HandleRxPkt

\* Description : receive a packet and copy it to memory pointed by ppkt.

\* Input : ppkt: pointer on application receive buffer.

\* Output : None

\* Return : ENET\_NOK - If there is no packet

\* : ENET\_OK - If there is a packet

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

u32 ENET\_HandleRxPkt ( void \*ppkt)

{

ENET\_DMADSCRBase \*pDescr;

u16 size;

static int iNextRx = 0;

if( dmaRxDscrBase[ iNextRx ].dmaPackStatus &

DMA\_DSCR\_RX\_STATUS\_VALID\_MSK )

{

return 0;

}

pDescr = &dmaRxDscrBase[ iNextRx ];

/\*Get the size of the packet\*/

size = ((pDescr->dmaPackStatus & 0x7ff) - 4);

//MEMCOPY\_L2S\_BY4((u8\*)ppkt, RxBuff, size); /\*optimized memcopy function\*/

**memcpy(ppkt, RxBuff[iNextRx], size);** //string.h library\*/

/\* Give the buffer back to ENET \*/

pDescr->dmaPackStatus = DMA\_DSCR\_RX\_STATUS\_VALID\_MSK;

iNextRx++;

if( iNextRx >= ENET\_NUM\_RX\_BUFFERS )

{

iNextRx = 0;

}

/\* Return no error \*/

return size;

}

这个函数也很好理解，是从DMA中直接拷贝数据到指定的pBuf。至此，input过程完事了，从代码调用流程上看真是千回百转，一会low\_level，一会ethernetif。不过，总体来说是系统通过一个task（ethernetif\_input）轮询检查DMA控制器的状态以判断是否有数据接收到。

下面再研究下output过程。Output过程是由应用程序以主动方式触发的，经过前面几篇的介绍，我们知道发送的数据后来被传递给了ip\_output\_if函数。我们就接着这个函数看，它直接调用了netif->output函数，刚才我们看到在ethernetif\_init中有对这个变量【函数指针】赋值，它就是ethernetif\_output。它倒是简单，直接return etharp\_output（）；而它有在最后调用了etharp\_send\_ip，在这个函数的最后调用了return netif->linkoutput(netif, p);好了终于找到根了这里的linkoutput函数也就是low\_level\_output，果然有如下调用：

memcpy(&TxBuff[l], (u8\_t\*)q->payload, q->len);

还记得ENET\_Init嘛，它的函数实现中有如下两句调用：

ENET\_TxDscrInit();

ENET\_RxDscrInit();

它们就是初始化DMA发送和接收的地址的，也就是上面所说的TxBuff和RxBuff。

OK，大功告成，自上而下的LWIP大体流程都说到了。如有以后碰到什么需要注意的细节之类，再补上吧。

#### SOCKET编程示例

首先要注意的一点是sys\_thread\_new和xTaskCreate的区别。我们知道xTaskCreate是用来创建一个task任务的。类似于windows下的process。

/\*

Starts a new thread with priority "prio" that will begin its execution in the

function "thread()". The "arg" argument will be passed as an argument to the

thread() function. The id of the new thread is returned. Both the id and

the priority are system dependent.

\*/

sys\_thread\_t sys\_thread\_new(void (\* thread)(void \*arg), void \*arg, int prio)

{

xTaskHandle CreatedTask;

int result;

result = xTaskCreate(thread, ( signed portCHAR \* ) s\_sys\_arch\_state.cTaskName, s\_sys\_arch\_state.nStackDepth, arg, prio, &CreatedTask );

// For each task created, store the task handle (pid) in the timers array.

// This scheme doesn't allow for threads to be deleted

timeoutlist[nextthread++].pid = CreatedTask;

if(result == pdPASS)

{

++s\_sys\_arch\_state.nTaskCount;

return CreatedTask;

}

else

{

return NULL;

}

}

很显然，sys\_thread\_new最终对task的创建也是通过xTaskCreate来实现的。但是请注意不同点，前者会把新创建的task的pid放入到一个timeoutlist的链表中。

起初我也没有主要这个小问题，直到用socket系列函数的select的时候，一直过不去，而是abort中断了，具体原因请看select函数实现，其中sys\_arch\_timeouts的调用就是获取当前task的timeout，如果没有用sys\_thread\_new创建的话，这里就没有我用来调用select的当前task，所以就有了以后的abort中断了。

下面给一个socket编程的实现

static void vLWIPSendTask( void \*pvParameters )

{

int listenfd;

int remotefd;

int len;

struct sockaddr\_in local\_addr,remote\_addr;

fd\_set readset;

fd\_set writeset;

struct timeval timeout;

timeout.tv\_sec = 1;

timeout.tv\_usec = 0;

//struct lwip\_socket\* sock;

listenfd = socket(AF\_INET,SOCK\_STREAM,0);

local\_addr.sin\_family = AF\_INET;

local\_addr.sin\_port = htons(80);

local\_addr.sin\_len = sizeof(local\_addr);

local\_addr.sin\_addr.s\_addr = INADDR\_ANY;

if (bind(listenfd, (struct sockaddr \*) &local\_addr, sizeof(local\_addr)) < 0)

{

return ;

}

if (listen(listenfd, 1) == -1)

{

return;

}

len = sizeof(remote\_addr);

while(1)

{

//这里注意一下，lwip的阻塞不是在listen函数，而是accept

remotefd = accept(listenfd, (struct sockaddr \*)&remote\_addr, &len);

//close(listenfd);

//listenfd = -1;

//getpeername(remotefd, (struct sockaddr \*)&remote\_addr, &len);

if(remotefd != -1)

{

int ret;

send(remotefd,"start to work!\r\n",16,0);

for(;;)

{

FD\_ZERO(&readset);

FD\_ZERO(&writeset);

FD\_SET(remotefd, &readset);

FD\_SET(remotefd, &writeset);

ret = lwip\_select(remotefd+1, &readset, &writeset, 0, &timeout);

if(ret > 0)

{

if (FD\_ISSET(remotefd, &readset))

{

memset(buf,0,50);

if(recv(remotefd,buf,50,0) <= 0)

{

close(remotefd);

remotefd = -1;

break;

}

else

{

int i = strlen(buf);

send(remotefd,buf,i ,0);

}

}

/\*

else if(FD\_ISSET(remotefd, &writeset))

{

send(remotefd,"this is time to send!\r\n",25,0);

}

\*/

}

else if(ret < 0)

{

close(remotefd);

remotefd = -1;

break;

}

}

}

}

vTaskDelete( NULL );

}

对的，这是作为server端实现的，可以和通用的socket客户端配。经过测试至少可以与windows上的socket程序匹配（接收，发送数据）。

## 第三部分 其他

MEM\_SIZE:内存堆的大小。如果应用程序要发送许多需要内存拷贝的数据，这个值应被设置得大些。

MEMP\_NUM\_PBUF:memp结构的pbufs数目。如果应用程序要发送许多ROM类型(或者其他静态内存)以外的数据,这个值应被设置得大些。

MEMP\_NUM\_UDP\_PCB:UDP协议控制块数目。每个活动的UDP“连接”对应一个。

MEMP\_NUM\_TCP\_PCB:同时活动的TCP连接数

MEMP\_NUM\_TCP\_PCB\_LISTEN:监听TCP连接数

MEMP\_NUM\_TCP\_SEG:同时排队TCP分段数

MEMP\_NUM\_SYS\_TIMEOUT:同时活动超时数

MEMP\_NUM\_NETBUF:结构netbuf数

MEMP\_NUM\_NETCONN:结构netconn数

MEMP\_NUM\_APIMSG:结构api\_msg数，用于在TCP/IP栈和顺序程序间通信。

MEMP\_NUM\_TCPIPMSG:结构tcpip\_msg数，用于顺序API通信和收包。详见src/api/tcpip.c。

PBUF\_POOL\_SIZE:pbuf池中缓冲区个数

PBUF\_POOL\_BUFSIZE:pbuf池中每个pbuf的大小

PBUF\_LINK\_HLEN:应该分配给一个链路级包头的字节数

TCP\_MSS TCP 最大分段长度

TCP\_SND\_BUF   设成20会造成链路断开，可能CDMA对窗口内数据大小有限制。抑或是队列长度要大于50。

TCP\_SND\_QUEUELEN         1200

TCP发送器缓冲区空间(pbufs)。这个值必须至少=2\*TCP\_SND\_BUF/TCP\_MSS以便正常工作。实测最好保持2倍关系，不要大于2倍。怀疑注释有误，不是至少，是至多。

TCP\_WND         TCP接收窗口

TCP\_MAXRTX   最大重发数据分段数

TCP\_SYNMAXRTX            最大重发SYN段数

IP\_FORWARD 如果你希望拥有在多个网络接口间进行IP包转发的能力，那么定义IP\_FORWARD为1。

IP\_OPTIONS  如果为1，IP选项被允许(但不解析)。如果为0，所有带IP选项的包均被抛掉。

DHCP\_DOES\_ARP\_CHECK  置1，如果你需要在给定地址上进行ARP检测(推荐)

参考文献：

嵌入式网络系统设计-基于Atmel ARM7系列