# 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；

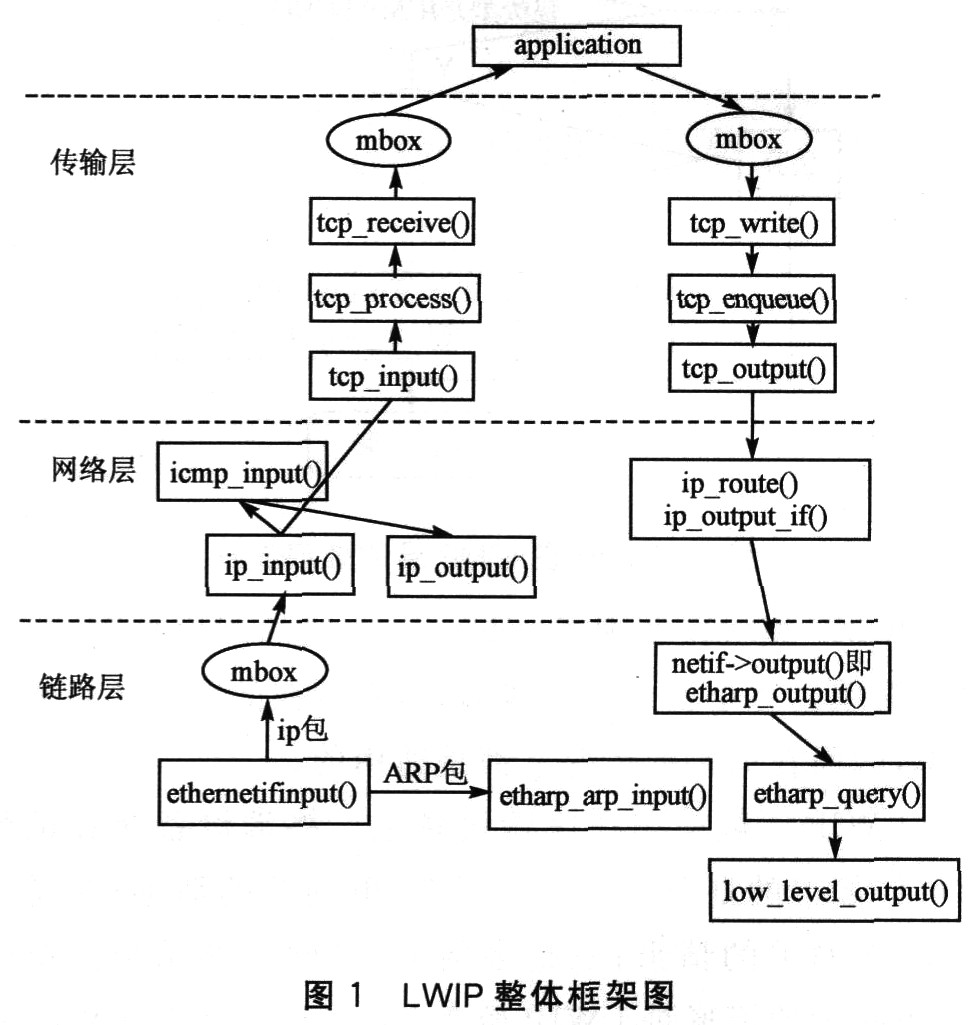
(2)正式的API。这种实现方式是在系统中单独建立了一个TCP/IP任务，由这个任务调用RAW API来处理网络通信，其它的网络任务都是利用消息机制与这个任务通信来收发数据。这也是本文中使用的方法；

(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。

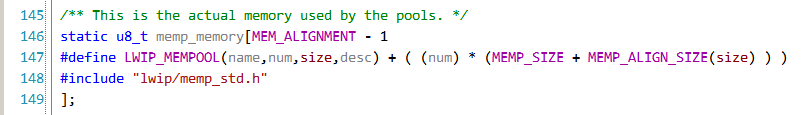
此外，LwIP为内部的一些结构设计了专用的内存池，比如netconn，协议控制块，数据包等，这些都是在memp.c/memp.h里用内存池进行管理的。

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：

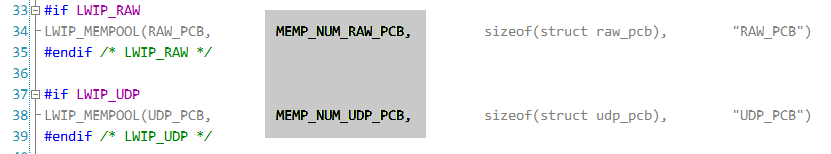
memp\_memory[]

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。

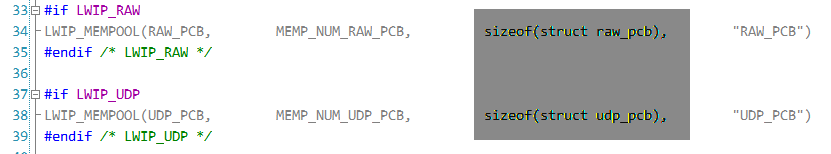
我们来看代码：



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。flags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。

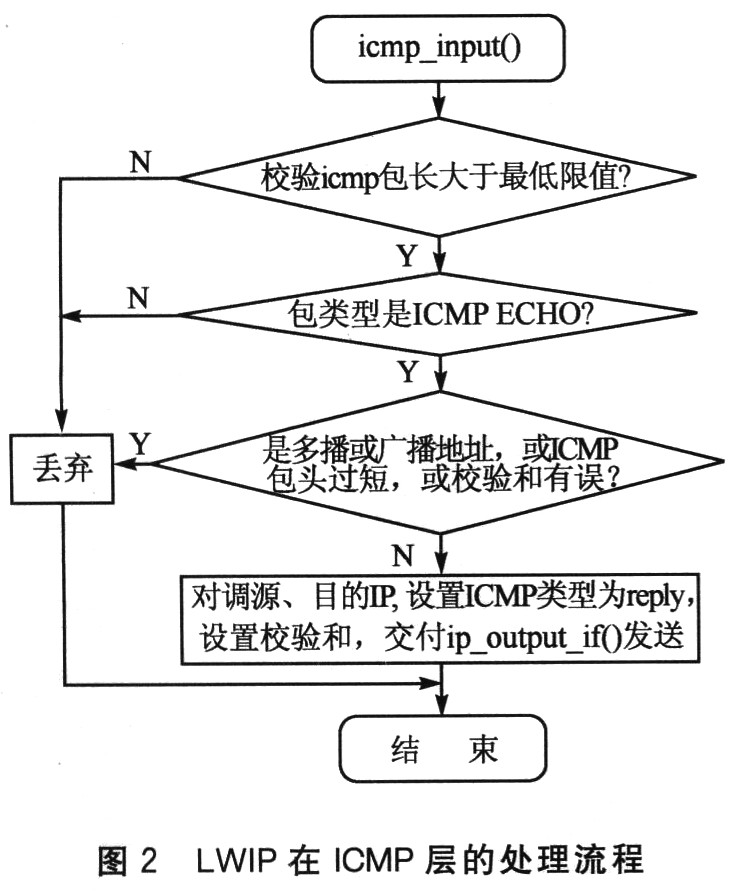
|  |  |  |
| --- | --- | --- |
| **主机名称** | **IP地址** | **MAC地址** |
| A | 192.168.38.10 | 00-AA-00-62-D2-02 |
| B | 192.168.38.11 | 00-BB-00-62-C2-02 |
| C | 192.168.38.12 | 00-CC-00-62-C2-02 |
| D | 192.168.38.13 | 00-DD-00-62-C2-02 |
| E | 192.168.38.14 | 00-EE-00-62-C2-02 |
| ... | ... |  |

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分包

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。

## 第三部分 其他

参考文献：

嵌入式网络系统设计-基于Atmel ARM7系列