

STM32F4 LWIP 开发手册 V3.0

—ALIENTEK STM32F4 LWIP 开发教程



淘宝店铺 1: <http://eboard.taobao.com>

淘宝店铺 2: <http://openedv.taobao.com>

技术支持论坛 (开源电子网) : www.openedv.com

官方网站: www.alientek.com

最新资料下载链接: <http://www.openedv.com/posts/list/13912.htm>

E-mail: 389063473@qq.com QQ: [389063473](https://www.qq.com/)

咨询电话: [020-38271790](tel:020-38271790)

传真号码: [020-36773971](tel:020-36773971)

团队: [正点原子团队](#)

正点原子, 做最全面、最优秀的嵌入式开发平台软硬件供应商。

友情提示

如果您想及时免费获取“正点原子”最新资讯, 敬请关注正点原子微信公众平台, 我们将及时给您发布最新消息和重要资料。



关注方法:

- (1) 微信“扫一扫”, 扫描右侧二维码, 添加关注
- (2) 微信→添加朋友→公众号→输入“正点原子”→关注
- (3) 微信→添加朋友→输入“alientek_stm32”→关注



文档更新说明

版本	版本更新说明	负责人	校审	发布日期
V1.0	初稿： 第一章 LWIP 无操作系统移植 第二章 LWIP 带操作系统移植 第三章 RAW 编程接口 UDP 实验 第四章 RAW 编程接口 TCP 客户端实验 第五章 RAW 编程接口 TCP 服务器实验 第六章 RAW 编程接口 Web Server 实验 第七章 NETCONN 编程接口简介 第八章 NETOCNN 编程接口 UDP 实验 第九章 NETCONN 编程接口 TCP 客户端实验 第十章 NETCONN 编程接口 TCP 服务器实验	左忠凯	刘军	2014.11.4
V2.0	修改： 修改上一版中出现的问题	左忠凯	刘军	2014.12.10
V2.1	修改： 修改文档中的一些错误。	左忠凯	刘军	2015.8.3
V3.0	添加： 第二章 2.2.2 和 2.3.2 小节 第八章 8.1.2 小节 第九章 9.1.2 小节 第十章 10.1.2 小节 第十一章 11.1.2 小节	左忠凯	刘军	2016.12.27

目录

第一章 LWIP 无操作系统移植.....	6
1.1 TCP/IP 协议以及 LWIP 简介	7
1.2 硬件设计	15
1.3 无操作系统 LWIP 移植	17
1.3.1 移植准备工作	17
1.3.2 添加及修改 ST 以太网库	19
1.3.3 添加网卡驱动程序	22
1.3.4 LWIP 数据包和网络接口管理	33
1.3.5 添加 LWIP 源文件	36
1.3.6 添加中间文件	37
1.3.7 LWIP 源码修改	47
1.3.8 LWIP 的裁剪与配置	52
1.4 软件设计	55
1.5 下载验证	57
第二章 LWIP 带操作系统移植.....	63
2.1 移植简介	64
2.2 带操作系统 LWIP 移植	65
2.2.1 UCOSII+LWIP 移植	65
2.2.2 UCOSIII+LWIP 移植	78
2.3 软件设计	89
2.4 下载验证	90
第三章 RAW 编程接口 UDP 实验.....	93
3.1 RAW 编程接口 UDP 简介	94
3.2 软件设计	95
3.3 下载验证	101
第四章 RAW 编程接口 TCP 客户端实验	104
4.1 RAW 编程接口 TCP 简介	105
4.2 软件设计	107
4.3 下载验证	114
第五章 RAW 编程接口 TCP 服务器实验	117
5.1 RAW 编程接口 TCP 简介	118
5.2 软件设计	118
5.3 下载验证	123
第六章 RAW 编程接口 WEB SERVER 实验	126
6.1 WEB SERVER 文件以及相关技术简介	127
6.2 软件设计	130

6.3 下载验证	135
第七章 NETCONN 编程接口简介	137
7.1 NETBUF 数据缓冲区	138
7.2 NETCONN 连接结构	139
7.3 NETCONN API 函数	140
第八章 NETCONN 编程接口 UDP 实验	143
8.1 软件设计	144
8.1.1 UCOSII 版本	144
8.1.2 UCOSIII 版本	148
8.2 下载验证	148
第九章 NETCONN 编程接口 TCP 客户端实验	151
9.1 软件设计	152
9.1.1 UCOSII 版本	152
9.1.2 UCOSIII 版本	154
9.2 下载验证	155
第十章 NETCONN 编程接口 TCP 服务器实验	157
10.1 软件设计	158
10.1.1 UCOSII 版本	158
10.1.2 UCOSIII 版本	160
10.2 下载验证	160

第一章 LWIP 无操作系统移植

本章,我们将向大家介绍 ALIENETK 探索者 STM32F4 开发板以太网接口及其使用。STM32F407 内置以太网 MAC 层因此可以用来做以太网通信,但是要想实现以太网还需要 PHY 层芯片和 TCP/IP 协议栈的支持。ALIENETK 板载一颗 PHY 层芯片。本章主要介绍基本的以太网知识以及 LWIP 在 ALIENETK STM32F407 开发板上的移植。本章将分为如下几个部分:

- 1.1 TCP/IP 协议、LAN8720 以及 LWIP 简介
- 1.2 硬件设计
- 1.3 LWIP 移植
- 1.4 软件设计
- 1.5 下载验证

1.1 TCP/IP 协议以及 LWIP 简介

1) TCP/IP 协议简介

TCP/IP 中文名为传输控制协议/因特网互联协议，又名网络通讯协议，是 Internet 最基本的协议、Internet 国际互联网络的基础，由网络层的 IP 协议和传输层的 TCP 协议组成。TCP/IP 定义了电子设备如何连入因特网，以及数据如何在它们之间传输的标准。协议采用了 4 层的层级结构，每一层都呼叫它的下一层所提供的协议来完成自己的需求。通俗而言：TCP 负责发现传输的问题，一有问题就发出信号，要求重新传输，直到所有数据安全正确地传输到目的地。而 IP 是给因特网的每一台联网设备规定一个地址。

TCP/IP 协议不是 TCP 和 IP 这两个协议的合称，而是指因特网整个 TCP/IP 协议族。从协议分层模型方面来讲，TCP/IP 由四个层次组成：网络接口层、网络层、传输层、应用层。OSI 是传统的开放式系统互连参考模型，该模型将 TCP/IP 分为七层：物理层、数据链路层（网络接口层）、网络层（网络层）、传输层（传输层）、会话层、表示层和应用层（应用层）。TCP/IP 模型与 OSI 模型对比如表 1.1.1 所示。

编号	OSI 模型	TCP/IP 模型
1	应用层	应用层
2	表示层	
3	会话层	
4	传输层	传输层
5	网络层	互联层
6	数据链路层	链路层
7	物理层	

表 1.1.1 TCP/IP 模型与 OSI 模型对比

在我们的 LWIP 实验中 PHY 层芯片 LAN8720 相当于物理层，STM32F407 自带的 MAC 层相当于数据链路层，而 LWIP 提供的就是网络层、传输层的功能，应用层是需要用户自己根据自己想要的功能去实现的。

2) LAN8720 简介

LAN8720 是低功耗的 10/100M 以太网 PHY 层芯片，I/O 引脚电压符合 IEEE802.3-2005 标准。LAN8720 支持通过 RMII 接口与以太网 MAC 层通信，内置 10-BASE-T/100BASE-TX 全双工传输模块，支持 10Mbps 和 100Mbps。LAN8720 可以通过自协商的方式与目的主机最佳的连接方式(速度和双工模式)。支持 HP Auto-MDIX 自动翻转功能，无需更换网线即可将连接更改为直连或交叉连接。LAN8720 的主要特点如下：

- 高性能的 10/100M 以太网传输模块
- 支持 RMII 接口以减少引脚数
- 支持全双工和半双工模式
- 两个状态 LED 输出
- 可以使用 25M 晶振以降低成本
- 支持自协商模式
- 支持 HP Auto-MDIX 自动翻转功能
- 支持 SMI 串行管理接口
- 支持 MAC 接口

LAN8720 功能框图如图 1.1.1 所示。

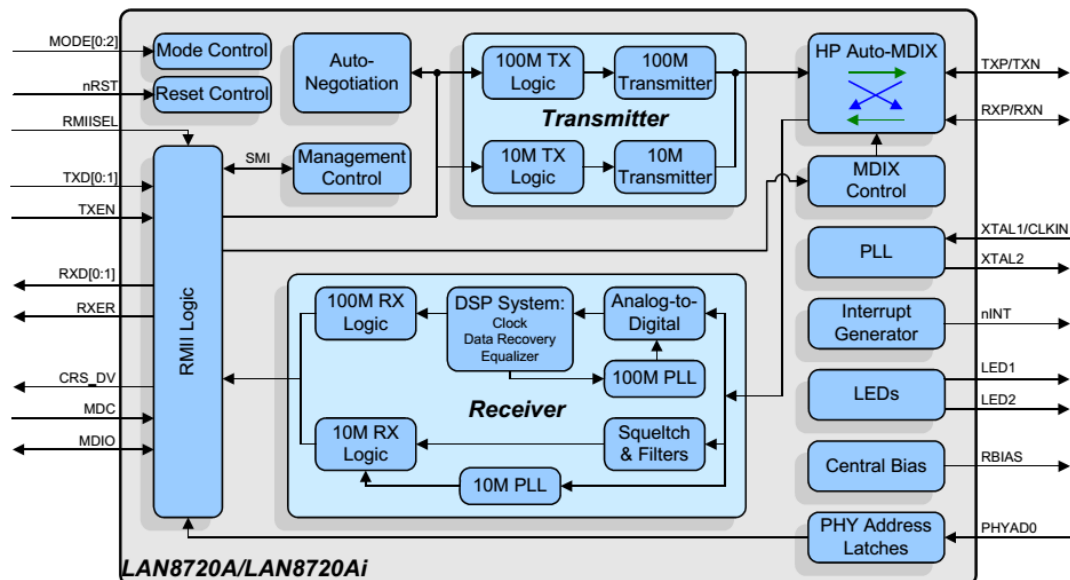


图 1.1.1 LAN8720 功能框图

1、LAN8720 中断管理

LAN8720 的器件管理接口支持非 IEEE 802.3 规范的中断功能。当一个中断事件发生并且相应事件的中断位使能，LAN8720 就会在 nINT(14 脚)产生一个低电平有效的中断信号。LAN8720 的中断系统提供两种中断模式：主中断模式和复用中断模式。主中断模式是默认中断模式，LAN8720 上电或复位后就工作在主中断模式，当模式控制/状态寄存器(十进制地址为 17)的 ALTINT 为 0 是 LAN8720 工作在主模式，当 ALTINT 为 1 时工作在复用中断模式。我们的 STM32F407 开发板中并未用到中断功能，关于中断的具体用法可以参考 LAN8720 数据手册的 29, 30 页。

2、PHY 地址设置

MAC 层通过 SMI 总线对 PHY 进行读写操作，SMI 可以控制 32 个 PHY 芯片，通过不同的 PHY 芯片地址来对不同的 PHY 操作。LAN8720 通过设置 RXER/PHYAD0 引脚来设置其 PHY 地址，默认情况下为 0，其地址设置如表 1.1.2 所示。我们 STM32F407 开发板使用的是默认地址，也就是 0X00。

RXER/PHYAD0 引脚状态	PHY 地址
上拉	0X01
下拉(默认)	0X00

表 1.1.2 LAN8720 地址设置

3、nINT/REFCLKO 配置

nINTSEL 引脚(2 号引脚)用于设置 nINT/REFCLKO 引脚(14 号引脚)的功能。nINTSEL 配置如表 1.1.3 所示。我们 STM32F407 开发板使用的是 REF_CLK Out 模式。

nINTSEL 引脚值	模式	nINT/REFCLKO 引脚功能
nINTSEL= 0	REF_CLK Out 模式	nINT/REFCLKO 作为 REF_CLK 时钟源
nINTSEL = 1	REF_CLK In 模式	nINT/REFCLKO 作为中断引脚

表 1.1.3 nINTSEL 配置

当工作在 REF_CLK In 模式时，50MHz 的外部时钟信号应接到 LAN8720 的 XTAL1/CKIN 引脚(5 号引脚)和 STM32F407 的 RMII_REF_CLK(PA1)引脚上，如图 1.1.2 所示。

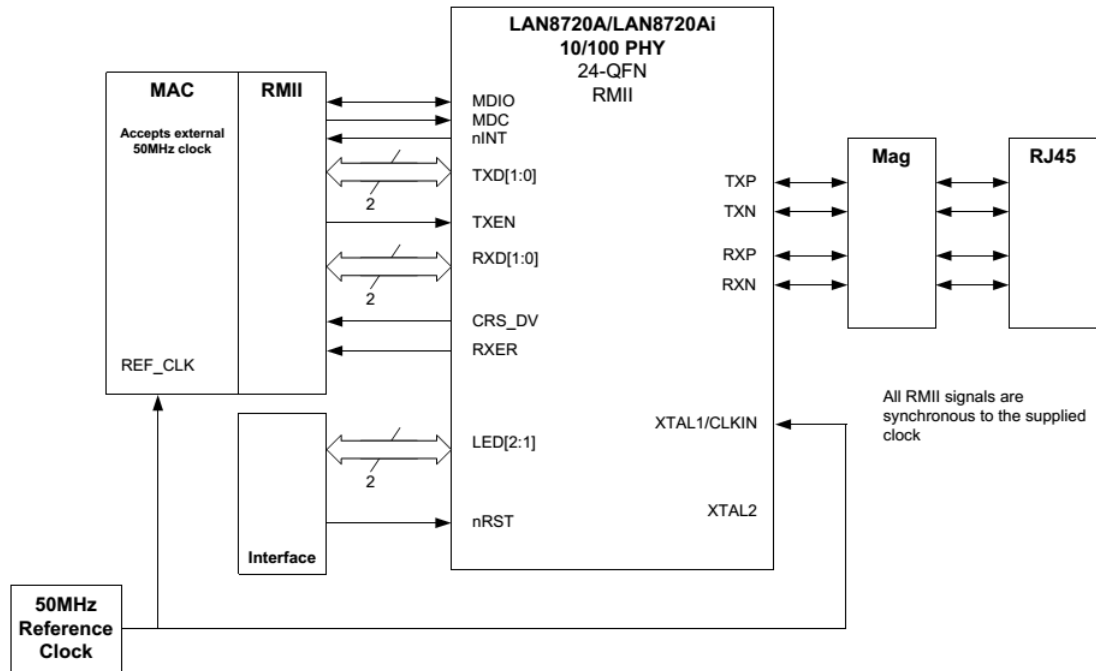


图 1.1.2 REF_CLK 连接外部 50MHz 时钟信号

为了降低成本，LAN8720 可以从外部的 25MHz 的晶振中产生 REF_CLK 时钟。到要使用此功能时应工作在 REF_CLK Out 模式时。当工作在 REF_CLO Out 模式时 REF_CLK 的时钟源如图 1.1.3 所示。

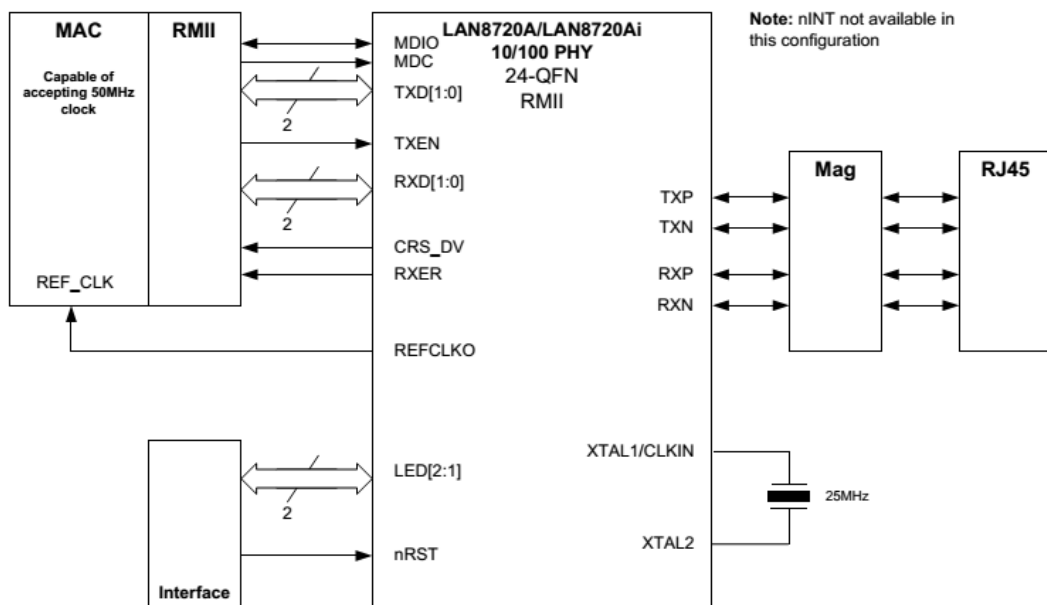


图 1.1.3 REF_CLK Out 模式时的 REF_CLK 时钟源

4、LAN8720 内部寄存器

PHY 是由 IEEE 802.3 定义的，一般通过 SMI 对 PHY 进行管理和控制,也就是读写 PHY 内部寄存器。PHY 寄存器的地址空间为 5 位，可以定义 0~31 共 32 个寄存器，但是随之 PHY 芯片功能的增加，很多 PHY 芯片采用分页技术来扩展地址空间，定义更多的寄存器，在这里我们不讨论这种情况。IEEE 802.3 定义了 0~15 这 16 个寄存器的功能，16~31 寄存器由芯片制造商自由定义。

在 LAN8720 有很多寄存器，在这里我们只介绍几个用到的寄存器（括号内为寄存器地址，此处使用十进制表示）：BCR(0)，BSR(1)，PHY 特殊功能寄存器(31)这三个寄存器。

BCR 寄存器各位介绍如表 1.1.4 所示。

位	描述	类型
15	软件复位 1: 软件复位，此位自动清零	R/W
14	回测 0: 正常运行 1: 回测模式	R/W
13	速度选择 0: 10Mbps 1: 100Mbps 注意：当使用自动协商功能时此位失能	R/W
12	自动协商功能 0: 关闭自动协商功能 1: 打开自动协商功能	R/W
11	掉电（power down） 0: 正常运行 1: 进入掉电模式 注意:进入掉电模式前自动协商必须失能	R/W
10	隔离 0: 正常运行 1: PHY 的 RMII 接口电气隔离	R/W
9	重启自动协商功能 0: 正常运行 1: 重启自动协商功能 注意:此位会被自动清零	R/W SC
8	双工模式 0: 半双工 1: 全双工 注意:开启自动协商功能后此位失效	R/W
7:0	保留	RO

表 1.1.4 BCR 寄存器

我们配置 LAN8720 其实就是配置 BCR 寄存器，我们通过调用 ST 官方的以太网库的 ETH_ReadPHYRegister 和 ETH_WritePHYRegister 函数来完成对 PHY 芯片寄存器的读写操作。在 ST 官方以太网库的 stm32f4x7_eth.h 中已经定义了 BCR 和 BSR 寄存器,代码如下:

```

/** @defgroup PHY_Register_address
 * @{
 */
#define PHY_BCR 0 /*!< Transceiver Basic Control Register */
#define PHY_BSR 1 /*!< Transceiver Basic Status Register */

```

在以后要讲的 LAN8720 初始化的中并没有看到我们直接操作 PHY 的寄存器，原因就在这里当我们调用 ETH_Init()函数以后就会根据我们输入的参数配置 LAN8720 的相应寄存器。

BSR 寄存器各个位介绍如表 1.1.5 所示

位	描述	类型
15	100BAST-T4 0: 不支持 T4 1: 支持 T4	RO
14	100BAST-TX 全双工 0: 不支持 TX 全双工 1: 支持 TX 全双工	RO
13	100BAST-TX 半双工 0: 不支持 TX 半双工 1: 支持 TX 半双工	RO
12	10BAST-T 全双工 0: 不支持 10Mbps 全双工 1: 支持 10Mbps 全双工	RO
11	10BAST-T 半双工 0: 不支持 10Mbps 半双工 1: 支持 10Mbps 半双工	RO
10:6	保留	RO
5	自协商功能完成 0: 自动协商未完成 1: 自动协商完成	RO
4	远端错误 0: 无远端错误 1: 检测到远端错误	RO/HL
3	自协商功能 0: 不能执行自协商功能 1: 可以执行自协商功能	RO
2	连接状态 0: 连接断开 1: 连接建立	RO/LL
1	Jabber 检测 0: 未检测到 jabber 1: 检测到 jabber	RO/LH
0	扩展功能 0: 不支持扩展寄存器 1: 支持扩展寄存器	RO

表 1.1.5 BSR 寄存器

BSR 寄存器为 LAN8720 的状态寄存器，通过读取该寄存器的值我们可以得到当前的连接速度、双工状态和连接状态等功能。在 ST 官方以太网库中的 ETH_Init 函数通过读取 PHY 的 BSR 寄存器来判断连接是否建立、自协商是否完成等信息，在 stm32f4x7_eth.c 文件中多次使用

到 BCR 和 BSR 寄存器。

接下来介绍的是 LAN8720 特殊功能寄存器，此寄存器的各个位如表 1.1.6 所示：

位	描述	类型
15:13	保留	RO
12	自协商完成 0: 自协商未完成或者自协商关闭 1: 自协商完成	RO
11:5	保留	R/W
4:2	速度指示 001: 10BASE-T 半双工 101: 10BASE-T 全双工 010: 100BASE-TX 半双工 110: 100BASE-TX 全双工	RO
1:0	保留	RO

表 1.1.6 LAN8720 特殊功能寄存器

在特殊功能寄存器中我们关心的是 bit2~bit4 这三位，因为通过这 3 位来确定连接的状态和速度。在 ETH_Init 函数中通过读取这 3 位的值来设置 BCR 寄存器的 bit8 和 bit13。由于特殊功能寄存器不属于 IEEE802.3 规定的前 16 个寄存器，所以每个厂家的可能不同，这个需要用户根自己实际使用的 PHY 芯片去修改，在 stm32f4x7_eth_conf.h 文件中定义，代码如下：

```
/* PHY 配置块 *****/
/* PHY 复位延时*/
#define PHY_RESET_DELAY    ((uint32_t)0x000FFFFF)

/* PHY 配置延时*/
#define PHY_CONFIG_DELAY    ((uint32_t)0x00FFFFF)

//PHY 的状态寄存器,用户需要根据自己的 PHY 芯片来改变 PHY_SR 的值
//define PHY_SR    ((uint16_t)16) //DP83848 的 PHY 状态寄存器
#define PHY_SR    ((uint16_t)31) //LAN8720 的 PHY 状态寄存器地址

//速度和双工类型的值,用户需要根据自己的 PHY 芯片来设置
//define PHY_SPEED_STATUS    ((uint16_t)0x0002) //DP83848 PHY 速度值
//define PHY_DUPLEX_STATUS    ((uint16_t)0x0004) //DP83848 PHY 连接状态值
#define PHY_SPEED_STATUS    ((uint16_t)0x0004) //LAN8720 PHY 速度值
#define PHY_DUPLEX_STATUS    ((uint16_t)0x0010) //LAN8720 PHY 连接状态值
```

通过以上代码可以看出定义了 PHY_SR，读取 PHY_SR 中的值与 PHY_SPEED_STATUS 和 PHY_DUPLEX_STATUS 进行计算来得到速度值和双工状态。关于 LAN8720 的寄存器就讲解到这里。

3) 站管理接口：SMI

站管理接口(SMI) 允许应用程序通过 2 条线：时钟(MDC)和数据线(MDIO)访问任意 PHY 寄存器。该接口支持访问多达 32 个 PHY。应用程序可以从 32 个 PHY 中选择一个 PHY，然后从任意 PHY 包含的 32 个寄存器中选择一个寄存器，发送控制数据或接收状态信息。任意给定时间内只能对一个 PHY 中的一个寄存器进行寻址。在 MAC 对 PHY 进行读写操作的时候，应

用程序不能修改 MII 的地址寄存器和 MII 的数据寄存器。在此期间对 MII 地址寄存器或 MII 数据寄存器执行的写操作将会被忽略。关于 SMI 接口的详细介绍大家可以参考 SSTM32F4xx 中文参考手册(RM009)的 824 页。

4) 介质独立接口：MII

MI 用于 MAC 层与 PHY 层进行数据传输。STM32F407 通过 MII 与 PHY 层芯片的连接如图 1.1.7 所示。

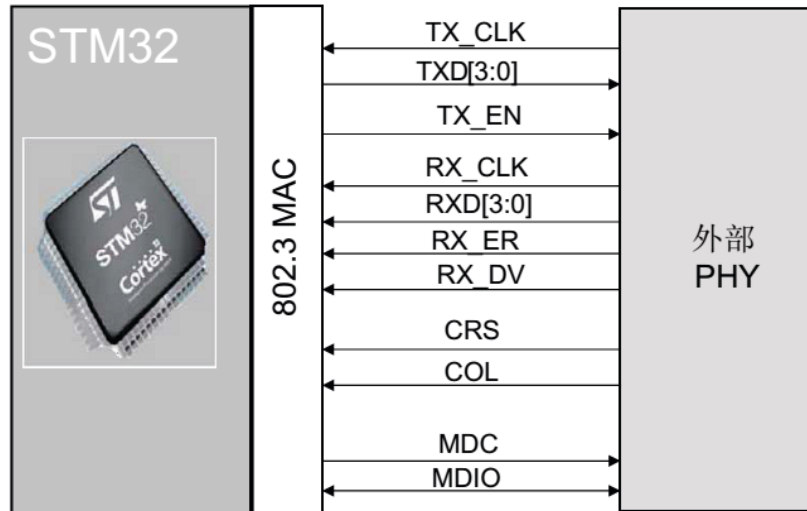


图 1.1.7 STM32F407 与 PHY 层芯片连接

- MII_TX_CLK: 连续时钟信号。该信号提供进行 TX 数据传输时的参考时序。标称频率为：速率为 10 Mbit/s 时为 2.5 MHz；速率为 100 Mbit/s 时为 25 MHz。
- MII_RX_CLK: 连续时钟信号。该信号提供进行 RX 数据传输时的参考时序。标称频率为：速率为 10 Mbit/s 时为 2.5 MHz；速率为 100 Mbit/s 时为 25 MHz。
- MII_TX_EN: 发送使能信号。
- MII_TXD[3:0]: 数据发送信号。该信号是 4 个一组的数据信号，
- MII_CRs: 载波侦听信号。
- MII_COL: 冲突检测信号。
- MII_RXD[3:0]: 数据接收信号。该信号是 4 个一组的数据信号
- MII_RX_DV: 接收数据有效信号。
- MII_RX_ER: 接收错误信号。该信号必须保持一个或多个周期(MII_RX_CLK)，从而向 MAC 子层指示在帧的某处检测到错误。

5) 精简介质独立接口：RMII

精简介质独立接口(RMII) 规范降低 10/100 Mbit/s 下微控制器以太网外设与外部 PHY 间的引脚数。根据 IEEE 802.3u 标准，MII 包括 16 个数据和控制信号的引脚。RMII 规范将引脚数减少为 7 个。

STM32F407 通过 RMII 接口与 PHY 层芯片的连接如图 1.1.8 所示。因为 RMII 相比 MII，其发送和接收都少了两条线。因此要达到 10Mbit/s 的速度，其时钟频率应为 5MHz，同理要达到 100Mbit/s 的速度其时钟频率应为 50MHz，而 MII 接口分别为 2.5MHz 和 25MHz。我们的 STM32F407 开发板采用 RMII 接口连接 LAN8720。

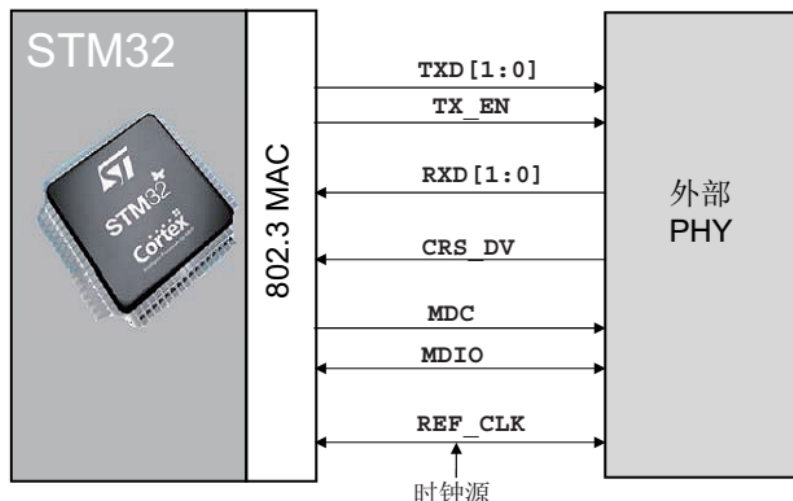


图 1.1.8 STM32F407 通过 RMII 与 PHY 芯片链接图

6) LWIP 简介

LWIP 是瑞典计算机科学院(SICS)的 Adam Dunkels 等开发的一个小型开源的 TCP/IP 协议栈。LWIP 是轻量级 IP 协议，有无操作系统的支持都可以运行，LWIP 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，它只需十几 KB 的 RAM 和 40K 左右的 ROM 就可以运行，这使 LWIP 协议栈适合在低端的嵌入式系统中使用。目前 LWIP 的最新版本是 1.4.1。本教程采用的就是 1.4.1 版本的 LWIP。关于 LWIP 的详细信息大家可以去 <http://savannah.nongnu.org/projects/lwip/> 这个网站去查阅，LWIP 的主要特性如下：

- ARP 协议，以太网地址解析协议；
- IP 协议，包括 IPv4 和 IPv6，支持 IP 分片与重装，支持多网络接口下数据转发；
- ICMP 协议，用于网络调试与维护；
- IGMP 协议，用于网络组管理，可以实现多播数据的接收；
- UDP 协议，用户数据报协议；
- TCP 协议，支持 TCP 拥塞控制，RTT 估计，快速恢复与重传等；
- 提供三种用户编程接口方式：raw/callback API、sequential API、BSD-style socket API；
- DNS，域名解析；
- SNMP，简单网络管理协议；
- DHCP，动态主机配置协议；
- AUTOIP，IP 地址自动配置；
- PPP，点对点协议，支持 PPPoE

我们从 LWIP 官网下载 LWIP1.4.1 版本，打开后如图 1.1.9 所示。

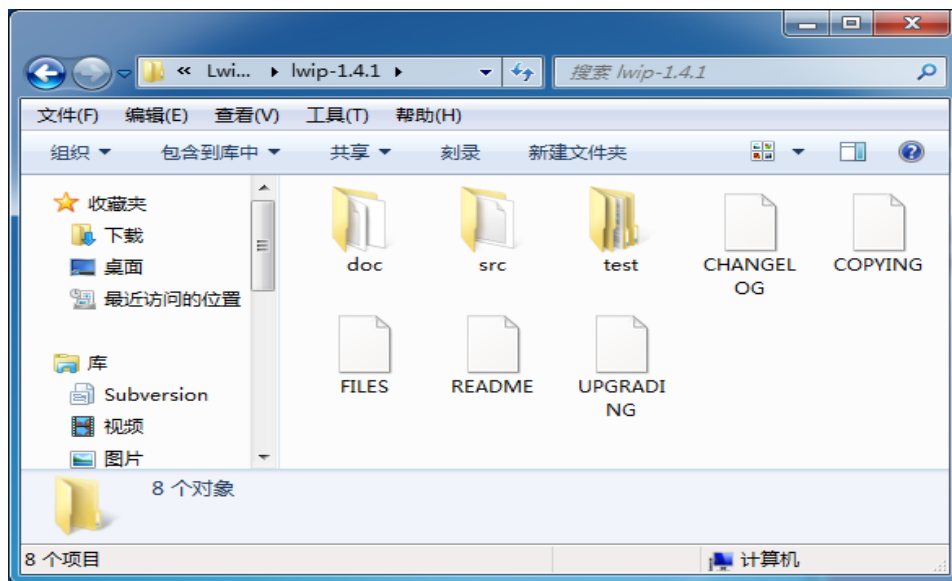


图 1.1.9 LWIP1.4.1 源码内容

打开从官网下载下来的 LWIP1.4.1 其中包括 doc,src 和 test 三个文件夹和 5 个其他文件。doc 文件夹下包含了几个与协议栈使用相关的文本文档, doc 文件夹里面有两个比较重要的文档:rawapi.txt 和 sys_arch.txt.rawapi.txt 告诉读者怎么使用 raw/callback API 进行编程,sys_arch.txt 包含了移植说明,在移植的时候会用到。src 文件夹是我们的重点,里面包含了 LWIP 的源码。test 是 LWIP 提供的一些测试程序,这里用不到。打开 src 源码文件夹,如图 1.1.10 所示。

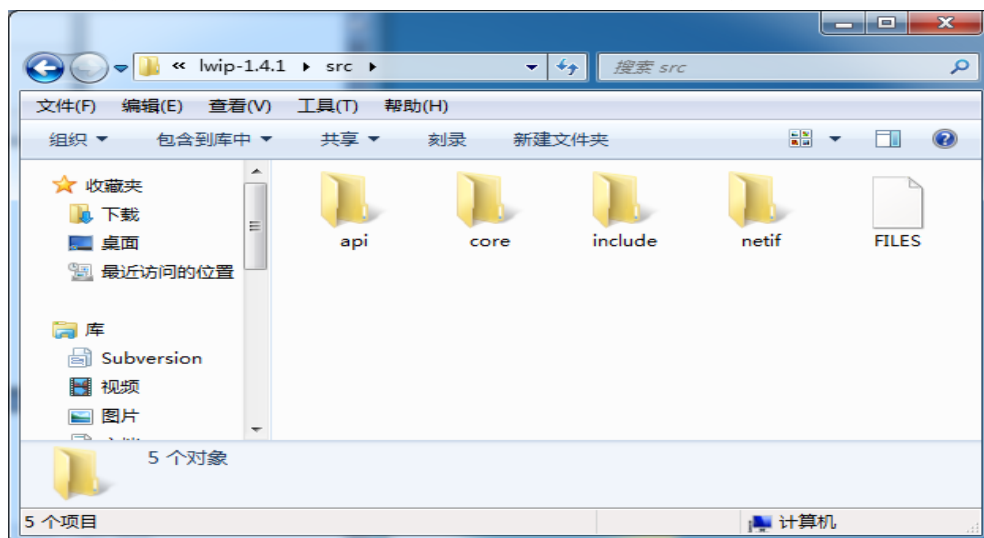


图 1.1.10 源码 src 文件夹

src 文件夹由 4 个文件夹组成: api、core、include、netif 四个文件夹。api 文件夹里面是 LWIP 的 sequential API(Netconn)和 socket API 两种接口函数的源码,要使用这两种 API 需要操作系统支持。core 文件夹是 LWIP 内核源码,include 文件夹里面是 LWIP 使用到的头文件,netif 文件夹里面是与网络底层接口有关的文件。

1.2 硬件设计

本章实验功能简介:编写 LAN8720 驱动程序,移植 LWIP。移植成功后开机初始化 LAN8720, LAN8720 通过自协商确定自身的工作速度以及双工模式,通过串口打印 MAC 地址、IP 地址、子网掩码和默认网关等信息。同时 DS0 提示程序正在运行。

- 1) 指示灯 DS0
- 2) TFTLCD 模块
- 3) LAN8720 模块

RMII REF CLK	PA1	35	PA1/U2_RTS/U4_RX/ETH_RMII_REF_CLK/ETH_MII_RX_CLK
ETH MDIO	USART2_TX	PA2	PA2/ETH_MDIO
RMII CRS DV	PA7	43	PA7/ETH_RMII_CRD_V
ETH MDC	PC1	27	PC1/ETH_MDC/ADC123_IN11
RMII RXD0	PC4	44	PC4/ETH_MII_RXD0/ETH_RMII_RXD0/ADC12_IN14
RMII RXD1	PC5	45	PC5/ETH_MII_RXD1/ETH_RMII_RXD1/ADC12_IN15
ETH RESET	PD3	117	PD3/FSMC_CLK/U2_CTS
PG11/FSMC_NCE4_2/ETH_MII_TX_EN/ETH_RMII_TX_EN	126	PG11	RMII_TX_EN
PG13/FSMC_A24/U6_CTS/ETH_MII_TXD0/ETH_RMII_TXD0	128	PG13	RMII_TXD0
PG14/FSMC_A25/U6_TX/ETH_MII_TXD1/ETH_RMII_TXD1	129	PG14	RMII_TXD1

因为 LAN8720 只有 RMII 接口,因此这里与开发板的连接采用了 RMII 接口。LAN8720 与开发板之间的引脚连接如表 1.2.1 所示。

LAN8720 引脚	STM32F407 引脚	LAN8720 引脚	STM32F407 引脚
ETH_MDIO	PA2	RMII_RXD0	PC4
ETH_MDC	PC1	RMII_RXD1	PC5
RMII_TXD0	PG13	RMII CRS_DV	PA7
RMII_TXD1	PG14	RMII_REFCLK	PA1
RMII_TX_EN	PG11	ETH_RESET	PD3

表 1.2.1 LAN8720 与开发板引脚连接情况

1.3 无操作系统 LWIP 移植

在本节教程和后续所有有关 LWIP 的教程中需要使用到动态内存管理实验的知识，因此请读者务必了解动态内存管理实验，此教程只针对 LWIP，不会对其他知识做讲解。

1.3.1 移植准备工作

1) 基础工程

在移植之前我们需要一个基础工程，因为我们要用到内存管理，因此这里我们使用实验 37 内存管理实验作为基础工程，我们在这个工程的基础上完成本章的移植过程。

本章我们要使用到 USMART 组件，因此我们要在内存管理实验的工程上添加 USMART 组件，如果已经添加了的话就不用再添加了，关于 USMART 组件的使用请参考我们的：STM32F4 开发指南实验 14 USMART 实验。

2) LWIP 文件下载

在移植的过程中我们需要两个文件，LWIP 源码和 LWIP 官方的例程，我们可以在 LWIP 官网：<http://download.savannah.gnu.org/releases/lwip/> 下载这两个文件，如图 1.3.1.1 所示为我们需要下载的两个文件。其中 lwip-1.4.1.zip 为 LWIP 的官方源码，contrib-1.4.1.zip 包含官方的一些例程和我们移植时所需要的一些头文件。

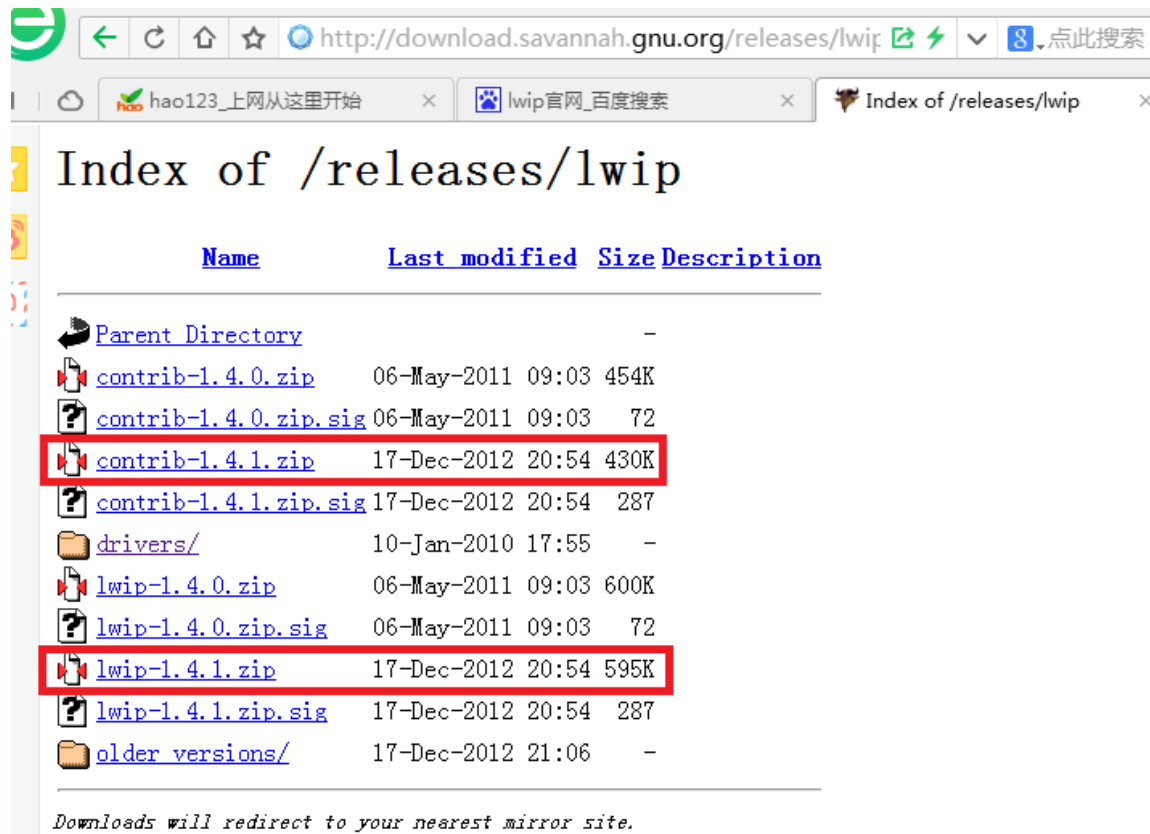


图 1.3.1 .LWIP 源文件

我们已经下载好了这两个文件放在光盘中，在软件资料文件夹下的 LWIP 学习资料文件夹中。目前最新的版本是 1.4.1，本教程所有的实验均使用 1.4.1 版本的 LWIP。

3) ST 以太网库

因为 STM32F407 带有以太网 MAC 模块，因此 ST 也提供了以太网库，我们可以在 ST 官网：<http://www.st.com/web/catalog/tools/FM147/CL1794/SC961/SS1743/LN1734/PF257906#> 下载，下载界面如图 1.3.1.2 所示。下载解压后为名为：STM32F4x7_ETH_LwIP_V1.1.0，我们已经将解压后的以太网库放到光盘中，在软件资料文件夹下的 LWIP 学习资料文件夹中。

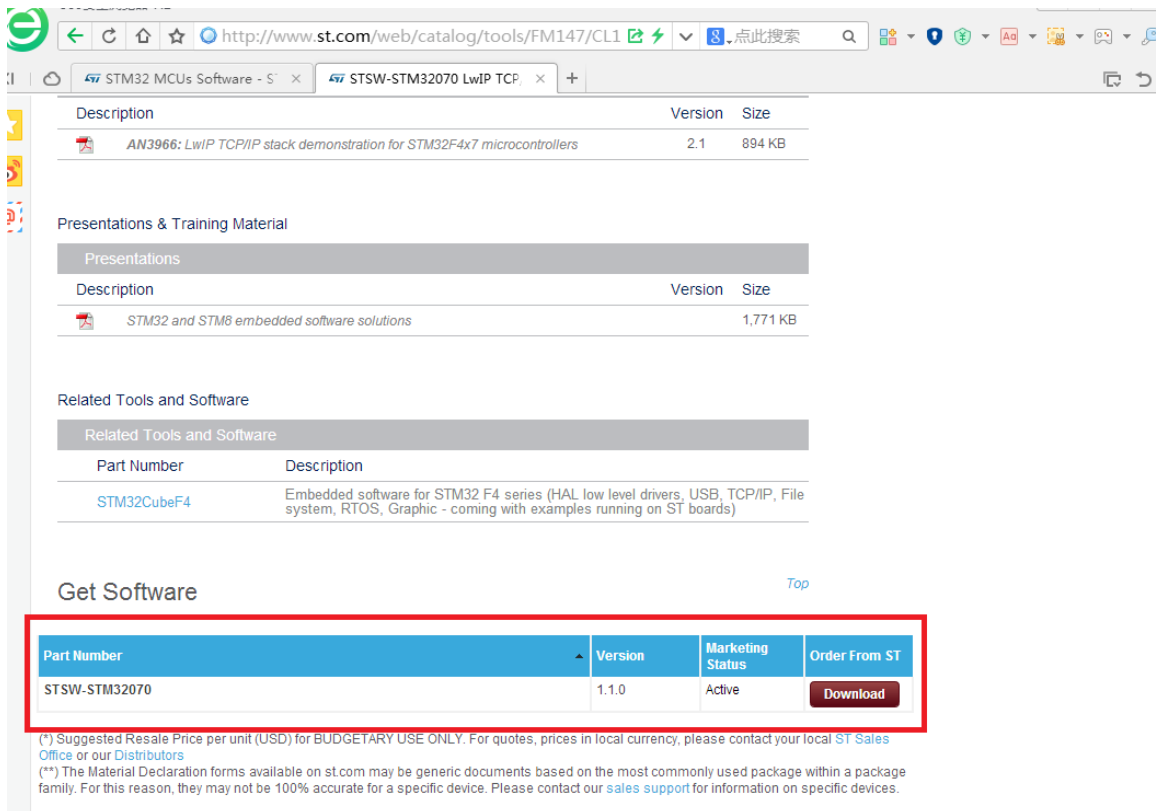


图 1.3.1.2 ST 官方以太网库下载

1.3.2 添加及修改 ST 以太网库

1) 添加以太网库

在上面一小节中我们已经准备好了移植所需要的文件，从这一节我们就开始 LWIP 的移植。我们首先将 ST 的以太网库添加到工程中，将 STM32F4x7_ETH_LwIP_V1.1.0 文件中 Libraries 文件夹下的 STM32F4x7_ETH_Driver 文件复制到我们基础工程的 FWLIB 文件夹下，如图 1.3.2.1 所示。



图 1.3.2.1 添加 ST 以太网库

在 STM32F4x7_ETH_Driver 文件夹中一共有 3 个文件，stm32f4x7_eth.h、stm32f4x7_eth.c 和 stm32f4x7_eth_conf_template.h。stm32f4x7_eth.h 为头文件，这个很好理解，stm32f4x7_eth.c 为 ST 的以太网库，里面有很多关于 STM32F4X7 的以太网的函数，stm32f4x7_eth_conf_template.h 里面定义了一些关于操作 PHY 芯片的信息，为了方便移植我们将 stm32f4x7_eth_conf_template.h 重命名为 stm32f4x7_eth_conf.h。最后我们将 stm32f4x7_eth.c 添加到我们的工程中，并且添加头文件路径，添加完成后的基础工程如图 1.3.2.2 所示。

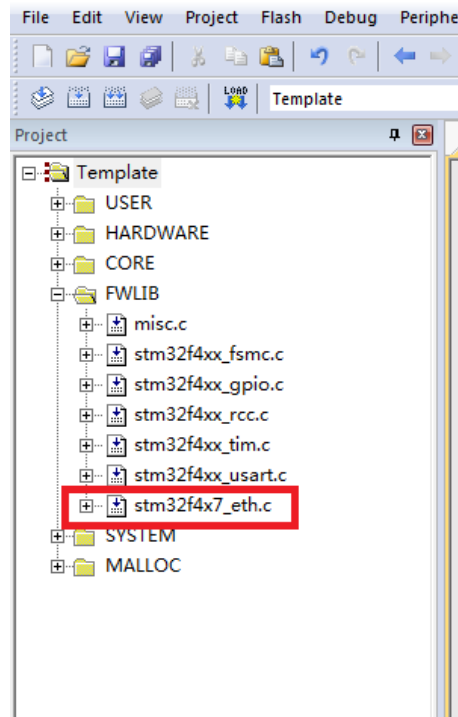


图 1.3.2.2 添加完 ST 以太网库后的工程

2) 修改 stm32f4x7_eth_conf.h 文件

ST 提供给我们的 stm32f4x7_eth_conf.h 文件只是一个参考文件，在这个文件是以 DP83848 为例编写的，而我们的 STM32F407 开发板使用的是 LAN8720，因此我们要根据 LAN8720 做一定的修改，针对 LAN8720 修改后的 stm32f4x7_eth_conf.h 文件代码如下。

```
#ifndef __STM32F4x7_ETH_CONF_H
#define __STM32F4x7_ETH_CONF_H
#include "stm32f4xx.h"

#define USE_ENHANCED_DMA_DESCRIPTOR

//如果使用自己定义的延时函数的话就注销掉下面一行代码，否则使用
//默认的低精度延时函数
//#define USE_Delay //使用默认延时函数，因此注销掉
#ifdef USE_Delay
#include "main.h"
#define _eth_delay_ Delay //Delay 为用户自己提供的高精度延时函数
#else
#define _eth_delay_ ETH_Delay //默认的_eth_delay 功能函数延时精度差
#endif

#ifdef CUSTOM_DRIVER_BUFFERS_CONFIG
//重新定义以太网接收和发送缓冲区的大小和数量
#define ETH_RX_BUF_SIZE ETH_MAX_PACKET_SIZE //接收缓冲区的大小
```

```

#define ETH_TX_BUF_SIZE    ETH_MAX_PACKET_SIZE //发送缓冲区的大小
#define ETH_RXBUFNB        20                  //接收缓冲区数量
#define ETH_TXBUFNB        5                  //发送缓冲区数量
#endif

//*****PHY 配置块*****
#ifdef USE_Delay
#define PHY_RESET_DELAY    ((uint32_t)0x000000FF) //PHY 复位延时
#define PHY_CONFIG_DELAY   ((uint32_t)0x00000FFF) //PHY 配置延时
//向以太网寄存器写数据时的延时
#define ETH_REG_WRITE_DELAY ((uint32_t)0x00000001)
#else
#define PHY_RESET_DELAY    ((uint32_t)0x000FFFFF) //PHY 复位延时
#define PHY_CONFIG_DELAY   ((uint32_t)0x00FFFFFF) //PHY 配置延时
//向以太网寄存器写数据时的延时
#define ETH_REG_WRITE_DELAY ((uint32_t)0x0000FFFF)
#endif

//LAN8720 PHY 芯片的状态寄存器
#define PHY_SR              ((uint16_t)31)        //LAN8720 的 PHY 状态寄存器地址
#define PHY_SPEED_STATUS    ((uint16_t)0x0004)    //LAN8720 PHY 速度状态值掩码
#define PHY_DUPLEX_STATUS   ((uint16_t)0x00010)    //LAN8720 PHY 连接状态值掩码
#endif

```

在上面代码中我们定义了操作 PHY 芯片时需要用到的延时配置，也定义了 LAN8720 的状态寄存器地址和 LAN8720 中的速度值、连接状态的掩码。通过读取 LAN8720 的状态寄存器并且与这两个掩码做计算即可得到网络的连接速度和双工模式。到这里我们编译一下工程，没有任何错误，如果有错误的话根据错误类型检查和修改工程。

2) 修改 stm32f4x7_eth.c 文件

在 stm32f4x7_eth.c 文件中针对不同的平台定义了四个数组：Rx_Buff[]、Tx_Buff[]、DMARxDscrTab[]和 DMATxDscrTab[]，这四个数组占用了大量的 RAM。我们在这里将这四个变量屏蔽掉，如图 1.3.2.3 所示。在其他文件中会采用内存管理的方式为这 4 个数组分配内存，这样我们就可以使用外部 SRAM，减少对 STM32F407 内部 RAM 的使用。

```

60 //if defined (__CC_ARM) /*!< ARM Compiler */
61 //  _align(4)
62 //  ETH_DMADescTypeDef  DMARxDscrTab[ETH_RXBUFNB]; /* Ethernet Rx MA Descriptor */
63 //  _align(4)
64 //  ETH_DMADescTypeDef  DMATxDscrTab[ETH_TXBUFNB]; /* Ethernet Tx DMA Descriptor */
65 //  _align(4)
66 //  uint8_t Rx_Buff[ETH_RXBUFNB][ETH_RX_BUF_SIZE]; /* Ethernet Receive Buffer */
67 //  _align(4)
68 //  uint8_t Tx_Buff[ETH_TXBUFNB][ETH_TX_BUF_SIZE]; /* Ethernet Transmit Buffer */
69
70 //elif defined (__ICCARM__) /*!< IAR Compiler */
71 //  #pragma data_alignment=4
72 //  ETH_DMADescTypeDef  DMARxDscrTab[ETH_RXBUFNB]; /* Ethernet Rx MA Descriptor */
73 //  #pragma data_alignment=4
74 //  ETH_DMADescTypeDef  DMATxDscrTab[ETH_TXBUFNB]; /* Ethernet Tx DMA Descriptor */
75 //  #pragma data_alignment=4
76 //  uint8_t Rx_Buff[ETH_RXBUFNB][ETH_RX_BUF_SIZE]; /* Ethernet Receive Buffer */
77 //  #pragma data_alignment=4
78 //  uint8_t Tx_Buff[ETH_TXBUFNB][ETH_TX_BUF_SIZE]; /* Ethernet Transmit Buffer */
79
80 //elif defined (__GNUC__) /*!< GNU Compiler */
81 //  ETH_DMADescTypeDef  DMARxDscrTab[ETH_RXBUFNB] __attribute__((aligned(4))); /* Ethernet Rx DMA Descriptor */
82 //  ETH_DMADescTypeDef  DMATxDscrTab[ETH_TXBUFNB] __attribute__((aligned(4))); /* Ethernet Tx DMA Descriptor */
83 //  uint8_t Rx_Buff[ETH_RXBUFNB][ETH_RX_BUF_SIZE] __attribute__((aligned(4))); /* Ethernet Receive Buffer */
84 //  uint8_t Tx_Buff[ETH_TXBUFNB][ETH_TX_BUF_SIZE] __attribute__((aligned(4))); /* Ethernet Transmit Buffer */
85
86 //elif defined (__TASKING__) /*!< TASKING Compiler */
87 //  _align(4)
88 //  ETH_DMADescTypeDef  DMARxDscrTab[ETH_RXBUFNB]; /* Ethernet Rx MA Descriptor */
89 //  _align(4)
90 //  ETH_DMADescTypeDef  DMATxDscrTab[ETH_TXBUFNB]; /* Ethernet Tx DMA Descriptor */
91 //  _align(4)
92 //  uint8_t Rx_Buff[ETH_RXBUFNB][ETH_RX_BUF_SIZE]; /* Ethernet Receive Buffer */
93 //  _align(4)
94 //  uint8_t Tx_Buff[ETH_TXBUFNB][ETH_TX_BUF_SIZE]; /* Ethernet Transmit Buffer */
95
96 //endif /* __CC_ARM */
97

```

图 1.3.2.3 屏蔽掉四个大数组

1.3.3 添加网卡驱动程序

1) STM32F4 以太网 DMA 描述符

在添加网卡驱动之前我们有必要了解一下 STM32F4x7 的 DMA 描述符，STM32F407 以太网模块中的接收/发送 FIFO 和内存之间的以太网数据包传输是以太网 DMA 使用 DMA 描述符完成的。一共有两个描述符列表：一个用于接收，一个用于发送，两个列表的基址分别写入 ETH_DMARDLAR 寄存器和 ETH_DMA_TDLAR 寄存器中。

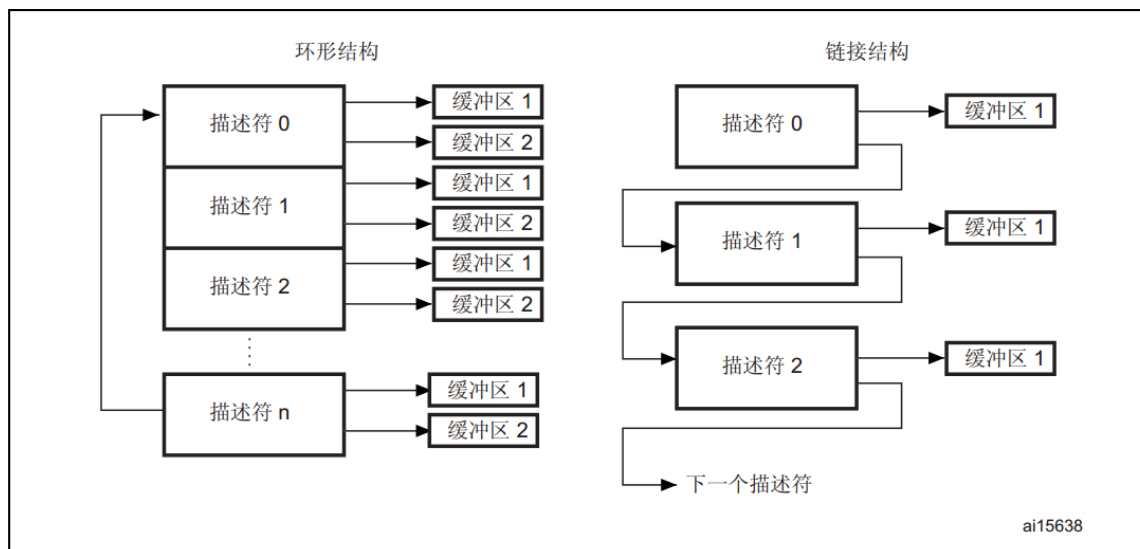


图 1.3.3.1 DMA 描述符列表两种结构

图 1.3.3.1 展示了 DMA 描述符的两种结构：环形结构和链接结构，在 ST 提供的以太网驱动程序 stm32f4x7_eth.c 中使用的是链 接结构，DMA 描述符连接结构的具体描述如图 1.3.7 所示。

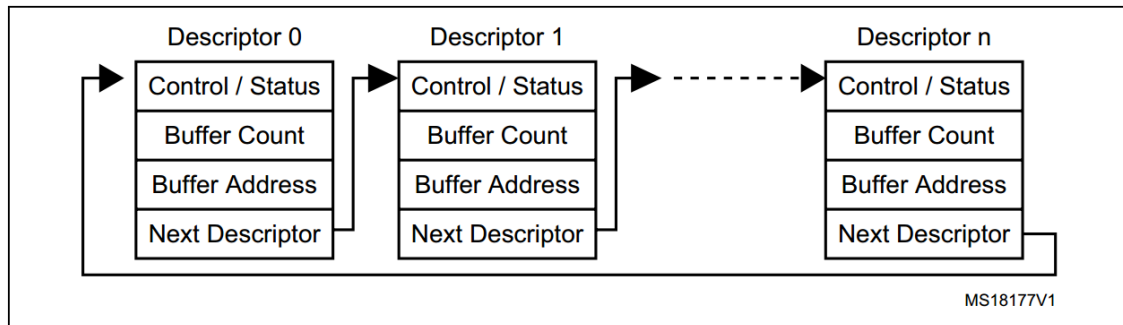


图 1.3.3.2 DMA 描述符链接结构

图 1.3.3.2 中就是 stm32f4x7_eth.c 采用的 DMA 链接结构，应注意以下几点。

- 1、一个以太网数据包可以跨越一个或多个 DMA 描述符。
- 2、一个 DMA 描述符只能用于一个以太网数据包
- 3、DMA 描述符列表中的最后一个描述符指向第一个，形成链式结构。

在 ST 的以太网驱动库 stm32f4x7_eth.h 中有个结构体 ETH_DMADESCTypeDef，这个结构体定义了 DMA 描述符，ETH_DMADESCTypeDef 结构体代码如下。

```
typedef struct {
    __IO uint32_t    Status;           //状态
    uint32_t    ControlBufferSize;     //控制和 buffer1, buffer2 的长度
    uint32_t    Buffer1Addr;           //buffer1 地址
    uint32_t    Buffer2NextDescAddr;    //buffer2 地址或下一个描述符地址
//增强的以太网 DMA 描述符
#ifdef USE_ENHANCED_DMA_DESCRIPTOR
    uint32_t    ExtendedStatus;        //增强描述符状态
    uint32_t    Reserved1;             //保留
    uint32_t    TimeStampLow;          //时间戳低位
    uint32_t    TimeStampHigh;         //时间戳高位
#endif
} ETH_DMADESCTypeDef;
```

DMA 描述符分为增强描述符和常规描述符，本教程中我们使用的是常规描述符，如果使用常规描述符的话就只使用 ETH_DMADESCTypeDef 结构体的前四个成员变量，有关常规描述符和增强描述符的区别大家可以参考 STM32F4xx 中文参考手册的 865 页和 869 页。

我们知道 STM32F407 的描述符分为发送描述符和接收描述符，发送描述符和接收描述符都用 ETH_DMADESCTypeDef 这个结构体来定义，我们这里只以发送描述符(常规 Tx DMA 描述符)为例讲解一下，接收描述符(常规 Rx DMA 描述符)与其类似。常规 Tx DMA 描述符可以用图 1.3.3.3 来表示。

31

0

TDES 0	OWN	Ctrl [30:26]	T T S E	保留 24	Ctrl [23:20]	保留 [19:18]	T T S S	状态 [16:0]	
TDES 1	保留 [31:29]		缓冲区 2 字节计数 [28:16]				保留 [15:13]	缓冲区 1 字节计数 [12:0]	
TDES 2	缓冲区 1 地址 [31:0]/时间戳低电平 [31:0]								
TDES 3	缓冲区 2 地址 [31:0] 或下一个描述符地址 [31:0]/时间戳高电平 [31:0]								

图 1.3.3.3 常规 Tx DMA 描述符图解

我们可以从图 1.3.3.3 中看到好像常规 Tx DMA 描述符有 4 个“寄存器”：TDES0、TDES1、TDES2 和 TDES3，如果是增强描述符的话就会有 8 个“寄存器”。这里一定要注意这 4 个“寄存器”并不是 STM32F407 真实存在的，你在 STM32F407 的寄存器列表里面是找不到的，这 4 个“寄存器”是由 ETH_DMADescTypeDef 这个结构体描述的，我将它们称之为“软件寄存器”，这些“寄存器”也叫做描述符字，描述符字和 ETH_DMADescTypeDef 这个结构体的关系如表 1.3.3.1 所示(此处以 Tx DMA 描述符为例)

描述符字	ETH_DMADescTypeDef 结构体成员	备注
TDES0	Status	常规/增强 Tx DMA 描述符共有
TDES1	ControlBufferSize	常规/增强 Tx DMA 描述符共有
TDES2	Buffer1Addr	常规/增强 Tx DMA 描述符共有
TDES3	Buffer2NextDescAddr	常规/增强 Tx DMA 描述符共有
TDES4	ExtendedStatus	增强 Tx DMA 描述符所特有的
TDES5	Reserved1	增强 Tx DMA 描述符所特有的
TDES6	TimeStampLow	增强 Tx DMA 描述符所特有的
TDES7	TimeStampHigh	增强 Tx DMA 描述符所特有的

表 1.3.3.1 描述符字和 ETH_DMADescTypeDef 成员之间的对应关系

以上表 1.3.3.1 所列的是常规 Tx DMA 描述符，常规 Rx DMA 描述符与之类似。同我们操作寄存器一样，描述符字的每个位代表不同的状态或控制，关于 Tx DMA 描述符中描述符字各个位的含意大家可以参考 STM32F4xx 中文参考手册的 865 页(Rx DMA 描述符在 873 页)。

我们前面说过 ST 的官方以太网库 stm32f4x7_eth.c 中使用链接结构的 DMA 描述符，那么在以太网描述符结构体 ETH_DMADescTypeDef 中 Buffer1Addr 就是缓冲区的地址，Buffer2NextDescAddr 就是下一个描述符的地址，如图 1.3.3.4 所示那样。

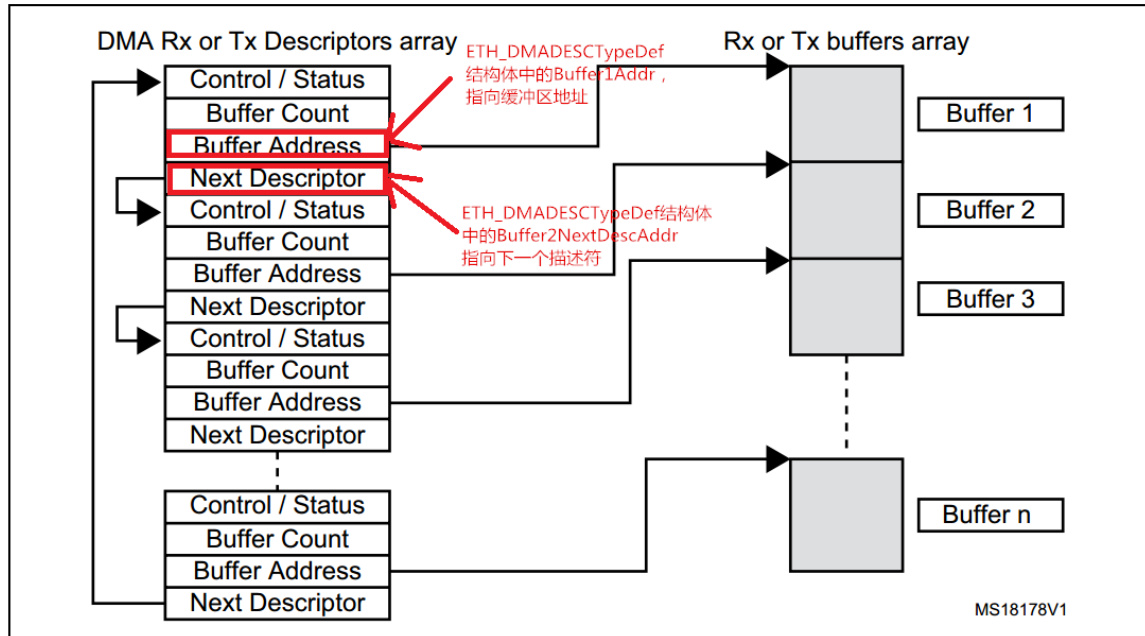


图 1.3.3.4ST 以太网驱动库中描述符和缓冲区关系

正如图 1.3.3.4 所示那样，在 `stm32f4x7_eth.c` 文件中定义了两个 DMA 描述符数组，一个用于 DMA 接收，一个用于 DAM 发送，代码如下。

```
ETH_DMADESCTypeDef DMARxDscrTab[ETH_RXBUFNB];
ETH_DMADESCTypeDef DMATxDscrTab[ETH_TXBUFNB];
```

这两个数组是不是看起来有点眼熟，没错！这两个数组就是我们前面屏蔽掉并且采用内存分配函数给其分配内存的 4 个数组中的其中两个。我们前面说过是由于它们占用 RAM 过大，所有才采用内存分配函数为他们分配内存。

我们说过 ST 以太网驱动库 `stm32f4x7_eth.c` 中我们采用的是 DMA 链接结构，但是 `DMARxDscrTab` 和 `DMATxDscrTab` 是两个数组，那么我们必然要将这两个数组改为链接结构。在 `stm32f4x7_eth.c` 文件中有两个函数 `ETH_DMARxDscrChainInit()` 和 `ETH_DMATxDscrChainInit()`，通过这两个函数我们就可以将上面两个数组变为链接结构，这两个函数会在我们的其他文件中调用。

我们知道如果采用链接结构的话 `ETH_DMADESCTypeDef` 结构体中的 `Buffer1Addr` 成员变量存放缓冲区地址，在 `stm32f4x7_eth.c` 中定义了两个数组用做发送和接收的缓冲区，代码如下。

```
uint8_t Rx_Buff[ETH_RXBUFNB][ETH_RX_BUF_SIZE];
uint8_t Tx_Buff[ETH_TXBUFNB][ETH_TX_BUF_SIZE];
```

这两个数组也是被我们屏蔽掉采用内存分配函数给其分配内存的 4 个数组中的另外两个，通过上面的讲解大家也应该知道了为什么这两个数组占用 RAM 过大了，因为他们被用做数据缓冲区，那么其所占 RAM 必然很大，大家也可以实际测量一下这两个数组的大小。在上面这四个数组中使用了四个宏定义：`ETH_RXBUFNB`、`ETH_TXBUFNB`、`ETH_RX_BUF_SIZE` 和 `ETH_TX_BUF_SIZE`，这四个宏在 `stm32f4x7_eth.h` 中定义了，他们的具体意义如下(以下并不是代码，是我们写的注释！)。

```
ETH_RXBUFNB           //接收缓冲区数量默认为 5 个
ETH_TXBUFNB           //发送缓冲区数量默认为 5 个
ETH_RX_BUF_SIZE       //接收缓冲区大小默认 1524 字节
ETH_TX_BUF_SIZE       //发送缓冲区大小默认 1524 字节
```

在 stm32f4x7_eth.c 文件中为了追踪 Rx/Tx DMA 描述符还定义了两个非常重要的全局指针变量，如下代码。

```
_IO ETH_DMADescTypeDef *DMATxDescToSet;
_IO ETH_DMADescTypeDef *DMARxDescToGet;
```

这两个指针变量指向 ETH_DMADescTypeDef 结构体，在使用中他们两个分别指向下一个要发送或者接受的数据包，如图 1.3.3.5 所示那样，至此 STM32F407 以太网模块中的 DMA 描述符讲解就晚了，关于 DMA 描述符的详细内容大家可以参考 STM32F4xx 中文参考手册。

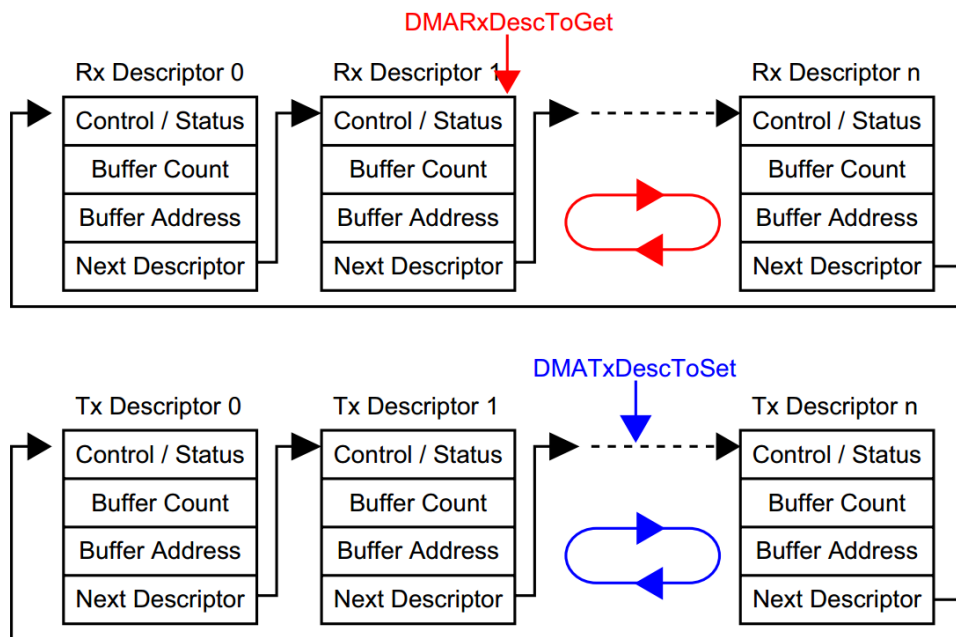


图 1.3.3.5 DMATxDescToSet 和 DMARxDescToGet 指针的使用

2) 添加 LAN8720 和 MAC /DMA 驱动

打开我们的 LWIP 无操作系统移植实验的 HARDWARE 文件夹，在里面有一个 ETHERNET 文件，在这个文件中有 lan8720.c 和 lan8720.h 这两个文件，这两个文件里面包含 LAN8720 和 STM32F407 自带的 MAC 的驱动程序，大家在移植的时候将 ETHERNET 文件拷贝到自己的 HARDWARE 文件中，并且将 lan8720.c 添加到工程中。在这里我们还要使用到 stm32f4xx_syscfg.c 文件，这个文件是官方的标准外设库文件，我们将其加入到工程中。添加完 lan8720.c 和 stm32f4xx_syscfg.c 文件后的工程如图 1.3.3.6 所示。

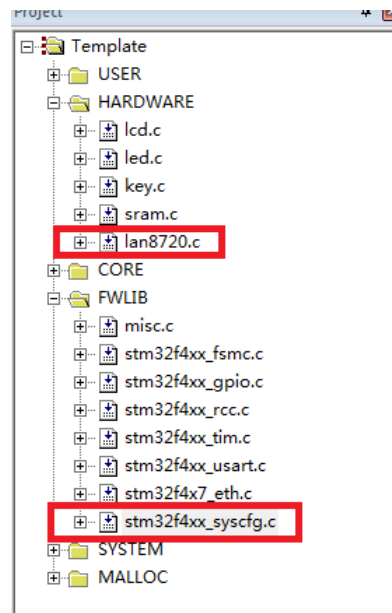


图 1.3.3.6 添加 lan8720.c 和 stm32f4xx_syscfg.c 后的工程

在 lan8720.c 中有 10 个函数，如表 1.3.3.2 所示。

函数	说明
LAN8720_Init()	LAN8720 初始化函数
ETHERNET_NVICConfiguration()	以太网 DMA 中断优先级配置
LAN8720_Get_Speed()	获取当前连接速度和双工状态
ETH_MACDMA_Config()	以太网 MAC 和 DMA 配置函数
ETH_IRQHandler()	以太网 DMA 接收中断服务函数
ETH_Rx_Packet()	从网卡中接收数据包
ETH_Tx_Packet()	从网卡发送数据包
ETH_GetCurrentTxBuffer()	得到当前描述符的发送缓冲区地址
ETH_Mem_Malloc()	Rx_Buff[], Tx_Buff[]、DMARxDscrTab[]和 DMATxDscrTab[] 这四个数组申请内存
ETH_Mem_Free()	释 放 Rx_Buff[] 、 Tx_Buff[] 、 DMARxDscrTab[] 和 DMATxDscrTab[]这四个数组的内存

表 1.3.3.2 lan8720.c 文件函数

接下来我们详细的介绍一下表 1.3.3.2 中的这几个函数，首先来看一下 LAN8720_Init()函数。LAN8720_Init()函数代码如下。

```
u8 LAN8720_Init(void)
{
    u8 rval=0;
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOC|\
    RCC_AHB1Periph_GPIOG , ENABLE); //使能 GPIO 时钟 RMII 接口
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE); //使能 SYSCFG 时钟

    //MAC 和 PHY 之间使用 RMII 接口
```

```

SYSCFG_ETH_MediaInterfaceConfig(SYSCFG_ETH_MediaInterface_RMII);
//配置 PA1 PA2 PA7
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_7;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
GPIO_Init(GPIOA, &GPIO_InitStructure);

GPIO_PinAFConfig(GPIOA, GPIO_PinSource1, GPIO_AF_ETH); //引脚复用到网络接口上
GPIO_PinAFConfig(GPIOA, GPIO_PinSource2, GPIO_AF_ETH);
GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_ETH);

//配置 PC1,PC4 and PC5
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_4 | GPIO_Pin_5;
GPIO_Init(GPIOC, &GPIO_InitStructure);
GPIO_PinAFConfig(GPIOC, GPIO_PinSource1, GPIO_AF_ETH); //引脚复用到网络接口上
GPIO_PinAFConfig(GPIOC, GPIO_PinSource4, GPIO_AF_ETH);
GPIO_PinAFConfig(GPIOC, GPIO_PinSource5, GPIO_AF_ETH);

//配置 PG11, PG14 and PG13
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_13 | GPIO_Pin_14;
GPIO_Init(GPIOG, &GPIO_InitStructure);
GPIO_PinAFConfig(GPIOG, GPIO_PinSource11, GPIO_AF_ETH);
GPIO_PinAFConfig(GPIOG, GPIO_PinSource13, GPIO_AF_ETH);
GPIO_PinAFConfig(GPIOG, GPIO_PinSource14, GPIO_AF_ETH);

//配置 PD3 为推挽输出
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
GPIO_Init(GPIOD, &GPIO_InitStructure);

LAN8720_RST=0; //硬件复位 LAN8720
delay_ms(50);
LAN8720_RST=1; //复位结束
ETHERNET_NVICConfiguration(); //设置中断优先级
rval=ETH_MACDMA_Config(); //配置 MAC 及 DMA
return !rval; //ETH 的规则为:0,失败;1,成功;所以要取反一下
}

```

在这个函数中首先是开启相应的时钟，然后调用 SYSCFG_ETH_MediaInterfaceConfig 函数

来指定接口为 RMII，指定 RMII 接口一定要在最前面进行。接着完成 RMII 接口所需要的 IO 口的配置。配置完成后复位 LAN8720。在这里我们使用 DMA 中断方式来接收数据，也可采用轮询方式接收，但是轮询方式太过占用 CPU，在以后的例程中均使用中断接收方式。复位完以后调用 ETHERNET_NVICConfiguration()和 ETH_MACDMA_Config()函数来完成 DMA 中断优先级、MAC 以及 DMA 的配置，在这里我们将 DMA 中断优先级设置为组 2 中最高优先级（这点非常重要）！

ETHERNET_NVICConfiguration 函数很简单，就是配置以太网 DMA 中断优先级，大家可以自行看一下源代码。

LAN8720_Get_Speed()函数是获取网络的连接速度和双工状态的，通过读取 LAN8720 内部寄存器得到，函数代码如下。

```
u8 LAN8720_Get_Speed(void)
```

```
{
    u8 speed;
    //从 LAN8720 的 31 号寄存器中读取网络速度和双工模式
    speed=((ETH_ReadPHYRegister(0x00,31)&0x1C)>>2);
    return speed;
}
```

接下来介绍 ETH_MACDMA_Config 函数，此函数用来配置以太网的 MAC、DMA。

```
//初始化 ETH MAC 层及 DMA 配置
```

```
//返回值:ETH_ERROR,发送失败(0)
```

```
//      ETH_SUCCESS,发送成功(1)
```

```
u8 ETH_MACDMA_Config(void)
```

```
{
    u8 rval;
    ETH_InitTypeDef ETH_InitStructure;

    //使能以太网 MAC 以及 MAC 接收和发送时钟
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_ETH_MAC\|
    RCC_AHB1Periph_ETH_MAC_Tx |RCC_AHB1Periph_ETH_MAC_Rx, ENABLE);

    ETH_DeInit();                //AHB 总线重启以太网
    ETH_SoftwareReset();          //软件重启网络
    while (ETH_GetSoftwareResetStatus() == SET); //等待软件重启网络完成
    ETH_StructInit(&ETH_InitStructure); //初始化网络为默认值

    //开启网络自适应功能
    ETH_InitStructure.ETH_AutoNegotiation = ETH_AutoNegotiation_Enable;
    ETH_InitStructure.ETH_LoopbackMode = ETH_LoopbackMode_Disable; //关闭反馈
    //关闭重传功能
    ETH_InitStructure.ETH_RetryTransmission = ETH_RetryTransmission_Disable;
    //关闭自动去除 PDA/CRC 功能
    ETH_InitStructure.ETH_AutomaticPadCRCStrip = ETH_AutomaticPadCRCStrip_Disable;
    //关闭接收所有的帧
```



```

ETH_InitStructure.ETH_ReceiveAll = ETH_ReceiveAll_Disable;
//允许接收所有广播帧
ETH_InitStructure.ETH_BroadcastFramesReception = ETH_BroadcastFramesReception_Enable;
//关闭混合模式的地址过滤
ETH_InitStructure.ETH_PromiscuousMode = ETH_PromiscuousMode_Disable;
//对于组播地址使用完美地址过滤
ETH_InitStructure.ETH_MulticastFramesFilter=ETH_MulticastFramesFilter_Perfect;
//对单播地址使用完美地址过滤
ETH_InitStructure.ETH_UnicastFramesFilter = ETH_UnicastFramesFilter_Perfect;
#ifdef CHECKSUM_BY_HARDWARE
//开启 ipv4 和 TCP/UDP/ICMP 的帧校验和卸载
ETH_InitStructure.ETH_ChecksumOffload = ETH_ChecksumOffload_Enable;
#endif

//当我们使用帧校验和卸载功能的时候，一定要使能存储转发模式,存储转发模式中要保证
//整个帧存储在 FIFO 中,这样 MAC 能插入/识别出帧校验值,当帧校验正确的时候 DMA 就
//可以处理帧,否则就丢弃掉该帧

//开启丢弃 TCP/IP 错误帧
ETH_InitStructure.ETH_DropTCPChecksumErrorFrame=\
ETH_DropTCPChecksumErrorFrame_Enable;
//开启接收数据的存储转发模式
ETH_InitStructure.ETH_ReceiveStoreForward=ETH_ReceiveStoreForward_Enable;
//开启发送数据的存储转发模式
ETH_InitStructure.ETH_TransmitStoreForward = ETH_TransmitStoreForward_Enable;
//禁止转发错误帧
ETH_InitStructure.ETH_ForwardErrorFrames = ETH_ForwardErrorFrames_Disable;
//不转发过小的帧
ETH_InitStructure.ETH_ForwardUndersizedGoodFrames=\
ETH_ForwardUndersizedGoodFrames_Disable;
//打开处理第二帧功能
ETH_InitStructure.ETH_SecondFrameOperate = ETH_SecondFrameOperate_Enable;
//开启 DMA 传输的地址对齐功能
ETH_InitStructure.ETH_AddressAlignedBeats = ETH_AddressAlignedBeats_Enable;
//开启固定突发功能
ETH_InitStructure.ETH_FixedBurst = ETH_FixedBurst_Enable;
//DMA 接收的最大突发长度为 32 个节拍
ETH_InitStructure.ETH_RxDMABurstLength = ETH_RxDMABurstLength_32Beat;
//DMA 接收的最大突发长度为 32 个节拍
ETH_InitStructure.ETH_TxDMABurstLength = ETH_TxDMABurstLength_32Beat;
ETH_InitStructure.ETH_DMAArbitration = ETH_DMAArbitration_RoundRobin_RxTx_2_1;
rval=ETH_Init(&ETH_InitStructure,LAN8720_PHY_ADDRESS); //配置 ETH
if(rval==ETH_SUCCESS)//配置成功
{

```

```

        //使能以太网接收中断
        ETH_DMAITConfig(ETH_DMA_IT_NIS|ETH_DMA_IT_R,ENABLE);
    }
    return rval;
}

```

在 ETH_MACDMA_Config 函数中首先使能 MAC 以及 MAC 接收和发送时钟。然后我们配置 MAC 和 DMA，配置 MAC 以及 DMA 是调用 ETH_Init 函数来完成，但是 ETH_Init 中有很多的参数需要设置，因此通过调用 ETH_StructInit 函数来将网络配置为默认值，然后在根据我们的实际需要去配置，比如是否开启自适应功能、是否开启反馈功能等等，大家可以看一下我们的具体代码，后面都有详细的注释，也可以根据自己的实际情况做出相应的修改。在最后如果调用 ETH_Init()函数配置成功的话就开启了 DMA 总中断和 DMA 接收中断。

ETH_IRQHandler()函数为以太网 DMA 接收中断服务函数，中断服务函数代码如下：

```

//以太网中断服务函数
void ETH_IRQHandler(void)
{
    while(ETH_GetRxPktSize(DMARxDescToGet)!=0) //检测是否收到数据包
    {
        lwip_pkt_handle();
    }
    ETH_DMAClearITPendingBit(ETH_DMA_IT_R); //清除 DMA 接收中断标志位
    ETH_DMAClearITPendingBit(ETH_DMA_IT_NIS); //清除 DMA 中断标志位
}

```

在中断服务函数中，我们通过判断接收到的数据包长度是否为 0，当不为 0 的时候就调用 lwip_pkt_handle 函数处理接收到的数据包，这个函数我们稍后会做讲解。处理完成后清除 DMA 接收中断标志位和 DMA 总中断标志位。

ETH_Rx_Packet() 函数是从以太网中接收数据，并将接收到的数据打包成一个 FrameTypeDef 类型的结构体返回，FrameTypeDef 结构体在 stm32f4x7_eth.h 中有定义。前面讲过 STM32F407 的网络模块是将接收到的数据放到 DMA 描述符中的。所以在 FrameTypeDef 结构体中有一个指向 DMA 描述符的指针变量，ETH_Rx_Packet()函数代码如下。

```

FrameTypeDef ETH_Rx_Packet(void)
{
    u32 framelength=0;
    FrameTypeDef frame={0,0};
    //检查当前描述符,是否属于 ETHERNET DMA(设置的时候)/CPU(复位的时候)
    if((DMARxDescToGet->Status&ETH_DMARxDesc_OWN)!=(u32)RESET)
    {
        frame.length=ETH_ERROR;
        if ((ETH->DMASR&ETH_DMASR_RBUS)!=(u32)RESET)
        {
            ETH->DMASR = ETH_DMASR_RBUS;//清除 ETH DMA 的 RBUS 位
            ETH->DMARPDR=0;//恢复 DMA 接收
        }
        return frame;//错误,OWN 位被设置了
    }
}

```

```

    }
    if(((DMARxDescToGet->Status&ETH_DMARxDesc_ES)==(u32)RESET)&&
        ((DMARxDescToGet->Status & ETH_DMARxDesc_LS)!=(u32)RESET)&&
        ((DMARxDescToGet->Status & ETH_DMARxDesc_FS)!=(u32)RESET))
    {
        //得到接收包帧长度(不包含 4 字节 CRC)
        framelength=((DMARxDescToGet->Status&ETH_DMARxDesc_FL)>>)\
            ETH_DMARxDesc_FrameLengthShift)-4;
        frame.buffer = DMARxDescToGet->Buffer1Addr;//得到包数据所在的位置
    }else framelength=ETH_ERROR;//错误
    frame.length=framelength;
    frame.descriptor=DMARxDescToGet;
    //更新 ETH DMA 全局 Rx 描述符为下一个 Rx 描述符
    //为下一次 buffer 读取设置下一个 DMA Rx 描述符
    DMARxDescToGet=(ETH_DMADESCTypeDef*)\
        (DMARxDescToGet->Buffer2NextDescAddr);
    return frame;
}

```

ETH_Tx_Packet()函数是将当前发送描述符中的数据发送出去，函数代码如下。

```

u8 ETH_Tx_Packet(u16 FrameLength)
{
    //检查当前描述符,是否属于 ETHERNET DMA(设置的时候)/CPU(复位的时候)
    if((DMATxDescToSet->Status&ETH_DMATxDesc_OWN)!=(u32)RESET)return
        ETH_ERROR;//错误,OWN 位被设置了
    //设置帧长度,bits[12:0]
    DMATxDescToSet->ControlBufferSize=(FrameLength&ETH_DMATxDesc_TBS1);
    //设置最后一个和第一个位段置位(1 个描述符传输一帧)
    DMATxDescToSet->Status|=ETH_DMATxDesc_LS|ETH_DMATxDesc_FS;
    //设置 Tx 描述符的 OWN 位,buffer 重归 ETH DMA
    DMATxDescToSet->Status|=ETH_DMATxDesc_OWN;
    //当 Tx Buffer 不可用位(TBUS)被设置的时候,重置它.恢复传输
    if((ETH->DMASR&ETH_DMASR_TBUS)!=(u32)RESET)
    {
        ETH->DMASR=ETH_DMASR_TBUS;//重置 ETH DMA TBUS 位
        ETH->DMATPDR=0;//恢复 DMA 发送
    }
    //更新 ETH DMA 全局 Tx 描述符为下一个 Tx 描述符
    //为下一次 buffer 发送设置下一个 DMA Tx 描述符
    DMATxDescToSet=(ETH_DMADESCTypeDef*)(DMATxDescToSet->Buffer2NextDescAddr);
    return ETH_SUCCESS;
}

```

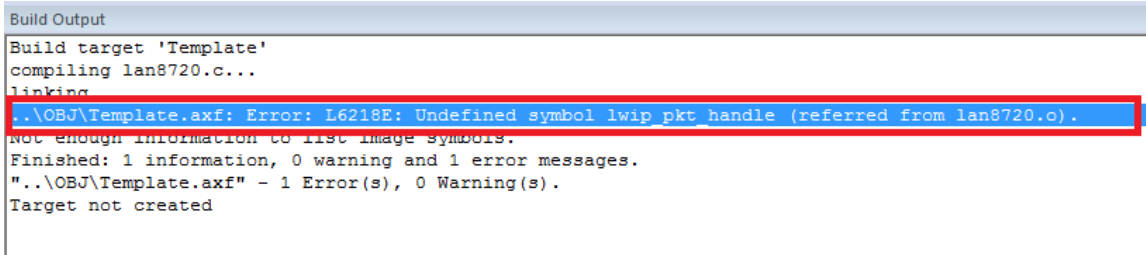
ETH_GetCurrentTxBuffer()函数为获取当前发送描述符的发送缓冲区地址，程序很简单，这里就不贴出代码了。

ETH_Mem_Malloc() 函数就是为我们前面提到的那四个数组：Rx_Buff[]、Tx_Buff[]、DMARxDscrTab[]和 DMATxDscrTab[]的内存分配，函数代码如下。

```
u8 ETH_Mem_Malloc(void)
{
    //申请内存
    DMARxDscrTab=mymalloc(SRAMIN,ETH_RXBUFNB*sizeof(ETH_DMADESCTypeDef));
    DMATxDscrTab=mymalloc(SRAMIN,ETH_TXBUFNB*sizeof(ETH_DMADESCTypeDef));
    Rx_Buff=mymalloc(SRAMIN,ETH_RX_BUF_SIZE*ETH_RXBUFNB); //申请内存
    Tx_Buff=mymalloc(SRAMIN,ETH_TX_BUF_SIZE*ETH_TXBUFNB); //申请内存
    if(!DMARxDscrTab||!DMATxDscrTab||!Rx_Buff||!Tx_Buff)
    {
        ETH_Mem_Free();
        return 1; //申请失败
    }
    return 0;      //申请成功
}
```

ETH_Mem_Free()函数为释放内存函数，功能是将 Rx_Buff[]、Tx_Buff[]、DMARxDscrTab[]和 DMATxDscrTab[]这四个数组的内存释放掉，这里就不贴出具体代码了。

至此网卡驱动添加完成，我们编译一下提示如图 1.3.3.7 所示错误，这是因为在以太网 DMA 接收中断服务函数 ETH_IRQHandler()中调用了 lwip_pkt_handle()函数，而这个函数没有在 lan8720.c 文件中定义，这个错误提示不用管，lwip_pkt_handle()函数后续会讲解。



```
Build Output
Build target 'Template'
compiling lan8720.c...
linking
..\OBJ\Template.axf: Error: L6218E: Undefined symbol lwip_pkt_handle (referred from lan8720.o).
not enough information to list image symbols.
Finished: 1 information, 0 warning and 1 error messages.
"..\OBJ\Template.axf" - 1 Error(s), 0 Warning(s).
Target not created
```

图 1.3.3.7 添加网卡驱动编译后的错误提示

1.3.4 LWIP 数据包和网络接口管理

以下关于 LWIP 数据包管理和网络接口管理部分参考自《嵌入式网络那些事 LWIP 协议深度剖析与实战演练》作者朱升林。

1) LWIP 数据包管理

LWIP 协议栈使用 pbuf 结构体来描述协议栈使用中的数据包，pbuf 结构体在 pbuf.h 中有定义，定义如下。

```
struct pbuf {
    struct pbuf *next;    //指向下一个 pbuf 结构体，可以构成链表
    void *payload;        //指向该 pbuf 真正的数据区
    u16_t tot_len;        //当前 pbuf 和链表中后面所有 pbuf 的数据长度，它们属于一个数据包
    u16_t len;            //当前 pbuf 的数据长度
    u8_t type;            //当前 pbuf 的类型
    u8_t flags;           //状态为，保留
```

```
u16_t ref;      //该 pbuf 被引用的次数
};
```

next: 指向下一个 pbuf 结构体，每个 pbuf 能存放的数据有限，如果应用有大量的数据的话我们就需要多个 pbuf 来存放，我们将同一个数据包的 pbuf 连接在一起形成一个链表，那么 next 字段就是实现这个链表的关键。

payload: 指向该 pbuf 的数据存储区的首地址，STM32F407 内部网络模块接收到数据，并将数据提交给 LWIP 的时候，就是将数据存储在 payload 指定的存储区中。同样在发送数据的时候将 payload 所指向的存储区数据转给 STM32F407 的网络模块去发送。

tot_len: 我们在接收或发送数据的时候数据会存放在 pbuf 链表中，tot_len 字段就表示当前 pbuf 和链表中以后所有 pbuf 总的长度。

len: 当前 pbuf 总数据的长度

type: 当前 pbuf 类型，一共有四种：PBUF_RAM、PBUF_ROM、PBUF_REF 和 PBUF_POOL。

flag: 保留位

ref: 该 pbuf 被引用的次数，当还有其他指针指向这个 pbuf 的时候 ref 字段就加一。

PBUF_RAM 类型的 pbuf 是通过内存堆分配得到的，PBUF_RAM 类型的 pbuf 如图 1.3.4.1 所示。从图 1.3.4.1 中可以看出 payload 并未指向数据区的起始地址，而是隔了一段区域，这段区域就是 offset，在里面通常存放 TCP 报文首部，IP 首部、以太网帧首部等等。

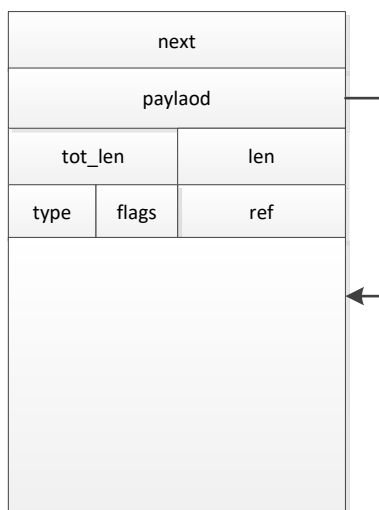


图 1.3.4.1 PBUF_RAM 类型 pbuf

PBUF_POOL 类型的 pbuf，PBUF_POOL 是通过内存池分配的到的，PBUF_POOL 类型的 pbuf 如图 1.3.4.2 所示。从图 1.3.4.2 中可以看出 pbuf 链表的第一个 pbuf 的 payload 未指向数据区的起始位置，原因同 PBUF_RAM 一样，用来存放一些首部的，pbuf 链表后面的 pbuf 结构体中 payload 字段就指向了数据区的起始位置。

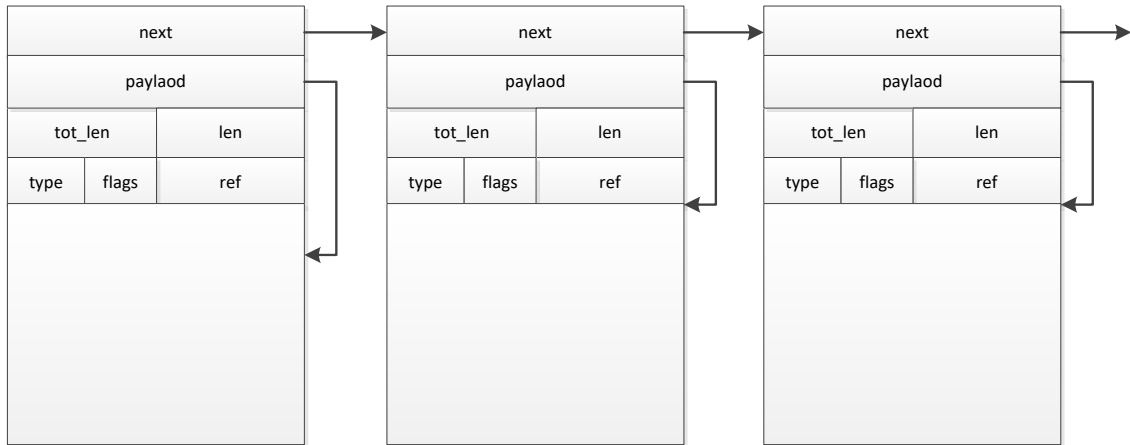


图 1.3.4.2 PBUF_POLL 类型的 pbuf

3) LWIP 网络接口管理

在 LWIP 中对于网络接口的描述是通过一个 `netif` 结构体完成的，`netif` 结构体在 `netif.h` 文件中有定义，定义如下，由于 `netif` 结构体比较大，这里只保留了一些比较重要的字段，具体的 `netif` 结构体定义请查阅 `netif.h` 文件。

```
struct netif {
    struct netif *next;           //指向下一个 netif 结构体
    ip_addr_t ip_addr;           //网络接口 IP 地址
    ip_addr_t netmask;           //子网掩码
    ip_addr_t gw;                //默认网关
    netif_input_fn input;        //IP 层接收数据函数
    netif_output_fn output;      //IP 层发送数据包调用
    netif_linkoutput_fn linkoutput; //底层数据包发送
    void *state;                 //设备的状态信息
    u16_t mtu;                   //该网络接口最大允许传输的数据长度
    u8_t hwaddr_len;             //物理地址长度
    u8_t hwaddr[NETIF_MAX_HWADDR_LEN]; //该网络接口的物理地址
    u8_t flags;                  //该网络接口的状态和属性
    char name[2];                //该网络接口的名字
    u8_t num;                    //该网络接口的编号
}
```

next: 该字段指向下一个 `netif` 类型的结构体，因为 LWIP 可以支持多个网络接口，当设备有多个网络接口的话 LWIP 就会把所有的 `netif` 结构体组成链表来管理这些网络接口。

ipaddr, netmask 和 gw 为分别为网络接口的 IP 地址、子网掩码和默认网关。

input: 此字段为一个函数，这个函数将网卡接收到的数据交给 IP 层。

output: 此字段为一个函数，当 IP 层向接口发送一个数据包时调用此函数。这个函数通常首先解析硬件地址，然后发送数据包。此字段我们一般使用 `etharp.c` 中的 `etharp_output()` 函数。

linkoutput: 此字段为一个函数，该函数被 ARP 模块调用，完成网络数据的发送。上面说的 `etharp_output` 函数将 IP 数据包封装成以太网数据帧以后就会调用 `linkoutput` 函数将数据发送出去。

state: 用来定义一些关于接口的信息，用户可以自行设置。

mtu: 网络接口所能传输的最大数据长度，一般设置为 1500

hwaddr_len 和 hwaddr 表示网络接口物理地址长度和具体的地址信息。

flags: 表示网络接口的状态和属性等信息，是很重要的字段，包括网卡功能使能，广播使能，ARP 使能等。

name: 网路接口的名字

num: 网络接口的编号

以上我们只列出了 netif 结构体中几个比较重要的字段，我们对于网络接口的初始化就是给这些字段赋值。

我们对于网络接口的初始化就是对 netif 结构体中各个字段的赋值。

1.3.5 添加 LWIP 源文件

本节中我们将 LWIP 源码添加到我们的工程中，我们在工程目录中新建一个 LWIP 文件夹，在这个文件夹中我们放置所有关于 LWIP 的文件，新建完成后我们将 LWIP 的源码 lwip-1.4.1 拷贝到 LWIP 文件夹下，如图 1.3.5.1 所示。



图 1.3.5.1 拷贝 LWIP 源码到工程中

将 LWIP 源文件拷贝到工程文件夹中以后我们就将里面的文件添加到工程中，按照图 1.3.4.2 所示添加到工程中，

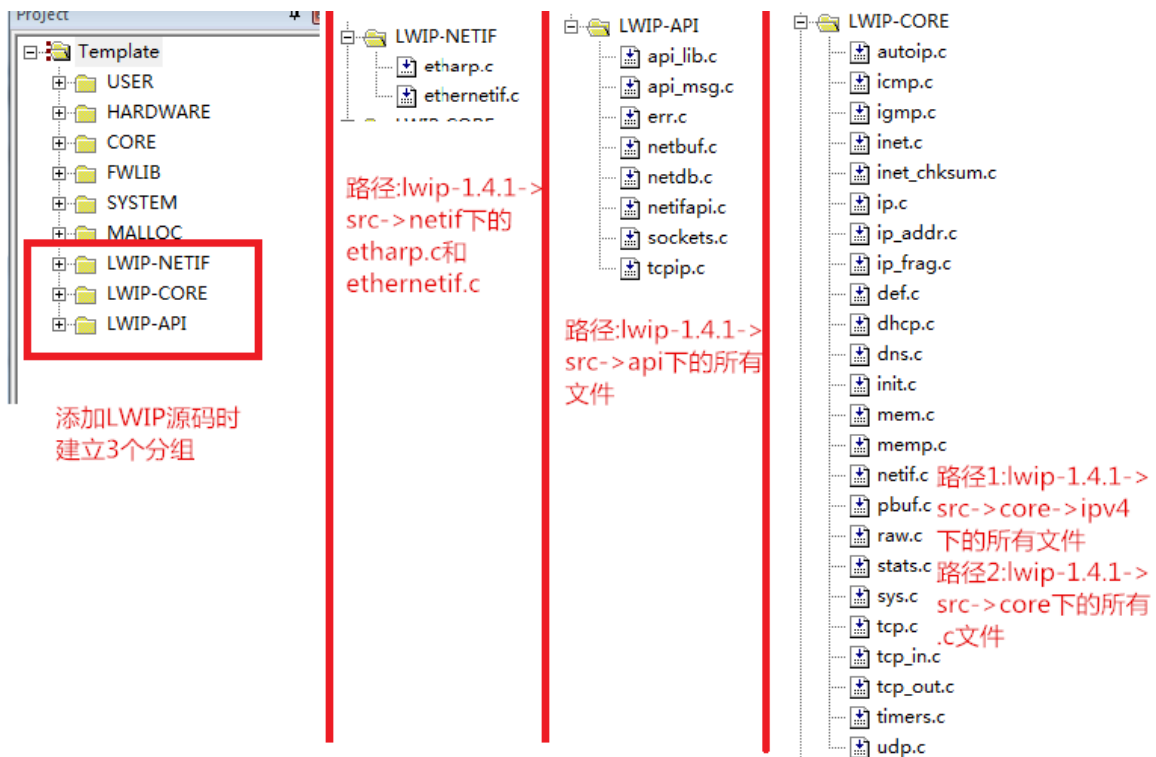


图 1.3.5.2 在工程中添加 LWIP 源码

我们将.c 文件添加到工程以后，还需要添加 LWIP 源码中的头文件路径，头文件路径按照图 1.3.5.3 所示添加，这时如果我们编译的话会有很多错误提示，这个不用管。

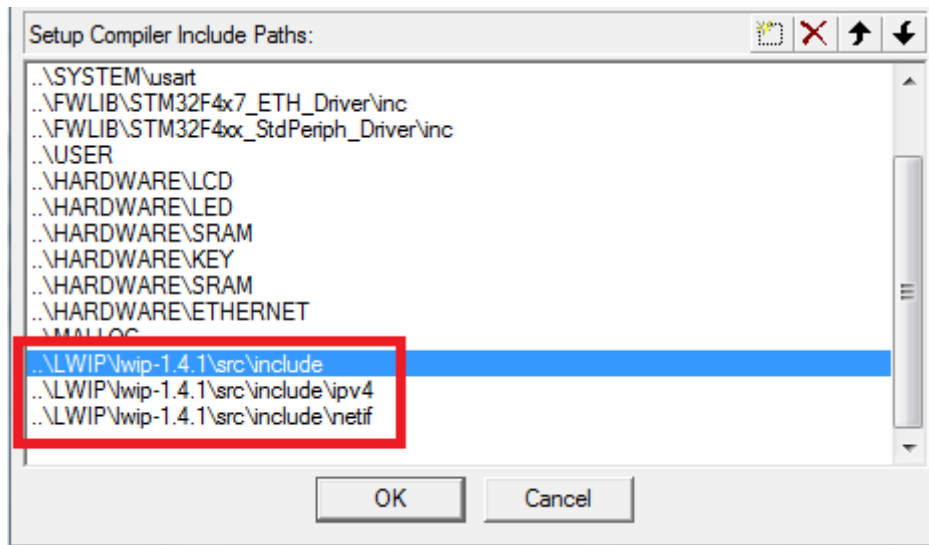


图 1.3.5.3 添加 LWIP 源码头文件

1.3.6 添加中间文件

我们上面只是将 LWIP 源文件和以太网的驱动都添加到工程中，要将以太网驱动和 LWIP 连接起来还需要一些其他文件，而这些文件非常重要，我们下面就讲解如何添加这些文件。

1) 添加 arch 文件

打开我们的网络实验 1 LWIP 无操作系统移植实验的 LWIP 文件夹可以发现有一个 arch 文件夹，将这个文件夹复制到自己工程中，在 arch 中有 5 个文件 cc.h、cpu.h、perf.h、sys_arch.h 和 sys_arch.c。根据 sys_arch.txt 中的描述，cc.h 主要完成了协议栈内部使用的数据类型的定义，如果使用操作系统的话还有临界代码区保护等等，cc.h 文件内容如下所示。

```
#ifndef __CC_H__
#define __CC_H__
#include "cpu.h"
#include "stdio.h"
//定义与平台无关的数据类型
typedef unsigned char u8_t; //无符号 8 位整数
typedef signed char s8_t; //有符号 8 位整数
typedef unsigned short u16_t; //无符号 16 位整数
typedef signed short s16_t; //有符号 16 位整数
typedef unsigned long u32_t; //无符号 32 位整数
typedef signed long s32_t; //有符号 32 位整数
typedef u32_t mem_ptr_t; //内存地址型数据
typedef int sys_prot_t; //临界保护型数据

//使用操作系统时的临界区保护，这里以 UCOS II 为例
//当定义了 OS_CRITICAL_METHOD 时就说明使用了 UCOS II
```

```

#if OS_CRITICAL_METHOD == 1
#define SYS_ARCH_DECL_PROTECT(lev)
#define SYS_ARCH_PROTECT(lev)      CPU_INT_DIS()
#define SYS_ARCH_UNPROTECT(lev)    CPU_INT_EN()
#endif

#if OS_CRITICAL_METHOD == 3
#define SYS_ARCH_DECL_PROTECT(lev)  u32_t lev
    //UCOS II 中进入临界区,关中断
    #define SYS_ARCH_PROTECT(lev)    lev = OS_CPU_SR_Save()
    //UCOS II 中退出 A 临界区, 开中断
    #define SYS_ARCH_UNPROTECT(lev)   OS_CPU_SR_Restore(lev)
#endif

//根据不同的编译器定义一些符号
#if defined (__ICCARM__)
#define PACK_STRUCT_BEGIN
#define PACK_STRUCT_STRUCT
#define PACK_STRUCT_END
#define PACK_STRUCT_FIELD(x) x
#define PACK_STRUCT_USE_INCLUDES

#elif defined (__CC_ARM)
#define PACK_STRUCT_BEGIN __packed
#define PACK_STRUCT_STRUCT
#define PACK_STRUCT_END
#define PACK_STRUCT_FIELD(x) x

#elif defined (__GNUC__)
#define PACK_STRUCT_BEGIN
#define PACK_STRUCT_STRUCT __attribute__((packed))
#define PACK_STRUCT_END
#define PACK_STRUCT_FIELD(x) x

#elif defined (__TASKING__)
#define PACK_STRUCT_BEGIN
#define PACK_STRUCT_STRUCT
#define PACK_STRUCT_END
#define PACK_STRUCT_FIELD(x) x

#endif

//LWIP 用 printf 调试时使用到的一些类型

```

```

#define U16_F "4d"
#define S16_F "4d"
#define X16_F "4x"
#define U32_F "8ld"
#define S32_F "8ld"
#define X32_F "8lx"

//宏定义
#ifndef LWIP_PLATFORM_ASSERT
#define LWIP_PLATFORM_ASSERT(x) \
do \
{   printf("Assertion \"%s\" failed at line %d in %s\r\n", x, __LINE__, __FILE__); \
} while(0)
#endif

#ifndef LWIP_PLATFORM_DIAG
#define LWIP_PLATFORM_DIAG(x) do {printf x;} while(0)
#endif
#endif

```

cpu.h 用来定义 CPU 的大小端模式, 因为 STM32 是小端模式, 因此这里定义 BYTE_ORDER 为小端模式, cpu.h 文件代码如下。

```

#ifndef __CPU_H__
#define __CPU_H__
#define BYTE_ORDER LITTLE_ENDIAN    //小端模式
#endif

```

perf.h 是和系统测量与统计相关的文件, 我们不使用任何的测量和统计, 因此这个文件中的两个宏定义为空, 代码如下所示。

```

#ifndef __PERF_H__
#define __PERF_H__
#define PERF_START    //空定义
#define PERF_STOP(x)  //空定义
#endif

```

sys_arch.h 和 sys_arch.c 是在使用操作系统的时候才使用到的文件, 在这里我们只是在 sys_arch.h 文件中简单的实现了获取时间的函数 sys_now(), 代码如下所示, 在 lwip_localtime 为一个全局变量, 用来为 LWIP 提供时钟, 这个变量我们会在下面的移植中介绍。

```

u32_t sys_now(void)
{
    return lwip_localtime;
}

```

我们在工程中新建一个 LWIP-ARCH 分组, 并将 sys_arch.c 文件添加到这个分组中并且添加相应的头文件路径, 如图 1.3.6.1 所示。

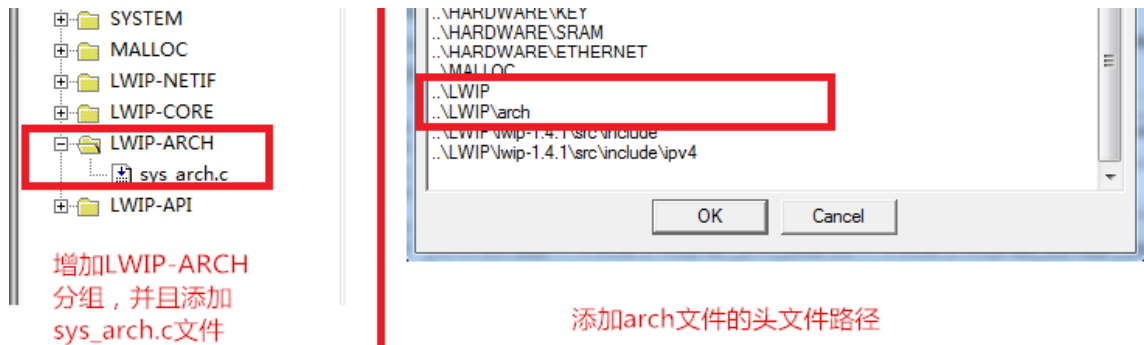


图 1.3.6.1 添加 LWIP-ARCH 分组并添加头文件路径

2) 添加 LWIP 通用文件

打开我们的网络实验 1 LWIP 无操作系统移植实验的 LWIP 文件夹可以发现有一个 lwip_app 文件夹，将这个文件夹复制到自己工程中，lwip_app 文件夹用来放我们以后所有实验的代码。在 lwip_app 下有一个 lwip_comm 文件夹，这个文件中有 lwip_comm.c、lwip_comm.h 和 lwipopts.h 这三个文件，lwip_comm.c 和 lwip_comm.h 是将 LWIP 源码和前面的以太网驱动库结合起来的桥梁！这两个文件非常重要，这两个文件有 ALIENTEK 提供。lwipopts.h 是用来裁剪和配置 LWIP 的文件，以后我们想要使用 LWIP 的什么功能的话就在这个文件中配置就行了。

同样的，我们在工程中新建一个 LWIP-APP 分组，并将 lwip_comm.c 文件添加到这个分组中并且添加相应的头文件路径，如图 1.3.6.2 所示

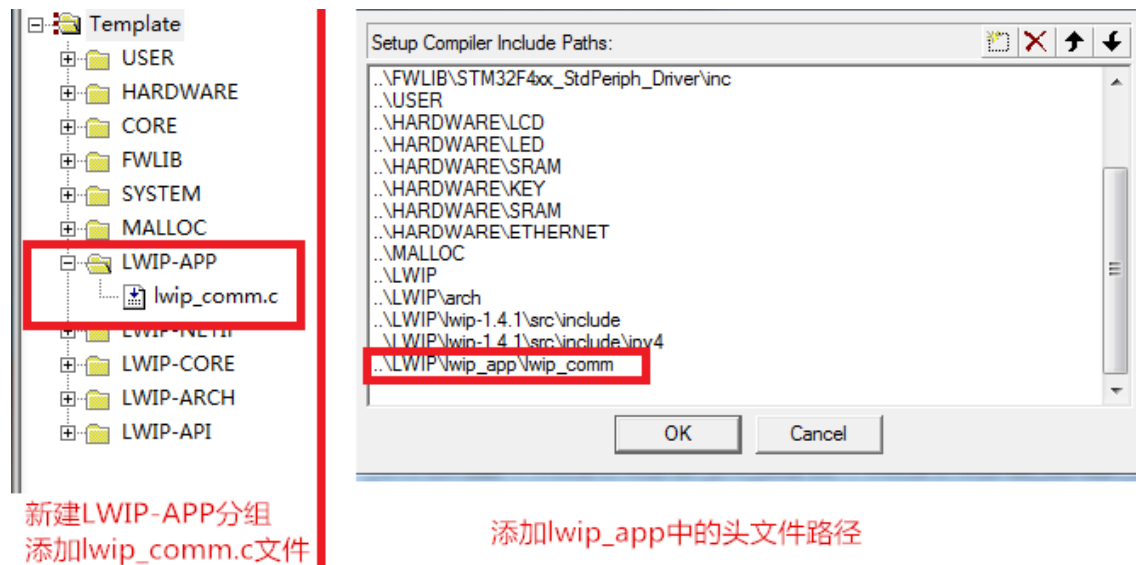


图 1.3.6.2 新建 LWIP-APP 分组并添加相应头文件

在 lwip_comm.h 中由 ALIENTEK 定义了一个重要的结构体 __lwip_dev，这个结构体如下。

```
typedef struct
{
    u8 mac[6];           //MAC 地址
    u8 remoteip[4];      //远端主机 IP 地址
    u8 ip[4];             //本机 IP 地址
    u8 netmask[4];        //子网掩码
    u8 gateway[4];        //默认网关的 IP 地址
}
```

```

vu8 dhcpstatus;    //dhcp 状态
                  //0,未获取 DHCP 地址;
                  //1,进入 DHCP 获取状态
                  //2,成功获取 DHCP 地址
                  //0XFF,获取失败.
}__lwip_dev;

```

这个结构体有本机 MAC 地址、远端主机 IP 地址、本机 IP 地址、子网掩码、默认网关和 DHCP 状态这几个成员变量。在 lwip_comm.c 中定义了一个 __lwip_dev 结构体类型变量 lwipdev，这是一个全局变量。

在 lwip_comm.c 中有一下 7 个函数：

```

u8 lwip_comm_mem_malloc(void)
void lwip_comm_mem_free(void)
void lwip_comm_default_ip_set(__lwip_dev *lwipx)
u8 lwip_comm_init(void)
void lwip_pkt_handle(void)
void lwip_periodic_handle()
void lwip_dhcp_process_handle(void)

```

首先是 lwip_comm_mem_malloc()函数，lwip_comm_mem_malloc()函数完成了对 mem.c 和 memp.c 中内存堆 ram_heap 和内存池 memp_memory 的内存分配，函数代码如下。

```

//lwip 中 mem 和 memp 的内存申请
//返回值:0,成功;
//    其他,失败
u8 lwip_comm_mem_malloc(void)
{
    u32 memsize;
    u32 ramheapsize;
    memsize=memp_get_memorysize();           //得到 memp_memory 数组大小
    memp_memory=mymalloc(SRAMIN,memsize);    //为 memp_memory 申请内存
    ramheapsize=LWIP_MEM_ALIGN_SIZE(MEM_SIZE)+2*\
    LWIP_MEM_ALIGN_SIZE(4*3)+MEM_ALIGNMENT;//得到 ram heap 大小
    ram_heap=mymalloc(SRAMIN,ramheapsize);   //为 ram_heap 申请内存
    if(!memp_memory||!ram_heap)//有申请失败的
    {
        lwip_comm_mem_free();
        return 1;
    }
    return 0;
}

```

lwip_comm_mem_free()函数用来释放内存堆 ram_heap 和内存池 memp_memory 的内存，函数比较简单。

lwip_comm_default_ip_set()函数用来设置默认地址，我们前面提过 __lwip_dev 结构体变量 lwipdev，lwip_comm_default_ip_set()函数就是用来设置 lwipdev 的各个成员变量的。因为 MAC 地址要为全球唯一，因此这里 MAC 地址的前 3 个字节我们设置为 2、0、0。后三个字节我们

取 STM32 的全球唯一 ID 的高三字节，当然在实际使用过程中也可以自行设置，只要保证在同一局域网中 MAC 地址不会重复就行。IP 地址、子网掩码、默认网关地址可以自行设置，这里我们设置 IP 地址为：192.168.1.30，子网掩码：255.255.255.0，默认网关：192.168.1.1。

接下来要介绍的 `lwip_comm_init` 函数是非常重要的一个函数，这个函数主要完成 LWIP 内核初始化、设置默认网卡并且打开指定的网卡，函数代码如下。

```
//LWIP 初始化(LWIP 启动的时候使用)
//返回值:0,成功
//      1,内存错误
//      2,LAN8720 初始化失败
//      3,网卡添加失败.
u8 lwip_comm_init(void)
{
    //调用 netif_add()函数时的返回值,用于判断网络初始化是否成功
    struct netif *Netif_Init_Flag;
    struct ip_addr ipaddr;          //ip 地址
    struct ip_addr netmask;         //子网掩码
    struct ip_addr gw;              //默认网关
    if(ETH_Mem_Malloc())return 1;    //内存申请失败
    if(lwip_comm_mem_malloc())return 1; //内存申请失败
    if(LAN8720_Init())return 2;      //初始化 LAN8720 失败
    lwip_init();                     //初始化 LWIP 内核
    lwip_comm_default_ip_set(&lwipdev); //设置默认 IP 等信息

#ifdef LWIP_DHCP                    //使用动态 IP
    ipaddr.addr = 0;
    netmask.addr = 0;
    gw.addr = 0;
#else                                //使用静态 IP
    IP4_ADDR(&ipaddr,lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
    IP4_ADDR(&netmask,lwipdev.netmask[0],lwipdev.netmask[1],\
    lwipdev.netmask[2],lwipdev.netmask[3]);
    IP4_ADDR(&gw,lwipdev.gateway[0],lwipdev.gateway[1],\
    lwipdev.gateway[2],lwipdev.gateway[3]);
    printf(" 网 卡 的 MAC 地 址 为 :.....%d.%d.%d.%d\r\n",lwipdev.mac[0],\
    lwipdev.mac[1],lwipdev.mac[2],lwipdev.mac[3],lwipdev.mac[4],lwipdev.mac[5]);
    printf(" 静 态 IP 地 址 .....%d.%d.%d.%d\r\n",lwipdev.ip[0],lwipdev.ip[1],\
    lwipdev.ip[2],lwipdev.ip[3]);
    printf(" 子 网 掩 码 .....%d.%d.%d.%d\r\n",lwipdev.netmask[0],\
    lwipdev.netmask[1],lwipdev.netmask[2],lwipdev.netmask[3]);
    printf(" 默 认 网 关 .....%d.%d.%d.%d\r\n",lwipdev.gateway[0],\
    lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);
#endif
    //向网卡列表中添加一个网口
```

```

Netif_Init_Flag=netif_add(&lwip_netif,&ipaddr,&netmask,&gw,NULL,\
    &ethernetif_init,&ethernet_input);
#if LWIP_DHCP           //如果使用 DHCP 的话
    lwipdev.dhcpstatus=0; //DHCP 标记为 0
    dhcp_start(&lwip_netif); //开启 DHCP 服务
#endif
if(Netif_Init_Flag==NULL)return 3; //网卡添加失败
else //网口添加成功后,设置 netif 为默认值,并且打开 netif 网口
{
    netif_set_default(&lwip_netif); //设置 netif 为默认网口
    netif_set_up(&lwip_netif);      //打开 netif 网口
}
return 0; //操作 OK.
}

```

上面这个函数主要完成以下功能:

1、调用 ETH_Mem_Malloc()和 lwip_comm_mem_malloc()这两个函数完成前面提到的那四个数组和内存堆 ram_heap 和内存池 memp_memory 的内存分配。

2、调用 LAN8720_Init()函数完成对 LAN8720 以及 STM32F407 的 MAC 和以太网 DMA 的初始化, LAN8720_Init()函数在 LAN8720.c 文件中, 由 ALIENTEK 提供。

3、调用 lwip_init 函数完成 LWIP 的内核初始化, lwip_init()通过调用各个模块的初始化函数来完成各个模块的初始化, 比如内存初始化函数、数据包结构初始化函数、网络接口初始化函数、IP 和 TCP 等的初始化函数, lwip_init()在 init.c 文件中, 属于 LWIP 源码。

4、调用 ip_comm_default_ip_set()函数设置静态地址等信息, 此函数由 ALIENTEK 提供。

5、判断是否使用 DHCP, 如果使用 DHCP 的话就通过 DHCP 服务来获取 IP 地址、子网掩码和默认网关等信息, 如果使用静态 IP 地址的话就用 ip_comm_default_ip_set()函数设置的地址信息。

6、我们通过 netif_add()函数来完成网卡的注册, netif_add()在 netif.c 文件中, 此函数属于 LWIP 源码。netif_add()函数中的参数 lwip_netif 是我们定义的一个网络接口, 这个函数除了使用上面说的 IP 地址, 子网掩码和默认网关作为参数外, 还使用了两个函数地址作为参数: ethernetif_init 和 ethernet_input, 这两个函数地址会被赋值给 netif 结构体的相关字段, ethernetif_init()在下一节会讲到, 这个函数由 ALIENTEK 提供, ethernet_input()函数在 etharp.c 文件中, 属于 LWIP 源码, 是 ARP 层数据包输入函数。

7、如果使用 DHCP 的话就开启 DHCP 服务, 通过调用 dhcp_start()函数开启 DHCP 服务。

8、当网卡注册成功后使用 netif_set_default()设置此网卡为默认网卡, 并且使用 netif_set_up()函数打开此网卡。

当程序中调用 lwip_comm_init()函数后, 网卡就可以工作了。STM32F407 内部网络模块接收数据有两种方式: 查询法和中断法, 在这里我们使用的是 STM32F407 以太网 DMA 的中断接收数据。前面我们介绍过 DMA 中断服务函数 ETH_IRQHandler(), 在中断服务函数中我们调用了 lwip_pkt_handle()函数来接收数据, lwip_pkt_handle()函数代码如下。

```

//当接收到数据后调用
void lwip_pkt_handle(void)
{
    //从网络缓冲区中读取接收到的数据包并将其发送给 LWIP 处理
}

```



```
ethernetif_input(&lwip_netif);
}
```

可以看出 `lwip_pkt_handle()` 函数其实只是对 `ethernetif_input()` 函数的简单封装，通过调用 `ethernetif_input()` 函数从指定的网络接口中接收数据，`ethernetif_input()` 函数是在 `ethernetif.c` 文件中定义的，我们在下一节会讲到。

LWIP 内核中有许多的周期性的定时器，相对应的定时处理函数也需要被周期性调用的，因为没有使用操作系统，所有需要我们自己使用定时器来实现。这里我们使用 STM32F407 定时器 3 提供这个系统时钟，定时器 3 定时周期为 10ms，定时器 3 的中断服务函数如下，`lwip_localtime` 是一个全局变量，在 `lwip_comm.c` 文件中有定义。我们新建 `timer.c` 文件放关于定时器 3 的程序，然后将 `timer.c` 添加到 HARDWARE 组下面，并且添加相应的头文件路径。

//定时器 3 中断服务函数

```
void TIM3_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM3,TIM_IT_Update)==SET) //溢出中断
    {
        lwip_localtime +=10; //加 10
    }
    TIM_ClearITPendingBit(TIM3,TIM_IT_Update); //清除中断标志位
}
```

我们上面说了 LWIP 中内核中有许多的周期性定时器，如 TCP 定时器、ARP 定时器、IP 如果使用 DHCP 的话还有 DHCP 定时器等等，我们可以将这些定时器封装在一个函数里面，然后周期性这个函数就行了，LWIP 协议栈要求的是每 250ms 处理一次 TCP 定时器，每 5s 处理一次 ARP 定时器，每 500ms 处理一次 DHCP 精细处理定时器，每 60s 执行一次 DHCP 的粗糙处理。我们就根据这个要求来编写 `lwip_periodic_handle()` 函数，函数代码如下，最后我们只要周期性的调用 `lwip_periodic_handle()` 函数就可以完成对 LWIP 内核的定时处理函数的周期性调用。

//LWIP 轮询任务

```
void lwip_periodic_handle()
{
#ifdef LWIP_TCP
    //每 250ms 调用一次 tcp_tmr()函数
    if (lwip_localtime - TCPTimer >= TCP_TMR_INTERVAL)
    {
        TCPTimer = lwip_localtime;
        tcp_tmr();
    }
#endif
    //ARP 每 5s 周期性调用一次
    if ((lwip_localtime - ARPTimer) >= ARP_TMR_INTERVAL)
    {
        ARPTimer = lwip_localtime;
        etharp_tmr();
    }
}
```

```

#if LWIP_DHCP //如果使用 DHCP 的话
    //每 500ms 调用一次 dhcp_fine_tmr()
    if (lwip_localtime - DHCPfineTimer >= DHCP_FINE_TIMER_MSECS)
    {
        DHCPfineTimer = lwip_localtime;
        dhcp_fine_tmr();
    }
    if ((lwipdev.dhcpstatus != 2)&&(lwipdev.dhcpstatus != 0XFF))
    {
        lwip_dhcp_process_handle(); //DHCP 处理
    }
}
//每 60s 执行一次 DHCP 粗糙处理
if (lwip_localtime - DHCPcoarseTimer >= DHCP_COARSE_TIMER_MSECS)
{
    DHCPcoarseTimer = lwip_localtime;
    dhcp_coarse_tmr();
}
#endif
}

```

lwip_dhcp_process_handle()函数为 DHCP 处理函数，函数代码如下。

```

//DHCP 处理任务
void lwip_dhcp_process_handle(void)
{
    u32 ip=0,netmask=0,gw=0;
    switch(lwipdev.dhcpstatus)
    {
        case 0: //开启 DHCP
            dhcp_start(&lwip_netif);
            lwipdev.dhcpstatus = 1; //等待通过 DHCP 获取到的地址
            printf("正在查找 DHCP 服务器,请稍等.....\r\n");
            break;
        case 1: //等待获取到 IP 地址
            {
                ip=lwip_netif.ip_addr.addr; //读取新 IP 地址
                netmask=lwip_netif.netmask.addr; //读取子网掩码
                gw=lwip_netif.gw.addr; //读取默认网关

                if(ip!=0) //正确获取到 IP 地址的时候
                {
                    lwipdev.dhcpstatus=2; //DHCP 成功
                    printf("网卡 en 的 MAC 地址为:.....%d.%d.%d.%d.%d.%d\r\n",\
                        lwipdev.mac[0],lwipdev.mac[1],lwipdev.mac[2],lwipdev.mac[3],\

```

```

lwipdev.mac[4],lwipdev.mac[5]);
//解析出通过 DHCP 获取到的 IP 地址
lwipdev.ip[3]=(uint8_t)(ip>>24);
lwipdev.ip[2]=(uint8_t)(ip>>16);
lwipdev.ip[1]=(uint8_t)(ip>>8);
lwipdev.ip[0]=(uint8_t)(ip);
printf("通过 DHCP 获取到 IP 地址.....%d.%d.%d.%d\r\n",\
lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
//解析通过 DHCP 获取到的子网掩码地址
lwipdev.netmask[3]=(uint8_t)(netmask>>24);
lwipdev.netmask[2]=(uint8_t)(netmask>>16);
lwipdev.netmask[1]=(uint8_t)(netmask>>8);
lwipdev.netmask[0]=(uint8_t)(netmask);
printf("通过 DHCP 获取到子网掩码.....%d.%d.%d.%d\r\n",\
lwipdev.netmask[0],lwipdev.netmask[1],lwipdev.netmask[2],lwipdev.netmask[3]);
//解析出通过 DHCP 获取到的默认网关
lwipdev.gateway[3]=(uint8_t)(gw>>24);
lwipdev.gateway[2]=(uint8_t)(gw>>16);
lwipdev.gateway[1]=(uint8_t)(gw>>8);
lwipdev.gateway[0]=(uint8_t)(gw);
printf("通过 DHCP 获取到的默认网关.....%d.%d.%d.%d\r\n",\
lwipdev.gateway[0],lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);
}else if(lwip_netif.dhcp->tries>LWIP_MAX_DHCP_TRIES)
{
    //通过 DHCP 服务获取 IP 地址失败,且超过最大尝试次数
    lwipdev.dhcpstatus=0XFF;//DHCP 超时失败.
    //使用静态 IP 地址
    IP4_ADDR(&(lwip_netif.ip_addr),lwipdev.ip[0],lwipdev.ip[1],\
lwipdev.ip[2],lwipdev.ip[3]);
    IP4_ADDR(&(lwip_netif.netmask),lwipdev.netmask[0],\
lwipdev.netmask[1],lwipdev.netmask[2],lwipdev.netmask[3]);
    IP4_ADDR(&(lwip_netif.gw),lwipdev.gateway[0],lwipdev.gateway[1],\
lwipdev.gateway[2],lwipdev.gateway[3]);
    printf("DHCP 服务超时,使用静态 IP 地址!\r\n");
    printf("网卡 en 的 MAC 地址为:.....%d.%d.%d.%d.%d.%d\r\n",\
lwipdev.mac[0],lwipdev.mac[1],lwipdev.mac[2],lwipdev.mac[3],\
lwipdev.mac[4],lwipdev.mac[5]);
    printf("静态 IP 地址.....%d.%d.%d.%d\r\n",\
lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
    printf("子网掩码.....%d.%d.%d.%d\r\n",\
lwipdev.netmask[0],lwipdev.netmask[1],lwipdev.netmask[2],lwipdev.netmask[3]);
    printf("默认网关.....%d.%d.%d.%d\r\n",\
lwipdev.gateway[0],lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);

```

```

    }
}
break;
default : break;
}
}

```

我们通过判断 `lwipdev` 结构体的 `dhcpstatus` 字段判断是否使用 DHCP 服务，当 `dhcpstatus=0` 时表示开启 DHCP，我们调用 `dhcp_start()` 函数开启相应网络接口的 DHCP 服务 `dhcp_start()` 函数由 LWIP 源码提供。开启 DHCP 以后让 `dhcpstatus=1`，表示开始进行 DHCP，等待 DHCP 完成。当 DHCP 完成以后让 `dhcpstatus=2`，表示 DHCP 成功。但是当 DHCP 重试次数大于 `LWIP_MAX_DHCP_TRIES` 时，意味着 DHCP 失败，这是 `dhcpstatus=0XFF`，表示 DHCP 失败，并且使用静态 IP 地址。

3) 添加 ethernetif.h 文件

打开我们的网络实验 1 LWIP 无操作系统移植实验，LWIP->lwip1.4.1->src->include->netif 中我们会发现有个 `ethernetif.h` 文件，这个文件在 LWIP 源码中是不存在的，这个文件由 ALIENTEK 提供，将 `ethernetif.h` 文件复制到自己工程的相应位置，如图 1.3.6.3 所示。

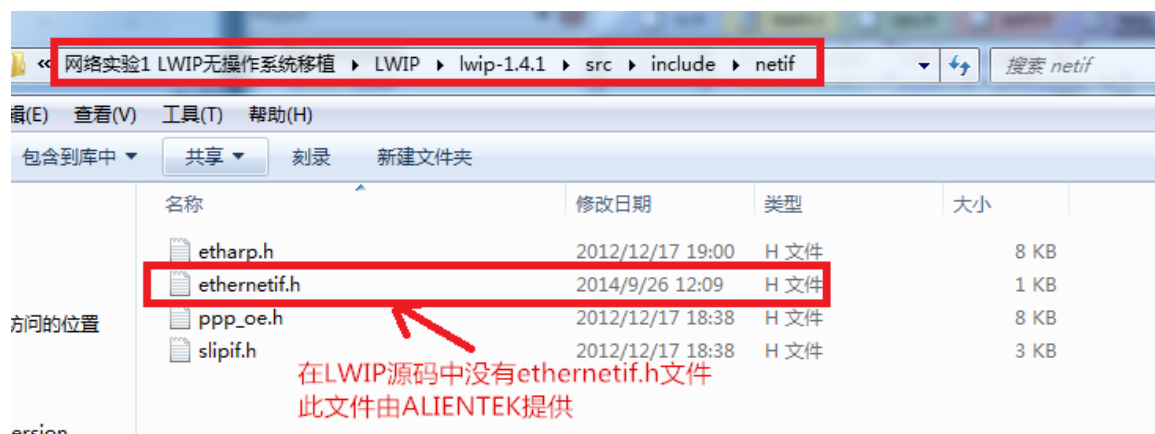


图 1.3.6.3 ethernetif.h 文件

1.3.7 LWIP 源码修改

在上面添加中间文件完成后，我们还有修改一下 LWIP 的源码，中间文件连接 LWIP 底层驱动和 LWIP，这样或多或少会对 LWIP 的源码做出一点小的改动，下面我们就讲解如何修改 LWIP 源码。

1) 修改 LWIP 源文件名字

我们按路径：lwip-1.4.1->src->core 可以发现在 core 文件下有一个 `sys.c` 文件，按路径 lwip-1.4.1->src->include->lwip 可以发现有一个 `sys.h` 文件。`sys.c` 和 `sys.h` 这两个文件和我们的 SYSTEM 文件中的 `sys.c` 和 `sys.h` 重名，因此我们将 LWIP 中 `sys.c` 和 `sys.h` 改为 `lwip_sys.c` 和 `lwip_sys.h`，然后在工程中将 LWIP 源码里面的 `#include "sys.h"` 代码也要改掉，更改为 `#include "lwip_sys.h"`。

2) 修改 ethernetif.c 文件

`ethernetif.c` 的文件路径为：LWIP->lwip1.4.1->src->netif。用我们网络实验 1 LWIP 无操作系统移植实验中的 `ethernetif.c` 文件替代 LWIP 源码中的这个文件，在这个文件中有 5 个函数。

```
static err_t low_level_init(struct netif *netif)
```

```
static err_t low_level_output(struct netif *netif, struct pbuf *p)
static struct pbuf * low_level_input(struct netif *netif)
err_t ethernetif_input(struct netif *netif)
err_t ethernetif_init(struct netif *netif)
```

这 5 个函数是移植 LWIP 的重点，其中前 3 个与网卡密切相关，low_level_init 函数主要完成网卡的复位、协议栈网络接口管理结构体 netif 中相关字段的初始化、发送和接收 DMA 描述符链表的初始化，硬件帧校验的开启并且打开以太网。low_level_init 函数如下：

```
//由 ethernetif_init()调用用于初始化硬件
//netif:网卡结构体指针
//返回值:ERR_OK,正常
//      其他,失败
static err_t low_level_init(struct netif *netif)
{
#ifdef CHECKSUM_BY_HARDWARE
    int i;
#endif
    netif->hwaddr_len = ETHARP_HWADDR_LEN; //设置 MAC 地址长度,为 6 个字节
    //初始化 MAC 地址,设置什么地址由用户自己设置,但是不能与网络中
    //其他设备 MAC 地址重复
    netif->hwaddr[0]=lwipdev.mac[0];
    netif->hwaddr[1]=lwipdev.mac[1];
    netif->hwaddr[2]=lwipdev.mac[2];
    netif->hwaddr[3]=lwipdev.mac[3];
    netif->hwaddr[4]=lwipdev.mac[4];
    netif->hwaddr[5]=lwipdev.mac[5];
    netif->mtu=1500; //最大允许传输单元,允许该网卡广播和 ARP 功能
    netif->flags=NETIF_FLAG_BROADCAST\
    NETIF_FLAG_ETHARP|NETIF_FLAG_LINK_UP;

    //向 STM32F4 的 MAC 地址寄存器中写入 MAC 地址
    ETH_MACAddressConfig(ETH_MAC_Address0,netif->hwaddr);
    ETH_DMATxDescChainInit(DMATxDscrTab, Tx_Buff, ETH_TXBUFNB);
    ETH_DMARxDescChainInit(DMARxDscrTab, Rx_Buff, ETH_RXBUFNB);
#ifdef CHECKSUM_BY_HARDWARE    //使用硬件帧校验
    for(i=0;i<ETH_TXBUFNB;i++)
    {
        //使能 TCP,UDP 和 ICMP 的发送帧校验,TCP,UDP 和 ICMP 的接收帧校验在 DMA 中配置了

        ETH_DMATxDescChecksumInsertionConfig(&DMATxDscrTab[i],\
        ETH_DMATxDesc_ChecksumTCPUDPICMPFull);
    }
#endif
    ETH_Start(); //开启 MAC 和 DMA
```

```

return ERR_OK;
}

```

low_level_output 函数用于发送数据，将 LWIP 协议栈准备好的数据通过网卡发送出去，我们通过 ETH_GetCurrentTxBuffer() 函数获取当前发送 DMA 描述符的数据缓冲区，并将要发送的 pbuf 数据拷贝到这个缓冲区中，然后调用 ETH_Tx_Packet 函数将数据发送出去，low_level_output 函数代码如下：

```

static err_t low_level_output(struct netif *netif, struct pbuf *p)
{
    u8 res;
    struct pbuf *q;
    int l = 0;
    //获取当前要发送的 DMA 描述符中的缓冲区地址
    u8 *buffer=(u8 *)ETH_GetCurrentTxBuffer();
    for(q=p;q!=NULL;q=q->next)
    {
        memcpy((u8_t*)&buffer[l], q->payload, q->len);
        l=l+q->len;
    }
    res=ETH_Tx_Packet(l);          //调用 ETH_Tx_Packet 函数发送数据
    if(res==ETH_ERROR)return ERR_MEM;//返回错误状态
    return ERR_OK;
}

```

low_level_input 函数是从网卡中提取数据，并将数据封装在 pbuf 结构体中供 LWIP 使用，STM32F407 在接收到数据后会存放在我们前面提到的接收 DMA 描述符中。我们需要将存放在接收 DMA 描述符中的数据拷贝出来放到 pbuf 结构体中，然后将这个 pbuf 返回，low_level_input 函数就是完成这个功能的，代码如下所示。

///用于接收数据包的最底层函数

//netif:网卡结构体指针

//返回值:pbuf 数据结构体指针

```

static struct pbuf * low_level_input(struct netif *netif)
{
    struct pbuf *p, *q;
    u16_t len;
    int l=0;
    FrameTypeDef frame;
    u8 *buffer;
    p = NULL;
    frame=ETH_Rx_Packet();
    len=frame.length;//得到包大小
    buffer=(u8 *)frame.buffer;//得到包数据地址
    p=pbuf_alloc(PBUF_RAW,len,PBUF_POOL);//pbufs 内存池分配 pbuf
    if(p!=NULL)
    {

```

```

    for(q=p;q!=NULL;q=q->next)
    {
        memcpy((u8_t*)q->payload,(u8_t*)&buffer[l], q->len);
        l=l+q->len;
    }
}
//设置 Rx 描述符 OWN 位,buffer 重归 ETH DMA
frame.descriptor->Status=ETH_DMARxDesc_OWN;
//当 Rx Buffer 不可用位(RBUS)被设置的时候,重置它.恢复传输
if((ETH->DMASR&ETH_DMASR_RBUS)!=(u32)RESET)
{
    ETH->DMASR=ETH_DMASR_RBUS;//重置 ETH DMA RBUS 位
    ETH->DMARPDR=0;//恢复 DMA 接收
}
return p;
}

```

在 ethernetif.c 中还剩下两个函数：ethernetif_input()和 ethernetif_init()。ethernetif_input()函数主要是对 low_level_input()函数做封装，然后将接收到的数据送入指定的网卡结构中。ethernetif_input()函数代码如下：

```

//网卡接收数据(lwip 直接调用)
//netif:网卡结构体指针
//返回值:ERR_OK,发送正常
//      ERR_MEM,发送失败
err_t ethernetif_input(struct netif *netif)
{
    err_t err;
    struct pbuf *p;
    p=low_level_input(netif);    //调用 low_level_input 函数接收数据
    if(p==NULL) return ERR_MEM;
    err=netif->input(p, netif);//调用 netif 结构体中的 input 字段(一个函数)来处理数据包
    if(err!=ERR_OK)
    {
        LWIP_DEBUGF(NETIF_DEBUG,("ethernetif_input: IP input error\n"));
        pbuf_free(p);
        p = NULL;
    }
    return err;
}

```

ethernetif_init()函数为 low_level_init()函数的简单封装，并且初始化了 netif 的相关字段，ethernetif_init()函数代码如下。

```

//使用 low_level_init()函数来初始化网络
//netif:网卡结构体指针
//返回值:ERR_OK,正常

```



```
// 其他,失败
err_t ethernetif_init(struct netif *netif)
{
    LWIP_ASSERT("netif!=NULL",(netif!=NULL));
#ifdef LWIP_NETIF_HOSTNAME //LWIP_NETIF_HOSTNAME
    netif->hostname="lwip"; //初始化名称
#endif
    netif->name[0]=IFNAME0; //初始化变量 netif 的 name 字段
    netif->name[1]=IFNAME1; //在文件外定义这里不用关心具体值
    netif->output=etharp_output; //IP 层发送数据包函数
    netif->linkoutput=low_level_output; //ARP 模块发送数据包函数
    low_level_init(netif); //底层硬件初始化函数
    return ERR_OK;
}
```

3)修改 mem.c 和 memp.c 文件

根据我们前面讲过的 LWIP 动态内存管理技术,我们知道 LWIP 有一个内存堆 ram_heap 和内存池 memp_memory,这两个是 LWIP 的内存来源。这两个分别在 mem.c 和 memp.c 中,我们将这两个数组改用 ALIENTEK 的内存分配函数对其进行内存分配。在 mem.c 文件中我们将 ram_heap 数组注销掉,定义为指向 u8_t 的指针,如图 1.3.7.1 所示。

```
170 // If so, make sure the memory at that location is big enough (see below on
171 // * how that space is calculated). */
172 #ifndef LWIP_RAM_HEAP_POINTER
173 /** the heap. we need one struct mem at the end and some room for alignment */
174 //u8_t ram_heap[MEM_SIZE_ALIGNED + (2*SIZEOF_STRUCT_MEM) + MEM_ALIGNMENT];
175 //ram_heap在lwip_comm.c文件中的lwip_comm_mem_malloc()函数采用ALIENTEK动态内存管理函数分配内存
176 u8_t *ram_heap;
177 #define LWIP_RAM_HEAP_POINTER ram_heap
178 #endif /* LWIP_RAM_HEAP_POINTER */
```

图 1.3.7.1 将 ram_heap 数组改为指针(mem.c 文件中)

同样我们也将 memp.c 文件中将 memp_memory 数组屏蔽掉改为指针,如图 1.3.7.2 所示。

```
168
169 /** This is the actual memory used by the pools (all pools in one big block). */
170 //static u8_t memp_memory[MEM_ALIGNMENT - 1
171 //define LWIP_MEMPOOL(name,num,size,desc) + ( (num) * (MEM_SIZE + MEM_ALIGN_SIZE(size) ) )
172 //include "lwip/memp_std.h"
173 //];
174 //memp_memory在lwip_comm.c文件中的lwip_comm_mem_malloc()函数采用ALIENTEK动态内存管理函数分配内存
175 u8_t *memp_memory;
176 #endif /* MEM_SEPARATE_POOLS */
177
```

图 1.3.7.2 memp_memory 数组改为指针(memp.c 文件中)

我们还要在 memp.c 文件中添加 memp_get_memorysize()函数用来获取 memp_memory 数组的大小,memp_get_memorysize()函数代码如下。memp_memory 我们会在 lwip_comm.c 文件中使用动态内存管理函数为其分配内存。

```
u32_t memp_get_memorysize(void)
{
    u32_t length=0;
    length=(
        MEM_ALIGNMENT-1
#define LWIP_MEMPOOL (name,num,size,desc) +((num)*\
        (MEM_SIZE+MEM_ALIGN_SIZE(size)))
```

```

#include "lwip/memp_std.h"

);

return length;

}

```

在 memp.c 文件中添加 memp_get_memorysize() 函数，如图 1.3.7.3 所示。

```

332
333
334 //得到memp_memory数组大小
335 u32_t memp_get_memorysize(void)
336 {
337     u32_t length=0;
338     length=(
339         MEM_ALIGNMENT-1 //全局型数组 为所有POOL分配的内存空间
340         //MEMP_SIZE表示需要在每个POOL头部预留的空间 MEMP_SIZE = 0
341         #define LWIP_MEMPOOL(name,num,size,desc)+((num)*(MEMP_SIZE+MEMP_ALIGN_SIZE(size)))
342         #include "lwip/memp_std.h"
343     );
344     return length;
345 }
346

```

图 1.3.7.3 memp_get_memorysize() 函数(memp.c 文件中)

4) 修改 icmp.c 文件

我们需要修改 icmp.c 文档使其支持硬件帧校验，修改部分如图 1.3.7.4 所示。图中红框部分为 icmp.c 的源码，我们将其注销掉，蓝框部分是我们需要添加进去的代码，这部分代码是由 ST 提供的。

```

193
194
195 // #if CHECKSUM_GEN_ICMP
196 // /* adjust the checksum */
197 // if (iecho->chksum >= PP_HTONS(0xffff - (ICMP_ECHO << 8))) {
198 //     iecho->chksum += PP_HTONS(ICMP_ECHO << 8) + 1;
199 // } else {
200 //     iecho->chksum += PP_HTONS(ICMP_ECHO << 8);
201 // }
202 // #else /* CHECKSUM_GEN_ICMP */
203 //     iecho->chksum = 0;
204 // #endif /* CHECKSUM_GEN_ICMP */
205 /* This part of code has been modified by ST's MCD Application Team */
206 /* To use the Checksum Offload Engine for the outgoing ICMP packets,
207    the ICMP checksum field should be set to 0, this is required only for Tx ICMP*/
208 #ifdef CHECKSUM_BY_HARDWARE
209     iecho->chksum = 0;
210 #else
211     /* adjust the checksum */
212     if (iecho->chksum >= htons(0xffff - (ICMP_ECHO << 8))) {
213         iecho->chksum += htons(ICMP_ECHO << 8) + 1;
214     } else {
215         iecho->chksum += htons(ICMP_ECHO << 8);
216     }
217 #endif
218
219

```

图 1.3.7.4 修改 icmp.c 以支持硬件帧校验

1.3.8 LWIP 的裁剪与配置

在 LWIP 的源码中有个 opt.h 的文件，这个文件是裁剪和配置 LWIP 的，不过我们最好不要直接在 opt.h 里面做修改，我们可以打开 opt.h 文件看一下，里面的配置都是条件编译的，如果我们在其他地方有定义过的话那么在 opt.h 里面的定义不起作用了。所以我们可以新建一个 .h 的文件来裁剪和配置 LWIP，我们前面提过在 LWIP->lwip_app->lwip_comm 下有一个 lwipopts.h 的文件，这个文件就是用来裁剪与配置 lwipopts.h 的，lwipopts.h 配置代码如下。

```

#ifndef __LWIPOPTS_H__

```

```

#define __LWIPOPTS_H__

#define SYS_LIGHTWEIGHT_PROT    0
//NO_SYS==1:不使用操作系统
#define NO_SYS                    1 //不使用 UCOS 操作系统
//使用 4 字节对齐模式
#define MEM_ALIGNMENT            4

//MEM_SIZE:heap 内存的大小,如果在应用中有大量数据发送的话这个值最好设置大一点
#define MEM_SIZE                  16000 //内存堆大小

//MEMP_NUM_PBUF:memp 结构的 pbuf 数量,如果应用从 ROM 或者静态存储区发送大量数据
//时,这个值应该设置大一点
#define MEMP_NUM_PBUF            10

//MEMP_NUM_UDP_PCB:UDP 协议控制块(PCB)数量.每个活动的 UDP"连接"需要一个 PCB.
#define MEMP_NUM_UDP_PCB        6

//MEMP_NUM_TCP_PCB:同时建立激活的 TCP 数量
#define MEMP_NUM_TCP_PCB        10

//MEMP_NUM_TCP_PCB_LISTEN:能够监听的 TCP 连接数量
#define MEMP_NUM_TCP_PCB_LISTEN 6

//MEMP_NUM_TCP_SEG:最多同时在队列中的 TCP 段数量
#define MEMP_NUM_TCP_SEG        15

//MEMP_NUM_SYS_TIMEOUT:能够同时激活的 timeout 个数
#define MEMP_NUM_SYS_TIMEOUT     8

/* ----- Pbuf 选项----- */
//PBUF_POOL_SIZE:pbuf 内存池个数.
#define PBUF_POOL_SIZE           20

//PBUF_POOL_BUFSIZE:每个 pbuf 内存池大小.
#define PBUF_POOL_BUFSIZE        512

/* ----- TCP 选项----- */
#define LWIP_TCP                  1 //为 1 是使用 TCP
#define TCP_TTL                   255//生存时间

/*当 TCP 的数据段超出队列时的控制位,当设备的内存过小的时候此项应为 0*/
#define TCP_QUEUE_OOSEQ          0

```

```

//最大 TCP 分段
#define TCP_MSS          (1500 - 40)      //TCP_MSS = (MTU - IP 报头大小 - TCP 报头大小

//TCP 发送缓冲区大小(bytes).
#define TCP_SND_BUF      (4*TCP_MSS)

//TCP_SND_QUEUELEN: TCP 发送缓冲区大小(pbuf).这个值最小为
//(2 * TCP_SND_BUF/TCP_MSS)
#define TCP_SND_QUEUELEN      (2* TCP_SND_BUF/TCP_MSS)

//TCP 发送窗口
#define TCP_WND            (2*TCP_MSS)

/* ----- ICMP 选项----- */
#define LWIP_ICMP          1 //使用 ICMP 协议

/* ----- DHCP 选项----- */
//当使用 DHCP 时此位应该为 1,LwIP 0.5.1 版本中没有 DHCP 服务.
#define LWIP_DHCP          1

/* ----- UDP 选项 ----- */
#define LWIP_UDP            1 //使用 UDP 服务
#define UDP_TTL            255 //UDP 数据包生存时间

/* ----- Statistics options ----- */
#define LWIP_STATS 0
#define LWIP_PROVIDE_ERRNO 1

//STM32F4x7 允许通过硬件识别和计算 IP,UDP 和 ICMP 的帧校验和
#define CHECKSUM_BY_HARDWARE //定义 CHECKSUM_BY_HARDWARE,使用硬件帧校验

#ifndef CHECKSUM_BY_HARDWARE
//CHECKSUM_GEN_IP==0: 硬件生成 IP 数据包的帧校验和
#define CHECKSUM_GEN_IP      0
//CHECKSUM_GEN_UDP==0: 硬件生成 UDP 数据包的帧校验和
#define CHECKSUM_GEN_UDP      0
//CHECKSUM_GEN_TCP==0: 硬件生成 TCP 数据包的帧校验和
#define CHECKSUM_GEN_TCP      0
//CHECKSUM_CHECK_IP==0: 硬件检查输入的 IP 数据包帧校验和
#define CHECKSUM_CHECK_IP      0
//CHECKSUM_CHECK_UDP==0: 硬件检查输入的 UDP 数据包帧校验和
#define CHECKSUM_CHECK_UDP      0

```

```
//CHECKSUM_CHECK_TCP==0: 硬件检查输入的 TCP 数据包帧校验和
#define CHECKSUM_CHECK_TCP                0
#else
//CHECKSUM_GEN_IP==1: 软件生成 IP 数据包帧校验和
#define CHECKSUM_GEN_IP                    1
//CHECKSUM_GEN_UDP==1: 软件生成 UDOP 数据包帧校验和
#define CHECKSUM_GEN_UDP                    1
//CHECKSUM_GEN_TCP==1: 软件生成 TCP 数据包帧校验和
#define CHECKSUM_GEN_TCP                    1
//CHECKSUM_CHECK_IP==1: 软件检查输入的 IP 数据包帧校验和
#define CHECKSUM_CHECK_IP                  1
//CHECKSUM_CHECK_UDP==1: 软件检查输入的 UDP 数据包帧校验和
#define CHECKSUM_CHECK_UDP                  1
//CHECKSUM_CHECK_TCP==1: 软件检查输入的 TCP 数据包帧校验和
#define CHECKSUM_CHECK_TCP                  1
#endif

//LWIP_NETCONN==1:使能 NETCON 函数(要求使用 api_lib.c)
#define LWIP_NETCONN                        0

//LWIP_SOCKET==1:使能 Socket API(要求使用 sockets.c)
#define LWIP_SOCKET                        0

#define LWIP_COMPAT_MUTEX                    1
#define LWIP_SO_RCVTIMEO                    1 //通过定义 LWIP_SO_RCVTIMEO 使能
netconn 结构体中 recv_timeout,使用 recv_timeout 可以避免阻塞线程

//#define LWIP_DEBUG                        1 //开启 DEBUG 选项
#define ICMP_DEBUG                          LWIP_DBG_OFF //开启/关闭 ICMPdebug

#endif
```

我们可以看到 lwipopts.h 中有很多的宏定义，每个宏定义后面已经给出了具体的解释，大家可以参考一下，这里我们只是给出了一个参考配置，大家在使用的过程中一定要按照自己的需求来编写 lwipopts.h 里面的内容

1.4 软件设计

经过上面几节的讲解，LWIP 移植部分已经完成，在本节我们可以编写 mian.c 文件来测试移植是否成功，在 main.c 文件中我们有两个函数 show_address()和 main()函数，show_address()函数用来在 LCD 上显示一些提示信息，比如 MAC 地址、IP 地址、子网掩码、默认网关等信息。

我们重点讲解一下 main 函数，main 函数代码如下。

```
int main(void)
{
    delay_init();           //延时初始化
```

```

NVIC_Configuration();    //中断分组配置
uart_init(115200);       //串口波特率设置
usmart_dev.init(84);     //初始化 USMART
LED_Init();              //LED 初始化
KEY_Init();              //按键初始化
LCD_Init();              //LCD 初始化
FSMC_SRAM_Init();        //初始化外部 SRAM

my_mem_init(SRAMIN);     //初始化内部内存池
my_mem_init(SRAMEX);     //初始化外部内存池
my_mem_init(SRAMCCM);    //初始化 CCM 内存池

POINT_COLOR = RED;
LCD_ShowString(30,30,200,16,16,"Explorer STM32F4");
LCD_ShowString(30,50,200,16,16,"Ethernet lwIP Test");
LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
LCD_ShowString(30,90,200,16,16,"2014/8/15");
TIM3_Int_Init(999,839); //100khz 的频率,计数 1000 为 10ms
while(lwip_comm_init()) //lwip 初始化
{
    LCD_ShowString(30,110,200,20,16,"LWIP Init Falied!");
    delay_ms(1200);
    LCD_Fill(30,110,230,130,WHITE); //清除显示
    LCD_ShowString(30,110,200,16,16,"Retrying...");
}
LCD_ShowString(30,110,200,20,16,"LWIP Init Success!");
LCD_ShowString(30,130,200,16,16,"DHCP IP configing..."); //等待 DHCP 获取
#ifdef LWIP_DHCP //使用 DHCP
    //等待 DHCP 获取成功/超时溢出
    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0XFF))
    {
        lwip_periodic_handle(); //LWIP 内核需要定时处理的函数
    }
#endif
    show_address(lwipdev.dhcpstatus); //显示地址信息
    while(1)
    {
        lwip_periodic_handle(); //LWIP 内核需要定时处理的函数
    }
}

```

在 main 函数中首先完成对外设的初始化，然后调用 lwip_comm_init 函数初始化 LWIP，如果使用 DHCP 的话就先等待 DHCP 获取 IP 地址完成，然后在 LCD 上显示地址信息，如果 DHCP 获取地址失败的话将使用我们的默认的地址，最后我们在一个 while 循环中循环调用

lwip_periodic_handle()函数。在 main 函数中我们要注意到不管是在等待 DHCP 完成还是 DHCP 成功后我们都要周期性调用 lwip_periodic_handle()函数，因为在这个函数中周期性调用协议栈内核的一些定时函数以满足 LWIP 的内核要求。

我们前面说过在本章中添加 USMART 组件，在这里我们就可以使用 USMART 组件，我们通过 USMART 组件就可以读取或者改写 LAN8720 内部寄存器的配置，这是非常好的一种调试网络的方法。这里我们要用到 stm32f4x7_eth.c 中的读写 PHY 芯片的函数：ETH_ReadPHYRegister()和 ETH_WritePHYRegister(),我们将这两个函数添加到 usmart_config.c 文件中的 usmart_nametab[]数组中，如图 1.4.1 所示，注意一定要把 stm32f4x7_eth.h 的头文件也要添加进来。

```

1 #include "usmart.h"
2 #include "usmart_str.h"
3 #include "stm32f4x7_eth.h"
4 //下面要包含所用到的函数所声明的头文件(用户自己添加)
5 //用户直接在这里输入要执行的函数名及其查找串
6 #include "delay.h"
7 #include "usart.h"
8 #include "sys.h"
9
10 //函数名列表初始化(用户自己添加)
11 //用户直接在这里输入要执行的函数名及其查找串
12 struct _m_usmart_nametab usmart_nametab[] =
13 {
14     #if USMART_USE_WRFUNS==1 //如果使能了读写操作
15     (void*)read_addr, "u32 read_addr(u32 addr)",
16     (void*)write_addr, "void write_addr(u32 addr, u32 val)",
17     #endif
18     (void*)delay_ms, "void delay_ms(u16 nms)",
19     (void*)delay_us, "void delay_us(u32 nus)",
20     (void*)ETH_ReadPHYRegister, "uint16_t ETH_ReadPHYRegister(uint16_t PHYAddress, uint16_t PHYReg)",
21     (void*)ETH_WritePHYRegister, "uint32_t ETH_WritePHYRegister(uint16_t PHYAddress, uint16_t PHYReg, uint16_t PHYValue)",
22 };
23
24

```

图 1.4.1 添加读写 PHY 函数(usmart_config.c 文件中)

注意，我们编译以后可能会提示如图 1.4.2 所示的两个 Warning，提示我们 tcphdr 定义了但是没有使用，这两个 Warning 我们不用管他。

```

compiling tcp_out.c...
..\\LWIP\\lwip-1.4.1\\src\\core\\tcp_out.c(845): warning: #550-D: variable "tcphdr" was set but never used
    struct tcp_hdr *tcphdr;
..\\LWIP\\lwip-1.4.1\\src\\core\\tcp_out.c(1367): warning: #550-D: variable "tcphdr" was set but never used
    struct tcp_hdr *tcphdr;
..\\LWIP\\lwip-1.4.1\\src\\core\\tcp_out.c: 2 warnings, 0 errors
compiling timers.c...

```

图 1.4.2 Warning 提示

1.5 下载验证

1) 连接设置

在代码编译成功以后，我们下载代码到开发板中，通过网线连接开发板到路由器上，如果没有路由器的话也可以直接连接到电脑的 RJ45 接口上，由于 LAN8720 具有自动翻转功能，所以连接电脑 RJ45 时就不需要更换网线。但是如果连接到电脑的 RJ45 接口上那么开发板就不能使用 DHCP 功能，需要使用静态地址，我们例程中的默认静态 IP 地址：192.168.1.30，子网掩码：255.255.255.0，默认网关：192.168.1.1。连接上电脑端的 RJ45 以后我们还需要更改一下电脑的网络设置，步骤如下，此处以 WIN7 系统为例。

1、打开控制面板中的“网络和共享中心”，如图 1.5.1 所示。



图 1.5.1 网络和共享中心

2、打开“网络和共享中心”以后点击右侧的“更改适配器设置”，如图 1.5.2 所示。



图 1.5.2 更改适配器设置

3、打开“更改适配器设置”后，点击“本地连接”，如图 1.5.3 所示。



图 1.5.3 本地连接

4、打开“本地连接”后，出现如图 1.5.4 所示对话框，点击“属性”。

5、点击“属性”以后出现如图 1.5.5 所示对话框，选中“Internet 协议版本 4(TCP/IPv4)”并点击“属性”。

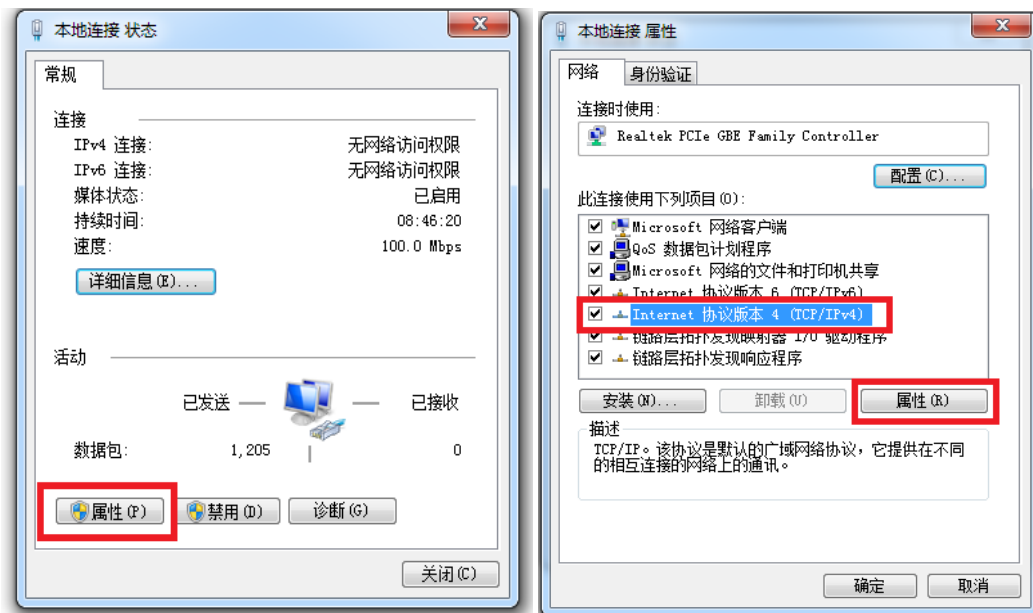


图 1.5.4 本地连接对话框图 1.5.5 本地连接属性对话框

6、点击“属性”后出现如图 1.5.6 所示对话框，选择“使用下面的 IP 地址”和“使用下面的 DNS 服务器地址”。在 IP 地址栏填入：192.168.1.x(x 为 2-254)，子网掩码：255.255.255.0 默认网关：192.168.1.1，首选 DNS 服务器：192.168.1.1，最后记得点击“确定”。注意！电脑的 IP 地址一定要和开发板的 IP 地址在一个网络内！以后我们的实验都是连接到路由器上的，如果没有路由器的话都可以使用上面这种方法这种方法完成实验，只是不能使用 DHCP 服务。

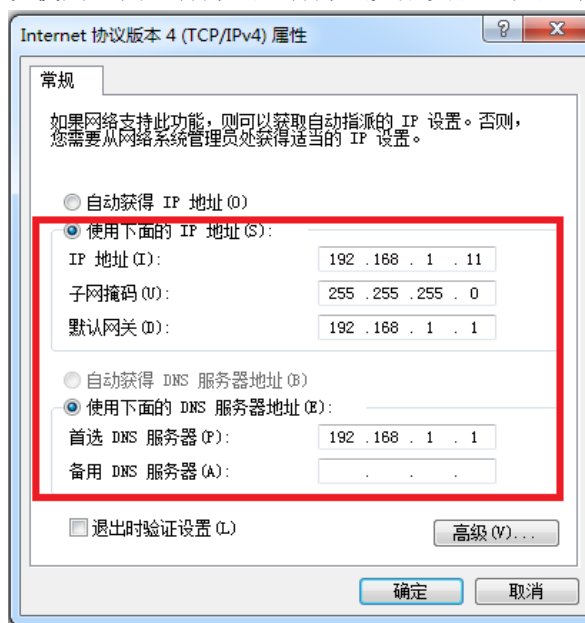


图 1.5.6 Inter 协议版本 4(TCP/IPv4)属性对话框

2) 验证测试

打开串口调试助手，复位一下开发板。这里我们开启了 DHCP，大家也可以自行尝试一下关闭 DHCP 使用静态 IP 地址，下载完成后 LCD 显示如图 1.5.7 所示。

注意：如果开发板是和电脑的 RJ45 相连，而且开启了 DHCP，开发板就会等待 DHCP 完

成，这个时候由于电脑没有 DHCP 服务因此会等待很久，直到 DHCP 超时使用默认地址。

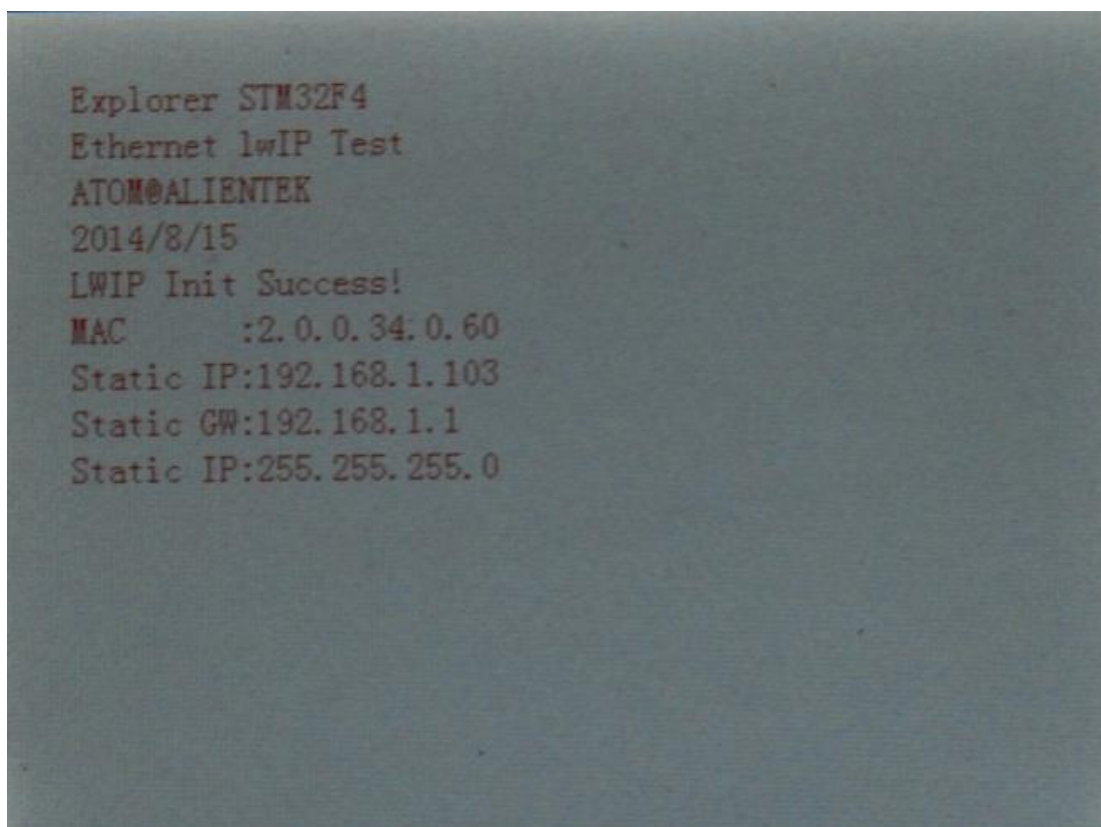


图 1.5.7 LCD 显示页面

在串口调试助手上显示如图 1.5.8 所示，从图 1.5.8 中可以看出和 LCD 上显示的地址信息是一致的。

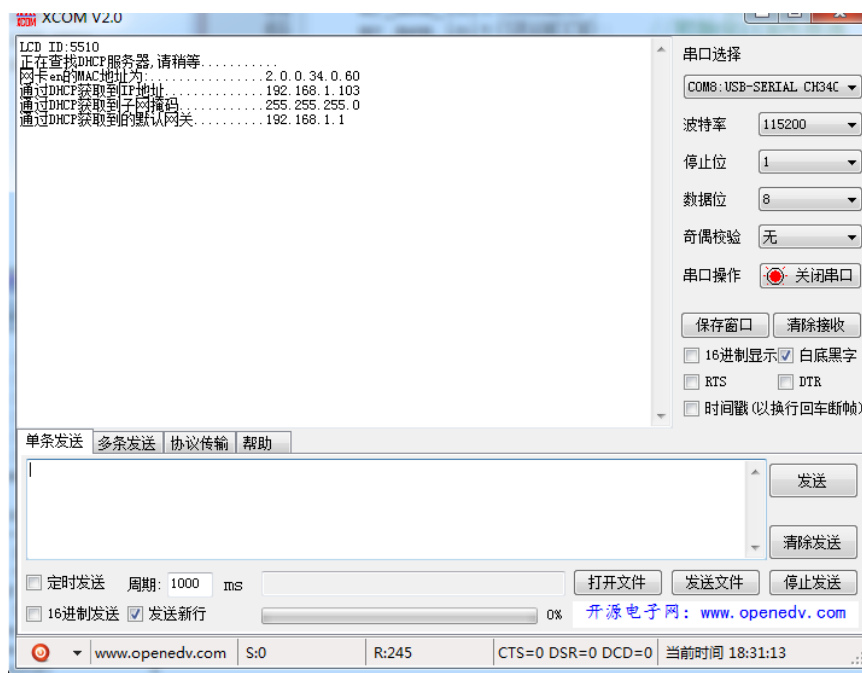



图 1.5.8 串口调试助手显示

可以看到此时通过路由器的 DHCP 分配到的 IP 地址为：192.168.1.118，默认网关为：

192.168.1.1, 子网掩码为: 255.255.255.0。在电脑上 ping 开发板的 IP 地址, 结果如图 1.5.9 所示。



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>ping 192.168.1.103

正在 Ping 192.168.1.103 具有 32 字节的数据:
来自 192.168.1.103 的回复: 字节=32 时间=3ms TTL=255
来自 192.168.1.103 的回复: 字节=32 时间=16ms TTL=255
来自 192.168.1.103 的回复: 字节=32 时间=1ms TTL=255
来自 192.168.1.103 的回复: 字节=32 时间=1ms TTL=255

192.168.1.103 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 1ms, 最长 = 16ms, 平均 = 5ms

C:\Users\Administrator>
C:\Users\Administrator>
```

图 1.5.9 ping 测试

我们可以使用 USMART 组件调试 LAN8720, 前面我们已经配置好了 USMART, 打开串口调试助手 (这里以我们 ALIENTEK 的 XCOM 串口调试助手为例) 打印出 USMART 支持的函数, 如图 1.5.10 所示, 从图中可以看出函数清单有 6 个函数, 其中就有我们前面添加过的 ETH_ReadPHYRegister() 和 ETH_WritePHYRegister() 这两个函数。

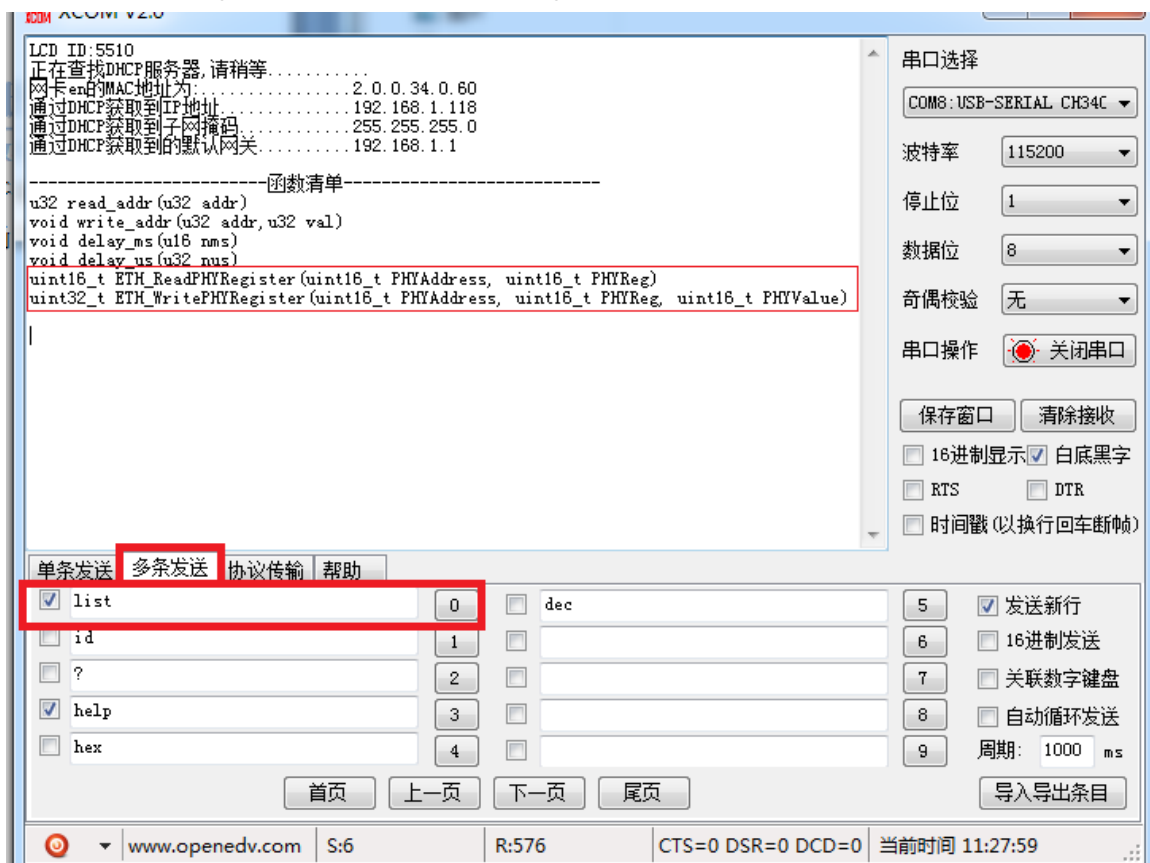


图 1.5.10 USMART 函数清单列表

我们在这里读取一下 LAN8720 的 BSR 寄存器(1)，可以查看一下当前网络的连接状态，我们使用 ETH_ReadPHYRegister()函数测试，ETH_ReadPHYRegister()有两个参数 PHYAddress 和 PHYReg，这两个参数分别为 PHY 地址和需要读取的寄存器编号，我们 STM32F407 开板的 PHY 地址为 0X00，这里我们读取 LAN8720 的寄存器 31 也就是特殊功能寄存器，所以串口调试助手发送框输入 ETH_ReadPHYRegister(0x00,31)，点击发送，我们就可以看到开发板返回给串口调试助手的信息，也就是寄存器 31 的值，如图 1.5.11 所示，我们可以看出此时 LAN8720 的值为 0X1058。

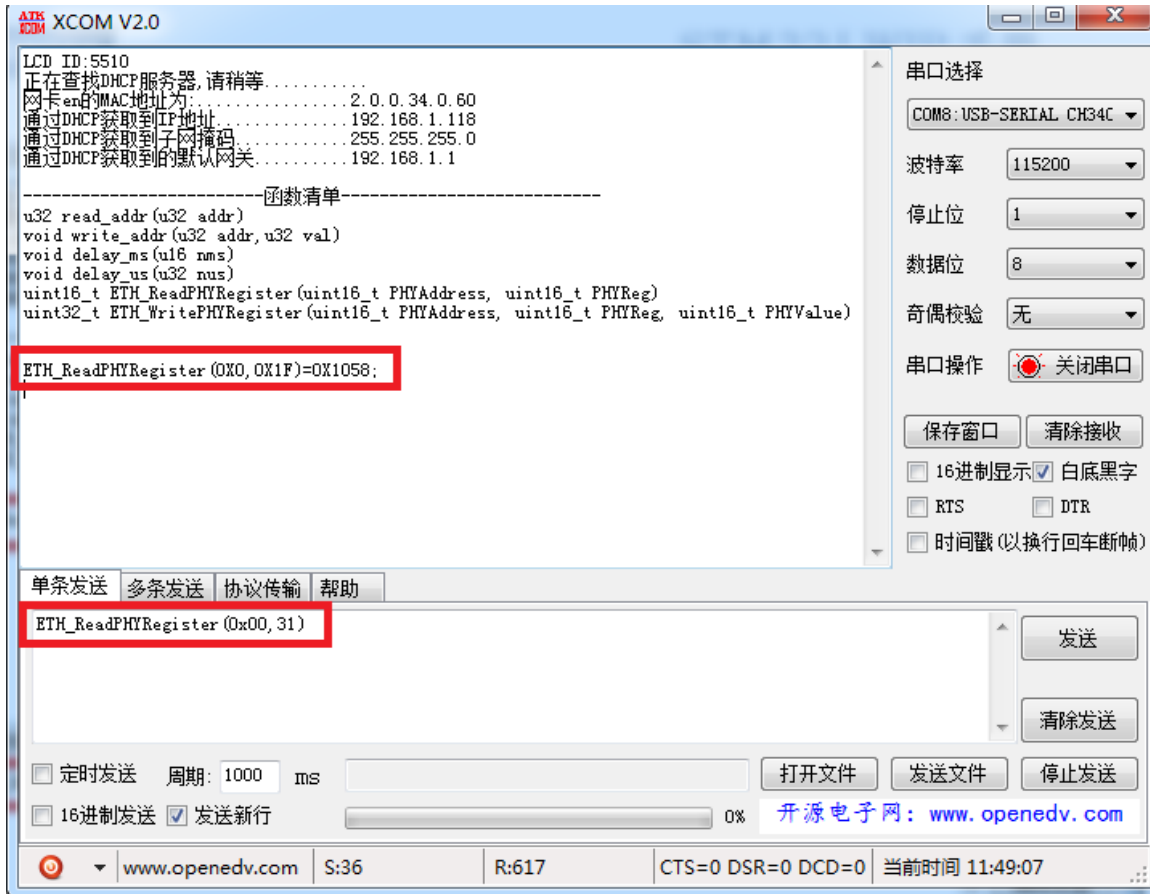


图 1.5.11 读取 LAN8720 寄存器 31

0X1058 的二进制为 0001000001011000，其中 bit2-4 为 110，通过查阅 LAN8720 手册，我们知道寄存器 31 的 bit2-4 代表着网络的连接速度和双工状态，可以看出“110”表示 100M 全双工连接，bit12 为 1 表示自动协商完成。我们也可以查看 LAN8720 的其他寄存器，也可以使用 ETH_WritePHYRegister()函数改写某些寄存器的值。

至此 LWIP 的无操作系统移植完成。

第二章 LWIP 带操作系统移植

LWIP 是支持操作系统的，在操作系统的支持下我们可以使用 LWIP 提供的另外两种 API 编程。没有操作的时候我们只能使用 raw 编程，相较于其他两种 API，raw 编程难度很大，需要用户对 LWIP 协议栈有一定的了解。而且使用操作系统以后我们可以多任务运行，将 LWIP 作为任务来运行。因此掌握基于操作系统的 LWIP 使用非常重要，本章就讲解带有 UCOS 的 LWIP 移植。

2.1 移植简介

2.2 软件设计

2.3 下载验证

2.1 移植简介

本章的移植是在前面无操作系统的基础上修改的，本章不讲解 UCOS 的移植，关于 UCOS 的移植请大家参考我们的《ALIENTEK STM32F4 UCOS 使用教程》。我们只需要对 lwipopts.h 和 lwip_comm.c 文件做简单修改，然后完成 sys_arch.h 和 sys_arch.c 这两个文件的编写。sys_arch.c 主要为协议栈提供邮箱、信号量等机制，在 LWIP 的 sys.h(这里为了避免与我们 SYSTEM 文件中的 sys.h 重名改为了 lwip_sys.h)中声明了这些需要实现的函数，在 sys_arch.c 文件中我们还需要实现的宏定义和函数如表 2.1.1 所示。

表 2.1.1 需要实现的宏和函数

名称	功能	所在文件
sys_sem_t	指针，指向信号量	sys_arch.h
sys_mutex_t	指针，指向互斥信号量	sys_arch.h
sys_mbox_t	指针，指向消息邮箱	sys_arch.h
sys_thread_t	任务 ID	sys_arch.h
sys_mbox_new	创建消息邮箱	sys_arch.c
sys_mbox_free	删除一个邮箱	sys_arch.c
sys_mbox_post	向邮箱投递消息，阻塞	sys_arch.c
sys_mbox_trypost	尝试向邮箱投递消息，不阻塞	sys_arch.c
sys_arch_mbox_fetch	获取消息，阻塞	sys_arch.c
sys_arch_mbox_tryfetch	尝试获取消息，不阻塞	sys_arch.c
sys_mbox_valid	检查一个邮箱是否有效	sys_arch.c
sys_mbox_set_invalid	设置一个邮箱无效	sys_arch.c
sys_sem_new	创建一个信号量	sys_arch.c
sys_arch_sem_wait	等待一个信号量	sys_arch.c
sys_sem_signal	释放一个信号量	sys_arch.c
sys_sem_free	删除一个信号量	sys_arch.c
sys_sem_valid	查询一个信号量是否有效	sys_arch.c
sys_sem_set_invalid	设置一个信号量无效	sys_arch.c
sys_thread_new	创建进程	sys_arch.c
sys_init	初始化操作系统模拟层	sys_arch.c
sys_msleep	LWIP 延时函数	sys_arch.c
sys_now	获取当前系统时间	sys_arch.c

从表 2.1.1 中可以看出，sys_arch.c 中要实现的主要是与消息邮箱、信号量和创建进程有关的函数，上层 API 与协议栈内核的数据交互就是通过消息邮箱和信号量来完成的。在 UCOS 中提供了一整套的邮箱、信号量等机制，我们只需要对这些函数做简单的封装即可。最后，协议栈在初始化的时候会调用 sys_thread_new 函数创建一个进程，因此这个函数必须实现，实现过程也是对 UCOS 中的进程创建函数做简单的封装。

2.2 带操作系统 LWIP 移植

2.2.1 UCOSII+LWIP 移植

1)修改 cc.h 文件

在 LWIP 中支持针对关键代码的保护，比如申请内存等，而我们知道在 UCOS II 有临界区保护，因此我们就可以使用 UCOS II 中的临界区保护函数。在 cc.h 文件中我们使用了宏定义来实现这一功能，其实讲解 LWIP 无操作系统移植的时候我们在 cc.h 文件中就已经添加了这段代码，代码如下所示。

```
//使用操作系统时的临界区保护，这里以 UCOS II 为例
//当定义了 OS_CRITICAL_METHOD 时就说明使用了 UCOS II
#if OS_CRITICAL_METHOD == 1
#define SYS_ARCH_DECL_PROTECT(lev)
#define SYS_ARCH_PROTECT(lev)    CPU_INT_DIS()
#define SYS_ARCH_UNPROTECT(lev)   CPU_INT_EN()
#endif

#if OS_CRITICAL_METHOD == 3
#define SYS_ARCH_DECL_PROTECT(lev)  u32_t lev
//UCOS II 中进入临界区,关中断
#define SYS_ARCH_PROTECT(lev)      lev = OS_CPU_SR_Save()
//UCOS II 中退出 A 临界区，开中断
#define SYS_ARCH_UNPROTECT(lev)     OS_CPU_SR_Restore(lev)
#endif
```

我们还需要在 cc.h 中添加如图 2.2.1.1 所示的 includes.h 头文件路径。

```
1 // cc.h属于LWIP TCP/IP协议栈一部分
2 // 作者: Adam Dunkels <adam@sics.se>
3
4 #ifndef __CC_H__
5 #define __CC_H__
6
7 #include "cpu.h"
8 #include "stdio.h"
9 #include "includes.h" //使用UCOS 要添加此头文件!
10
```

图 2.2.1.1 添加 includes.h 头文件(cc.h 文件中)

2)修改 lwipopts.h 头文件

我们说过 lwipopts.h 是用来裁剪和配置 LWIP 的，那么我们要使用操作系统的话就需要对其进行相应的配置，在这里与无操作系统 LWIP 中的 lwipopts.h 中的配置有点不同，本章中的配置如下。

```
#ifndef __LWIPOPTS_H__
#define __LWIPOPTS_H__

//线程优先级
#ifndef TCPIP_THREAD_PRIO
```

```

#define TCPIP_THREAD_PRIO      5    //定义内核任务的优先级为 5
#endif
#undef  DEFAULT_THREAD_PRIO
#define DEFAULT_THREAD_PRIO    2

//为 1 时使用实时操作系统的轻量级保护,保护关键代码不被中断打断
#define SYS_LIGHTWEIGHT_PROT    1
#define NO_SYS                  0    //使用 UCOS 操作系统
#define MEM_ALIGNMENT          4    //使用 4 字节对齐模式
#define MEM_SIZE                16000 //内存堆 heap 大小
//MEMP_NUM_PBUF:memp 结构的 pbuf 数量,如果应用从 ROM
//或者静态存储区发送大量数据时,这个值应该设置大一点
#define MEMP_NUM_PBUF          20
//MEMP_NUM_UDP_PCB:UDP 协议控制块(PCB)数量.每个活动的 UDP"连接"需要一个 PCB
#define MEMP_NUM_UDP_PCB       6
//MEMP_NUM_TCP_PCB:同时建立激活的 TCP 数量
#define MEMP_NUM_TCP_PCB       10
//MEMP_NUM_TCP_PCB_LISTEN:能够监听的 TCP 连接数量
#define MEMP_NUM_TCP_PCB_LISTEN 6
//MEMP_NUM_TCP_SEG:最多同时在队列中的 TCP 段数量
#define MEMP_NUM_TCP_SEG       15
//MEMP_NUM_SYS_TIMEOUT:能够同时激活的 timeout 个数
#define MEMP_NUM_SYS_TIMEOUT    8

//PBUF_POOL_SIZE:pbuf 内存池个数
#define PBUF_POOL_SIZE          20
//PBUF_POOL_BUFSIZE:每个 pbuf 内存池大小
#define PBUF_POOL_BUFSIZE       512
#define LWIP_TCP                 1    //使用 TCP
#define TCP_TTL                  255   //生存时间

#undef TCP_QUEUE_OOSEQ
//当 TCP 的数据段超出队列时的控制位,当设备的内存过小的时候此项应为 0
#define TCP_QUEUE_OOSEQ         0

#undef TCPIP_MBOX_SIZE
//tcpip 创建主线程时的消息邮箱大小
#define TCPIP_MBOX_SIZE         MAX_QUEUE_ENTRIES

#undef DEFAULT_TCP_RECVMBOX_SIZE
#define DEFAULT_TCP_RECVMBOX_SIZE MAX_QUEUE_ENTRIES

#undef DEFAULT_ACCEPTMBOX_SIZE

```

```

#define DEFAULT_ACCEPTMBOX_SIZE          MAX_QUEUE_ENTRIES

//最大 TCP 分段,TCP_MSS = (MTU - IP 报头大小 - TCP 报头大小)
#define TCP_MSS                          (1500 - 40)
#define TCP_SND_BUF                      (4*TCP_MSS)    //TCP 发送缓冲区大小(bytes)
//TCP_SND_QUEUELEN: TCP 发送缓冲区大小(pbuf).这个值
//最小为(2 * TCP_SND_BUF/TCP_MSS)
#define TCP_SND_QUEUELEN                (2* TCP_SND_BUF/TCP_MSS)
#define TCP_WND                          (2*TCP_MSS)    //TCP 发送窗口
#define LWIP_ICMP                        1              //使用 ICMP 协议
#define LWIP_DHCP                        1              //使用 DHCP
#define LWIP_UDP                        1              //使用 UDP 服务
#define UDP_TTL                          255            //UDP 数据包生存时间
#define LWIP_STATS 0
#define LWIP_PROVIDE_ERRNO 1

//帧校验和选项, STM32F4x7 允许通过硬件识别和计算 IP,UDP 和 ICMP 的帧校验和
#define CHECKSUM_BY_HARDWARE //定义 CHECKSUM_BY_HARDWARE,使用硬件帧校验
#ifndef CHECKSUM_BY_HARDWARE
    //CHECKSUM_GEN_IP==0: 硬件生成 IP 数据包的帧校验和
    #define CHECKSUM_GEN_IP              0
    //CHECKSUM_GEN_UDP==0: 硬件生成 UDP 数据包的帧校验和
    #define CHECKSUM_GEN_UDP              0
    //CHECKSUM_GEN_TCP==0: 硬件生成 TCP 数据包的帧校验和
    #define CHECKSUM_GEN_TCP              0
    //CHECKSUM_CHECK_IP==0: 硬件检查输入的 IP 数据包帧校验和
    #define CHECKSUM_CHECK_IP              0
    //CHECKSUM_CHECK_UDP==0: 硬件检查输入的 UDP 数据包帧校验和
    #define CHECKSUM_CHECK_UDP              0
    //CHECKSUM_CHECK_TCP==0: 硬件检查输入的 TCP 数据包帧校验和
    #define CHECKSUM_CHECK_TCP              0
#else
    //CHECKSUM_GEN_IP==1: 软件生成 IP 数据包帧校验和
    #define CHECKSUM_GEN_IP              1
    //CHECKSUM_GEN_UDP==1: 软件生成 UDP 数据包帧校验和
    #define CHECKSUM_GEN_UDP              1
    //CHECKSUM_GEN_TCP==1: 软件生成 TCP 数据包帧校验和
    #define CHECKSUM_GEN_TCP              1
    //CHECKSUM_CHECK_IP==1: 软件检查输入的 IP 数据包帧校验和
    #define CHECKSUM_CHECK_IP              1
    //CHECKSUM_CHECK_UDP==1: 软件检查输入的 UDP 数据包帧校验和

```

```

#define CHECKSUM_CHECK_UDP          1
//CHECKSUM_CHECK_TCP==1: 软件检查输入的 TCP 数据包帧校验和
#define CHECKSUM_CHECK_TCP          1
#endif

#define LWIP_NETCONN      1 //LWIP_NETCONN==1:使能 NETCON 函数(要求使用 api_lib.c)
#define LWIP_SOCKET      1 //LWIP_SOCKET==1:使能 Sicket API(要求使用 sockets.c)
#define LWIP_COMPAT_MUTEX      1
//通过定义 LWIP_SO_RCVTIMEO 使能 netconn 结构体中 recv_timeout
//使用 recv_timeout 可以避免阻塞线程
#define LWIP_SO_RCVTIMEO      1

#define TCPIP_THREAD_STACKSIZE      1000 //内核任务堆栈大小
#define DEFAULT_UDP_RECVMBOX_SIZE    2000
#define DEFAULT_THREAD_STACKSIZE     512

//LWIP 调试选项
#define LWIP_DEBUG      0 //关闭 DEBUG 选项
#define ICMP_DEBUG      LWIP_DBG_OFF //开启/关闭 ICMPdebug
#endif

```

在上面 lwipopts.h 文件的定义中我们要注意一下几点。

1、因为我们要使用 UCOS II 系统，所以 NO_SYS 定义为 0。

2、使用操作系统时我们要对 LWIP 中关键代码做保护，因此要定义 SYS_LIGHTWEIGHT_PROT 为 1。

3、如果使用 NETCONN 编程方式的话就定义 LWIP_NETCONN 为 1。

4、如果使用 SOCKET 编程方式的话就定义 LWIP_SOCKET 为 1。

上面只是我们列举的一小部分配置，使用 UCOS 的话还需要很多的配置，具体配置大家参考我们的 lwipopts.h 文件。这里就不一一解释了。

3) 修改 sys_arch.h 头文件

在 sys_arch.h 中定义了消息邮箱的数量和每个消息邮箱的大小，在这个文件中还定义了 4 中数据类型：sys_sem_t、sys_mutex_t、sys_mbox_t 和 sys_thread_t。分别为信号量、互斥信号量、消息邮箱和线程 ID。除了 sys_mbox_t 外其他三个很好理解，sys_mbox_t 的定义比较麻烦，这里不能直接使用 UCOS 的消息邮箱，使用的是消息队列，移植的时候我们需要自己定义在 LWIP 中使用的邮箱类型结构体，sys_arch.h 文件代码如下。

```

#ifndef __ARCH_SYS_ARCH_H__
#define __ARCH_SYS_ARCH_H__
#include <includes.h>
#include "arch/cc.h"
#include "includes.h"

#ifdef SYS_ARCH_GLOBALS
#define SYS_ARCH_EXT
#else

```

```

#define SYS_ARCH_EXT extern
#endif

#define MAX_QUEUES                10 // 消息邮箱的数量
#define MAX_QUEUE_ENTRIES        20 // 每个消息邮箱的大小
//LWIP 消息邮箱结构体
typedef struct {
    OS_EVENT*   pQ;    //UCOS 中指向事件控制块的指针
    //消息队列, MAX_QUEUE_ENTRIES 消息队列中最多消息数
    void*       pvQEntries[MAX_QUEUE_ENTRIES];
} TQ_DESCR, *PQ_DESCR;
typedef OS_EVENT *sys_sem_t;    //LWIP 使用的信号量
typedef OS_EVENT *sys_mutex_t; //LWIP 使用的互斥信号量
typedef PQ_DESCR sys_mbox_t;    //LWIP 使用的消息邮箱,其实就是 UCOS 中的消息队列
typedef INT8U sys_thread_t;     //线程 ID,也就是任务优先级
#endif

```

4) 修改 sys_arch.c 文件

sys_arch.c 文件是非常重要的一个文件,在这个文件中定义了 LWIP 使用到的关于信号量、消息邮箱的操作函数。

1、消息邮箱函数

与邮箱有关的函数要使用操作系统中的消息队列,有关于消息邮箱的这些函数功能来自于移植手册 sys_arch.txt,相关代码如下。

```

//创建一个消息邮箱
//*mbox:消息邮箱
//size:邮箱大小
//返回值:ERR_OK,创建成功
//      其他,创建失败
err_t sys_mbox_new( sys_mbox_t *mbox, int size)
{
    (*mbox)=mymalloc(SRAMIN,sizeof(TQ_DESCR)); //为消息邮箱申请内存
    mymemset((*mbox),0,sizeof(TQ_DESCR));      //清除 mbox 的内存
    if(*mbox)//内存分配成功
    {
        //消息队列最多容纳 MAX_QUEUE_ENTRIES 消息数目
        if(size>MAX_QUEUE_ENTRIES)size=MAX_QUEUE_ENTRIES;
        //使用 UCOS 创建一个消息队列
        (*mbox)->pQ=OSQCreate(&((*mbox)->pvQEntries[0]),size);
        LWIP_ASSERT("OSQCreate",(*mbox)->pQ!=NULL);
        //返回 ERR_OK,表示消息队列创建成功 ERR_OK=0
        if((*mbox)->pQ!=NULL)return ERR_OK;
        else
        {
            myfree(SRAMIN,(*mbox));

```

```

        return ERR_MEM;           //消息队列创建错误
    }
    }else return ERR_MEM;         //消息队列创建错误
}
//释放并删除一个消息邮箱
//*mbox:要删除的消息邮箱
void sys_mbox_free(sys_mbox_t * mbox)
{
    u8_t ucErr;
    sys_mbox_t m_box=*mbox;
    (void)OSQDel(m_box->pQ,OS_DEL_ALWAYS,&ucErr);
    LWIP_ASSERT( "OSQDel ",ucErr == OS_ERR_NONE );
    myfree(SRAMIN,m_box);
    *mbox=NULL;
}
//向消息邮箱中发送一条消息(必须发送成功)
//*mbox:消息邮箱
//*msg:要发送的消息
void sys_mbox_post(sys_mbox_t *mbox,void *msg)
{
    //当 msg 为空时 msg 等于 pvNullPointer 指向的值
    if(msg==NULL)msg=(void*)&pvNullPointer;
    while(OSQPost((*mbox)->pQ,msg)!=OS_ERR_NONE);//死循环等待消息发送成功
}
//尝试向一个消息邮箱发送消息
//此函数相对于 sys_mbox_post 函数只发送一次消息,
//发送失败后不会尝试第二次发送
//*mbox:消息邮箱
//*msg:要发送的消息
//返回值:ERR_OK,发送 OK
//      ERR_MEM,发送失败
err_t sys_mbox_trypost(sys_mbox_t *mbox, void *msg)
{
    //当 msg 为空时 msg 等于 pvNullPointer 指向的值
    if(msg==NULL)msg=(void*)&pvNullPointer;
    if((OSQPost((*mbox)->pQ, msg))!=OS_ERR_NONE)return ERR_MEM;
    return ERR_OK;
}

//等待邮箱中的消息
//*mbox:消息邮箱
//*msg:消息
//timeout:超时时间, 如果 timeout 为 0 的话,就一直等待

```

```

//返回值:当 timeout 不为 0 时如果成功的话就返回等待的时间,
//      失败的话就返回超时 SYS_ARCH_TIMEOUT,
u32_t sys_arch_mbox_fetch(sys_mbox_t *mbox, void **msg, u32_t timeout)
{
    u8_t ucErr;
    u32_t uc_timeout, timeout_new;
    void *temp;
    sys_mbox_t m_box=*mbox;
    if(timeout!=0)
    {
        //转换为节拍数,因为 UCOS 延时使用的是节拍数,而 LWIP 是用 ms
        uc_timeout=(timeout*OS_TICKS_PER_SEC)/1000;
        if(uc_timeout<1)uc_timeout=1;//至少 1 个节拍
    }else uc_timeout = 0;
    timeout = OSTimeGet(); //获取系统时间
    //请求消息队列,等待时限为 uc_timeout
    temp=OSQPend(m_box->pQ,(u16_t)uc_timeout,&ucErr);
    if(msg!=NULL)
    {
        //因为 lwip 发送空消息的时候我们使用了 pvNullPointer 指针,
        //所以判断 pvNullPointer 指向的值就可知道请求到的消息是否有效
        if(temp==(void*)&pvNullPointer)*msg = NULL;
        else *msg=temp;
    }
    if(ucErr==OS_ERR_TIMEOUT)timeout=SYS_ARCH_TIMEOUT; //请求超时
    else
    {
        LWIP_ASSERT("OSQPend ",ucErr==OS_ERR_NONE);
        timeout_new=OSTimeGet();
        //算出请求消息或使用的的时间
        if (timeout_new>timeout) timeout_new = timeout_new - timeout;
        else timeout_new = 0xffffffff - timeout + timeout_new;
        timeout=timeout_new*1000/OS_TICKS_PER_SEC + 1;
    }
    return timeout;
}

//尝试获取消息
//*mbox:消息邮箱
//*msg:消息
//返回值:等待消息所用的时间/SYS_ARCH_TIMEOUT
u32_t sys_arch_mbox_tryfetch(sys_mbox_t *mbox, void **msg)
{
    return sys_arch_mbox_fetch(mbox,msg,1);//尝试获取一个消息
}

```



```

}
//检查一个消息邮箱是否有效
//*mbox:消息邮箱
//返回值:1,有效.
//      0,无效
int sys_mbox_valid(sys_mbox_t *mbox)
{
    sys_mbox_t m_box=*mbox;
    u8_t ucErr;
    int ret;
    OS_Q_DATA q_data;
    memset(&q_data,0,sizeof(OS_Q_DATA));
    ucErr=OSQQuery (m_box->pQ,&q_data);
    ret=(ucErr<2&&(q_data.OSNMsgs<q_data.OSQSize))?1:0;
    return ret;
}
//设置一个消息邮箱为无效
//*mbox:消息邮箱
void sys_mbox_set_invalid(sys_mbox_t *mbox)
{
    *mbox=NULL;
}

```

这里我们对 `sys_arch_mbox_fetch()` 函数做一下简单的讲解，根据 LWIP 协议栈要求当 `sys_arch_mbox_fetch()` 函数的参数 `timeout` 不为 0 时需要我们返回等待消息所使用的时间(ms)，UCOS II 操作系统提供的 `OSQPend()` 函数并没有这个功能，因此需要我们自行实现。因为在 LWIP 中时间单位为 ms，而 UCOS II 中的时间单位为节拍数，因此在 `sys_arch_mbox_fetch()` 函数一开始我们先将输入的参数 `timeout` 转化为 UCOS 使用的节拍数。为了记录等待消息时用的时间，我们在调用 `OSQPend()` 函数之前记录下当前的系统节拍为 `timeout`，在等到消息以后记录下系统节拍 `timeout_new`。通过计算得到等待消息时所耗费的时间，当然结果是系统节拍数，因此最后要将这个系统节拍数转化为 ms，并返回该值。如果等待消息超时的话就直接返回 `SYS_ARCH_TIMEOUT`。

2、信号量相关函数

LWIP 要求的信号量功能和 UCOS 提供的信号量功能相似，因此我们只是对 UCOS 的信号量做了简单的封装而已，关于 LWIP 中信号量函数的功能来自于移植文档 `sys_arch.txt`，有关信号量代码如下。

```

//创建一个信号量
//*sem:创建的信号量
//count:信号量值
//返回值:ERR_OK,创建 OK
//      ERR_MEM,创建失败
err_t sys_sem_new(sys_sem_t * sem, u8_t count)
{
    u8_t err;

```

```

    *sem=OSSemCreate((u16_t)count);
    if(*sem==NULL)return ERR_MEM;
    OSEventNameSet(*sem,"LWIP Sem",&err);
    LWIP_ASSERT("OSSemCreate ",*sem != NULL);
    return ERR_OK;
}
//等待一个信号量
//*sem:要等待的信号量
//timeout:超时时间
//返回值:当 timeout 不为 0 时如果成功的话就返回等待的时间,
//      失败的话就返回超时 SYS_ARCH_TIMEOUT
u32_t sys_arch_sem_wait(sys_sem_t *sem, u32_t timeout)
{
    u8_t ucErr;
    u32_t ucos_timeout, timeout_new;
    if( timeout!=0)
    {
        //转换为节拍数,因为 UCOS 延时使用的是节拍数,而 LWIP 是用 ms
        ucos_timeout = (timeout * OS_TICKS_PER_SEC) / 1000;
        if(ucos_timeout < 1)
            ucos_timeout = 1;
        }else ucos_timeout = 0;
    timeout = OSTimeGet();
    OSSemPend (*sem,(u16_t)ucos_timeout, (u8_t *)&ucErr);
    if(ucErr == OS_ERR_TIMEOUT)timeout=SYS_ARCH_TIMEOUT;//请求超时
    else
    {
        timeout_new = OSTimeGet();
        if (timeout_new>=timeout) timeout_new = timeout_new - timeout;
        else timeout_new = 0xffffffff - timeout + timeout_new;
        //算出请求消息或使用的的时间(ms)
        timeout = (timeout_new*1000/OS_TICKS_PER_SEC + 1);
    }
    return timeout;
}
//发送一个信号量
//sem:信号量指针
void sys_arch_signal(sys_sem_t *sem)
{
    OSSemPost(*sem);
}
//释放并删除一个信号量
//sem:信号量指针

```

```

void sys_sem_free(sys_sem_t *sem)
{
    u8_t ucErr;
    (void)OSSemDel(*sem,OS_DEL_ALWAYS,&ucErr );
    if(ucErr!=OS_ERR_NONE)LWIP_ASSERT("OSSemDel ",ucErr==OS_ERR_NONE);
    *sem = NULL;
}
//查询一个信号量的状态,无效或有效
//sem:信号量指针
//返回值:1,有效.
//      0,无效
int sys_sem_valid(sys_sem_t *sem)
{
    OS_SEM_DATA  sem_data;
    return (OSSemQuery (*sem,&sem_data) == OS_ERR_NONE )? 1:0;
}
//设置一个信号量无效
//sem:信号量指针
void sys_sem_set_invalid(sys_sem_t *sem)
{
    *sem=NULL;
}

```

sys_arch_sem_wait() 函数为等待信号量函数，和 sys_arch_mbox_fetch() 函数一样，sys_arch_sem_wait() 函数也要返回待信号量所用的时间，这里我们的处理方法和 sys_arch_mbox_fetch() 函数相同，大家可以看一下对比的看一下 sys_arch_mbox_fetch() 函数的处理过程。

3、创建新进程

在有操作系统的支持下，LWIP 内核会用 sys_thread_new 函数来创建一个内核进程处理协议栈所有任务。当然这个创建进程的函数是对 UCOS 中创建新任务函数 OSTaskCreate() 的简单分装，注意我们对 sys_thread_new() 这个函数做了特殊处理，使得这个函数仅仅用于 LWIP 自身调用创建 tcpip 内核线程，如果要创建其他任务一定要使用 UCOS 的 OSTaskCreate() 函数！sys_thread_new 代码如下。

```

//创建一个新进程
//*name:进程名称
//thred:进程任务函数
//*arg:进程任务函数的参数
//stacksize:进程任务的堆栈大小
//prio:进程任务的优先级
sys_thread_t sys_thread_new(const char *name, lwip_thread_fn thread, void *arg, int stacksize, int prio)
{
    OS_CPU_SR cpu_sr;
    if(strcmp(name,TCPIP_THREAD_NAME)==0)//创建 TCP IP 内核任务

```

```

{
    OS_ENTER_CRITICAL(); //进入临界区
    //创建 TCP IP 内核任务
    OSTaskCreate(thread,arg,(OS_STK*)&TCPIP_THREAD_TASK_STK[stacksize-1],prio);
    OS_EXIT_CRITICAL(); //退出临界区
}
return 0;
}

```

最后我们还实现了 `sys_msleep` 和 `sys_now` 这两个函数，他们分别为 LWIP 使用到的延时函数和获取系统时间函数。`sys_msleep` 函数很简单，`sys_now` 函数是先获取到 UCOS 的时间节拍，然后将其转换为 LWIP 使用的 ms，并返回这个时间值。

5) 修改 `lwip_comm.c` 文件

`lwip_comm.c` 文件基本与前面介绍的不带操作系统相同，只有 `lwip_comm_init()` 函数和有关 DHCP 的 3 个函数不同，`lwip_comm_init()` 函数代码如下。

```

//LWIP 初始化(LWIP 启动的时候使用)
//返回值:0,成功
//      1,内存错误
//      2,LAN8720 初始化失败
//      3,网卡添加失败.
u8 lwip_comm_init(void)
{
    OS_CPU_SR cpu_sr;
    //调用 netif_add()函数时的返回值,用于判断网络初始化是否成功
    struct netif *Netif_Init_Flag;
    struct ip_addr ipaddr;           //ip 地址
    struct ip_addr netmask;          //子网掩码
    struct ip_addr gw;               //默认网关
    if(ETH_Mem_Malloc())return 1;    //内存申请失败
    if(lwip_comm_mem_malloc())return 1; //内存申请失败
    if(LAN8720_Init())return 2;      //初始化 LAN8720 失败
    //初始化 tcp ip 内核,该函数里面会创建 tcpip_thread 内核任务
    tcpip_init(NULL,NULL);
    lwip_comm_default_ip_set(&lwipdev); //设置默认 IP 等信息
    #if LWIP_DHCP //使用 DHCP
        ipaddr.addr = 0;
        netmask.addr = 0;
        gw.addr = 0;
    #else //使用静态 IP
        IP4_ADDR(&ipaddr,lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
        IP4_ADDR(&netmask,lwipdev.netmask[0],lwipdev.netmask[1],\
        lwipdev.netmask[2],lwipdev.netmask[3]);
        IP4_ADDR(&gw,lwipdev.gateway[0],lwipdev.gateway[1],\
        lwipdev.gateway[2],lwipdev.gateway[3]);
    #endif
}

```

```

printf(" 网 卡 的 MAC 地 址 为 :.....%d.%d.%d.%d.%d.%d\r\n",lwipdev.mac[0],\
lwipdev.mac[1],lwipdev.mac[2],lwipdev.mac[3],lwipdev.mac[4],lwipdev.mac[5]);
printf("    静    态    IP    地    址    .....%d.%d.%d.%d\r\n",\
lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
printf("    子    网    掩    码    .....%d.%d.%d.%d\r\n",lwipdev.netmask[0],\
lwipdev.netmask[1],lwipdev.netmask[2],lwipdev.netmask[3]);
printf("    默    认    网    关    .....%d.%d.%d.%d\r\n",lwipdev.gateway[0],\
lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);
#endif
OS_ENTER_CRITICAL(); //进入临界区
//向网卡列表中添加一个网口
Netif_Init_Flag=netif_add(&lwip_netif,&ipaddr,&netmask,&gw,NULL,\
&ethernetif_init,&tcpip_input);
OS_EXIT_CRITICAL(); //退出临界区
if(Netif_Init_Flag==NULL)return 3;//网卡添加失败
else//网口添加成功后,设置 netif 为默认值,并且打开 netif 网口
{
    netif_set_default(&lwip_netif); //设置 netif 为默认网口
    netif_set_up(&lwip_netif);      //打开 netif 网口
}
return 0;//操作 OK.
}

```

lwip_comm_init()函数和无操作系统的基本相同,不同之处在于其所调用的 LWIP 内核初始化函数 tcpip_init()和 netif_add()函数。在无操作系统中使用 lwip_init()函数来初始化 LWIP 内核,在这里使用了 tcpip_init()来初始化 LWIP 内核,我们查看 tcpip_init()函数可以看出其实在 tcpip_init()函数中是调用了 lwip_init()的。我们再来看看 netif_add()函数,与前面不同的是,这里将 netif_add 函数最后一个参数改为函数 tcpip_input()。

在带操作系统的 lwip_comm.c 文件中我们把 DHCP 作为一个任务来处理,当 DHCP 任务执行完成后就删除掉这个任务。关于 DHCP 有三个函数: lwip_comm_dhcp_creat、lwip_comm_dhcp_delete 和 lwip_dhcp_task。前两个函数比较好理解,就是建立和删除 DHCP 任务, lwip_dhcp_task 函数就是 DHCP 的任务函数了,代码如下。

//DHCP 处理任务

```

void lwip_dhcp_task(void *pdata)
{
    u32 ip=0,netmask=0,gw=0;
    dhcp_start(&lwip_netif);//开启 DHCP
    lwipdev.dhcpstatus=0; //正在 DHCP
    printf("正在查找 DHCP 服务器,请稍等.....\r\n");
    while(1)
    {
        printf("正在获取地址...\r\n");
        ip=lwip_netif.ip_addr.addr;      //读取新 IP 地址
        netmask=lwip_netif.netmask.addr;//读取子网掩码
    }
}

```

```

gw=lwip_netif.gw.addr;           //读取默认网关
if(ip!=0)                         //当正确读取到 IP 地址的时候
{
    lwipdev.dhcpstatus=1; //DHCP 成功
    printf("网卡 en 的 MAC 地址为:.....%d.%d.%d.%d.%d.%d\r\n",\
    lwipdev.mac[0],lwipdev.mac[1],lwipdev.mac[2],\
    lwipdev.mac[3],lwipdev.mac[4],lwipdev.mac[5]);
    //解析出通过 DHCP 获取到的 IP 地址
    lwipdev.ip[3]=(uint8_t)(ip>>24);
    lwipdev.ip[2]=(uint8_t)(ip>>16);
    lwipdev.ip[1]=(uint8_t)(ip>>8);
    lwipdev.ip[0]=(uint8_t)(ip);
    printf("通过 DHCP 获取到 IP 地址.....%d.%d.%d.%d\r\n",\
    lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
    //解析通过 DHCP 获取到的子网掩码地址
    lwipdev.netmask[3]=(uint8_t)(netmask>>24);
    lwipdev.netmask[2]=(uint8_t)(netmask>>16);
    lwipdev.netmask[1]=(uint8_t)(netmask>>8);
    lwipdev.netmask[0]=(uint8_t)(netmask);
    printf("通过 DHCP 获取到子网掩码.....%d.%d.%d.%d\r\n",\
    lwipdev.netmask[0],lwipdev.netmask[1],lwipdev.netmask[2],lwipdev.netmask[3]);
    //解析出通过 DHCP 获取到的默认网关
    lwipdev.gateway[3]=(uint8_t)(gw>>24);
    lwipdev.gateway[2]=(uint8_t)(gw>>16);
    lwipdev.gateway[1]=(uint8_t)(gw>>8);
    lwipdev.gateway[0]=(uint8_t)(gw);
    printf("通过 DHCP 获取到的默认网关.....%d.%d.%d.%d\r\n",\
    lwipdev.gateway[0],lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);
    break;
}
else if(lwip_netif.dhcp->tries>LWIP_MAX_DHCP_TRIES)
{
    //通过 DHCP 服务获取 IP 地址失败,且超过最大尝试次数
    lwipdev.dhcpstatus=0XFF;//DHCP 失败.
    //使用静态 IP 地址
    IP4_ADDR(&(lwip_netif.ip_addr),lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],\
    lwipdev.ip[3]);
    IP4_ADDR(&(lwip_netif.netmask),lwipdev.netmask[0],lwipdev.netmask[1],\
    lwipdev.netmask[2],lwipdev.netmask[3]);
    IP4_ADDR(&(lwip_netif.gw),lwipdev.gateway[0],\
    lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);
    printf("DHCP 服务超时,使用静态 IP 地址!\r\n");
    printf("网卡 en 的 MAC 地址为:.....%d.%d.%d.%d.%d.%d\r\n",\
    lwipdev.mac[0],lwipdev.mac[1],lwipdev.mac[2],lwipdev.mac[3],\
    lwipdev.mac[4],lwipdev.mac[5]);
}

```

```

printf("静态 IP 地址.....%d.%d.%d.%d\r\n",\
lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
printf("子网掩码.....%d.%d.%d.%d\r\n",\
lwipdev.netmask[0],lwipdev.netmask[1],lwipdev.netmask[2],lwipdev.netmask[3]);
printf("默认网关.....%d.%d.%d.%d\r\n",\
lwipdev.gateway[0],lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);
break;
}
delay_ms(250); //延时 250ms
}
lwip_comm_dhcp_delete();//删除 DHCP 任务
}

```

在这里我们也是通过结构体 `lwipdev` 的成员变量 `dhcpstatus` 来判断 DHCP 的处理状态。当 `dhcpstatus=0` 时表示开启 DHCP，当 DHCP 完成以后让 `dhcpstatus=2`，表示 DHCP 成功。但是当 DHCP 重试次数大于 `LWIP_MAX_DHCP_TRIES` 时，意味着 DHCP 失败，这是 `dhcpstatus=0XFF`，表示 DHCP 失败，并且使用静态 IP 地址。注意最后在 DHCP 任务执行完成后调用 `lwip_comm_dhcp_delete()` 函数删除 DHCP 任务。

2.2.2 UCOSIII+LWIP 移植

UCOSIII+LWIP 的移植和 2.2.1 小节中的 UCOSII+LWIP 的移植类似，最主要的区别就是文件 `sys_arch.c` 和 `sys_arch.h` 了。本节就在 2.2.1 小节的基础上来完成 UCOSIII+LWIP 的移植。移植所使用的模板工程就使用“网络实验 2 LWIP 带 UCOSII 操作系统移植”。

1) 修改文件 `cpu.h`

UCOSIII 和 LWIP 中都有个 `cpu.h` 文件，一山不能容二虎！因此其中的某一个必须改名。这里我们将 LWIP 中的 `cpu.h` 文件名字修改为 `lwip_cpu.h`，直接在模板工程中修改。文件路径为：LWIP->arch->`cpu.h`，如图 2.2.2.1 所示：

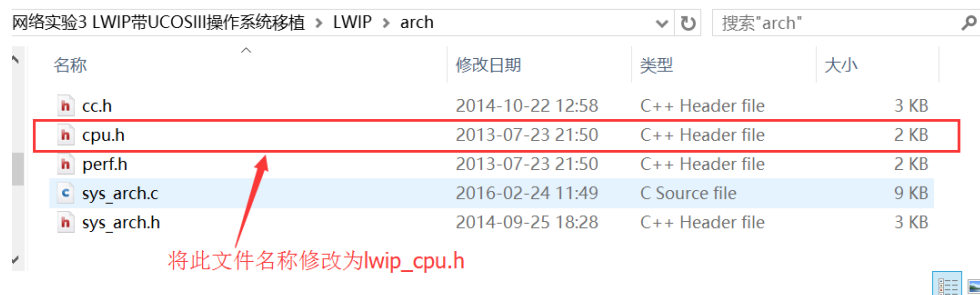


图 2.2.2.1 修改文件名

文件名修改完成以后，LWIP 中引用到 `cpu.h` 这个文件的地方也需要修改为引用 `lwip_cpu.h`。修改完成以后确保编译没有错误！

2) 添加 UCOSIII 系统源码

“网络实验 2 LWIP 带 UCOSII 操作系统移植”中使用的是 UCOSII 操作系统，因此我们要将其更换为 UCOSIII 系统。方法很简单，将《STM32F407 UCOS 开发手册》中 UCOSIII 移植例程中的 UCOSIII 文件夹复制过来。复制完成以后如图 2.2.2.2 所示：

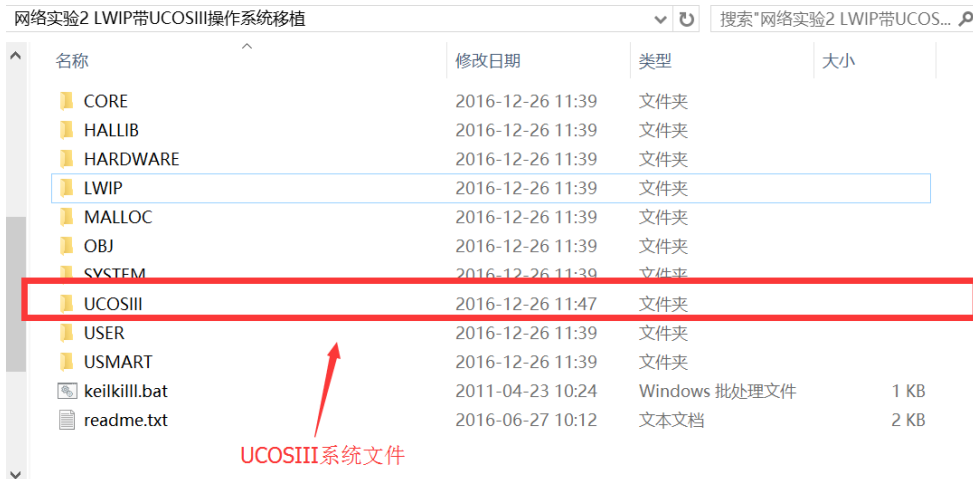


图 2.2.2.2 添加 UCOSIII 系统源码

接下来就是将 UCOSIII 系统源码添加到工程中去，添加完成以后如图 2.2.2.3 所示：

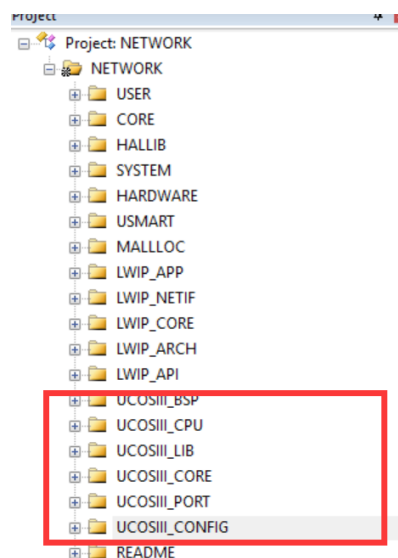


图 2.2.2.3 向工程中添加 UCOSIII 源码

UCOSIII 系统文件添加到工程中以后话需要添加相应的头文件路径，添加完成以后如图 2.2.2.4 所示：

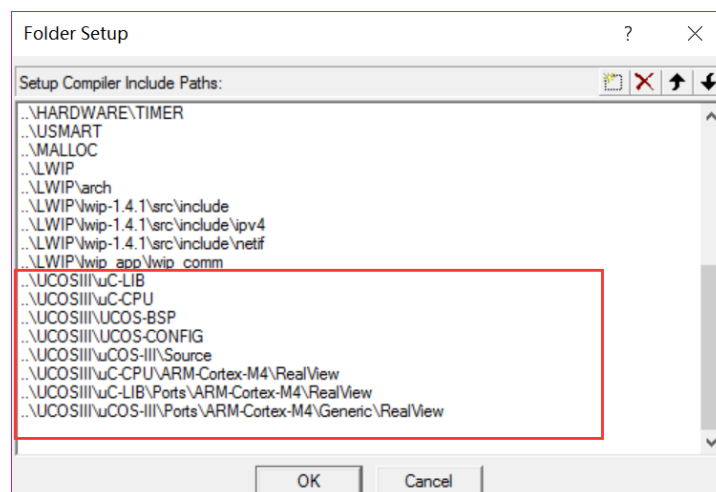


图 2.2.2.4 添加头文件路径

3) 修改 sys_arch.h 和 sys_arch.c 文件

由于 UCOSIII 和 UCOSII 的 API 函数不同，所有 sys_arch.c 和 sys_arch.h 这两个也要做相应的修改，修改后的 sys_arch.h 文件如下：

```
#ifndef __ARCH_SYS_ARCH_H__
#define __ARCH_SYS_ARCH_H__
#include <includes.h>
#include "arch/cc.h"
#include "includes.h"

#ifdef SYS_ARCH_GLOBALS
#define SYS_ARCH_EXT
#else
#define SYS_ARCH_EXT extern
#endif

#define MAX_QUEUES                OS_CFG_MSG_POOL_SIZE// 消息邮箱的数量
#define MAX_QUEUE_ENTRIES        20                // 每个消息邮箱的大小

typedef OS_SEM        sys_sem_t;    //LWIP 使用的信号量
typedef OS_MUTEX     sys_mutex_t; //LWIP 使用的互斥信号量
typedef OS_Q         sys_mbox_t;    //LWIP 使用的消息邮箱,其实就是 UCOS 中的消息队列
typedef CPU_INT08U   sys_thread_t; //线程 ID,也就是任务优先级
#endif
```

接着修改 sys_arch.c 文件，修改完成以后代码如下：

```
#define SYS_ARCH_GLOBALS

/* lwIP includes. */
#include "lwip/debug.h"
#include "lwip/def.h"
#include "lwip/lwip_sys.h"
#include "lwip/mem.h"
#include "includes.h"
#include "delay.h"
#include "arch/sys_arch.h"
#include "malloc.h"
#include "os_cfg_app.h"

//当消息指针为空时,指向一个常量 pvNullPointer 所指向的值.
//在 UCOS 中如果 OSQPost()中的 msg==NULL 会返回一条 OS_ERR_POST_NULL
//错误,而在 lwip 中会调用 sys_mbox_post(mbox,NULL)发送一条空消息,我们
//在本函数中把 NULL 变成一个常量指针 0xffffffff
const void * const pvNullPointer = (mem_ptr_t*)0xffffffff;
```

```

//创建一个消息邮箱
/*mbox:消息邮箱
//size:邮箱大小
//返回值:ERR_OK,创建成功
//      其他,创建失败
err_t sys_mbox_new( sys_mbox_t *mbox, int size)
{
    OS_ERR err;
    //消息队列最多容纳 MAX_QUEUE_ENTRIES 消息数目
    if(size>MAX_QUEUE_ENTRIES)size=MAX_QUEUE_ENTRIES;
    OSQCreate((OS_Q*      )mbox,                //消息队列
              (CPU_CHAR*  )"LWIP Quiue",        //消息队列名称
              (OS_MSG_QTY)size,                 //消息队列长度
              (OS_ERR*     )&err);              //错误码
    if(err==OS_ERR_NONE) return ERR_OK;
    return ERR_MEM;
}

//释放并删除一个消息邮箱
/*mbox:要删除的消息邮箱
void sys_mbox_free(sys_mbox_t * mbox)
{
    OS_ERR err;

#ifdef OS_CFG_Q_FLUSH_EN > 0u
    OSQFlush(mbox,&err);
#endif

    OSQDel((OS_Q*  )mbox,
           (OS_OPT )OS_OPT_DEL_ALWAYS,
           (OS_ERR* )&err);
    LWIP_ASSERT( "OSQDel ",err == OS_ERR_NONE );
}

//向消息邮箱中发送一条消息(必须发送成功)
/*mbox:消息邮箱
/*msg:要发送的消息
void sys_mbox_post(sys_mbox_t *mbox,void *msg)
{
    OS_ERR err;
    CPU_INT08U i=0;

```

```
//当 msg 为空时 msg 等于 pvNullPointer 指向的值
if(msg==NULL)msg=(void*)&pvNullPointer;
//发送消息
while(i<10) //试 10 次
{
    OSQPost((OS_Q*          )mbox,
            (void*          )msg,
            (OS_MSG_SIZE    )strlen(msg),
            (OS_OPT         )OS_OPT_POST_FIFO,
            (OS_ERR*        )&err);
    if(err==OS_ERR_NONE) break;
    i++;
    OSTimeDlyHMSM(0,0,0,5,OS_OPT_TIME_HMSM_STRICT,&err); //延时 5ms
}
LWIP_ASSERT( "sys_mbox_post error!\n", i !=10 );
}

//尝试向一个消息邮箱发送消息
//此函数相对于 sys_mbox_post 函数只发送一次消息,
//发送失败后不会尝试第二次发送
//*mbox:消息邮箱
//*msg:要发送的消息
//返回值:ERR_OK,发送 OK
//      ERR_MEM,发送失败
err_t sys_mbox_trypost(sys_mbox_t *mbox, void *msg)
{
    OS_ERR err;
```

```

//当 msg 为空时 msg 等于 pvNullPointer 指向的值
if(msg==NULL)msg=(void*)&pvNullPointer;
OSQPost((OS_Q*      )mbox,
        (void*      )msg,
        (OS_MSG_SIZE)sizeof(msg),
        (OS_OPT      )OS_OPT_POST_FIFO,
        (OS_ERR*     )&err);
if(err!=OS_ERR_NONE) return ERR_MEM;
return ERR_OK;
}

//等待邮箱中的消息
//*mbox:消息邮箱
//*msg:消息
//timeout:超时时间, 如果 timeout 为 0 的话,就一直等待
//返回值:当 timeout 不为 0 时如果成功的话就返回等待的时间,
//      失败的话就返回超时 SYS_ARCH_TIMEOUT
u32_t sys_arch_mbox_fetch(sys_mbox_t *mbox, void **msg, u32_t timeout)
{
    OS_ERR err;
    OS_MSG_SIZE size;
    u32_t ucos_timeout,timeout_new;
    void *temp;
    if(timeout!=0)
    {
        //转换为节拍数,因为 UCOS 延时使用的是节拍数,而 LWIP 是用 ms
        ucos_timeout=(timeout*OS_CFG_TICK_RATE_HZ)/1000;
        if(ucos_timeout<1)ucos_timeout=1;//至少 1 个节拍
    }else ucos_timeout = 0;
    timeout = OSTimeGet(&err); //获取系统时间
    //请求消息
    temp=OSQPend((OS_Q*      )mbox,
                (OS_TICK      )ucos_timeout,
                (OS_OPT      )OS_OPT_PEND_BLOCKING,
                (OS_MSG_SIZE* )&size,
                (CPU_TS*      )0,
                (OS_ERR*     )&err);
    if(msg!=NULL)
    {
        //因为 lwip 发送空消息的时候我们使用了 pvNullPointer 指针,所以判断 pvNullPointer
        //指向的值就可知道请求到的消息是否有效。
        if(temp==(void*)&pvNullPointer)*msg = NULL;
    }
}

```

```
        else *msg=temp;
    }
    if(err==OS_ERR_TIMEOUT)timeout=SYS_ARCH_TIMEOUT; //请求超时
    else
    {
        LWIP_ASSERT("OSQPend ",err==OS_ERR_NONE);
        timeout_new=OSTimeGet(&err);
        //算出请求消息或使用的的时间
        if (timeout_new>=timeout) timeout_new = timeout_new - timeout;
        else timeout_new = 0xffffffff - timeout + timeout_new;
        timeout=timeout_new*1000/OS_CFG_TICK_RATE_HZ + 1;
    }
    return timeout;
}

//尝试获取消息
//*mbox:消息邮箱
//*msg:消息
//返回值:等待消息所用的时间/SYS_ARCH_TIMEOUT
u32_t sys_arch_mbox_tryfetch(sys_mbox_t *mbox, void **msg)
{
    return sys_arch_mbox_fetch(mbox,msg,1);//尝试获取一个消息
}

//检查一个消息邮箱是否有效
//*mbox:消息邮箱
//返回值:1,有效.
//      0,无效
int sys_mbox_valid(sys_mbox_t *mbox)
{
    if(mbox->NamePtr)
        return (strcmp(mbox->NamePtr,"?Q"))? 1:0;
    else
        return 0;
}

//设置一个消息邮箱为无效
//*mbox:消息邮箱
void sys_mbox_set_invalid(sys_mbox_t *mbox)
{
    if(sys_mbox_valid(mbox))
        sys_mbox_free(mbox);
}
```

```

//创建一个信号量
//*sem:创建的信号量
//count:信号量值
//返回值:ERR_OK,创建 OK
//      ERR_MEM,创建失败
err_t sys_sem_new(sys_sem_t * sem, u8_t count)
{
    OS_ERR err;
    OSSemCreate ((OS_SEM*      )sem,
                  (CPU_CHAR*    )"LWIP Sem",
                  (OS_SEM_CTR   )count,
                  (OS_ERR*      )&err);
    if(err!=OS_ERR_NONE)return ERR_MEM;
    LWIP_ASSERT("OSSemCreate ",sem != NULL);
    return ERR_OK;
}

//等待一个信号量
//*sem:要等待的信号量
//timeout:超时时间
//返回值:当 timeout 不为 0 时如果成功的话就返回等待的时间,
//      失败的话就返回超时 SYS_ARCH_TIMEOUT
u32_t sys_arch_sem_wait(sys_sem_t *sem, u32_t timeout)
{
    OS_ERR err;
    u32_t ucos_timeout, timeout_new;
    if( timeout!=0)
    {
        //转换为节拍数,因为 UCOS 延时使用的是节拍数,而 LWIP 是用 ms
        ucos_timeout = (timeout * OS_CFG_TICK_RATE_HZ) / 1000;
        if(ucos_timeout < 1)
            ucos_timeout = 1;
        }else ucos_timeout = 0;
    timeout = OSTimeGet(&err);
    OSSemPend(sem,ucos_timeout,OS_OPT_PEND_BLOCKING,0,&err); //请求信号量
    if(err == OS_ERR_TIMEOUT)timeout=SYS_ARCH_TIMEOUT;        //请求超时
    else
    {
        timeout_new = OSTimeGet(&err);
        if (timeout_new>=timeout) timeout_new = timeout_new - timeout;
        else timeout_new = 0xffffffff - timeout + timeout_new;
        //算出请求消息或使用的时间(ms)
    }
}

```



```
        timeout = (timeout_new*1000/OS_CFG_TICK_RATE_HZ + 1);
    }
    return timeout;
}

//发送一个信号量
//sem:信号量指针
void sys_sem_signal(sys_sem_t *sem)
{
    OS_ERR err;
    OSSemPost(sem,OS_OPT_POST_1,&err);//发送信号量
    LWIP_ASSERT("OSSemPost ",err == OS_ERR_NONE );
}

//释放并删除一个信号量
//sem:信号量指针
void sys_sem_free(sys_sem_t *sem)
{
    OS_ERR err;
    OSSemDel(sem,OS_OPT_DEL_ALWAYS,&err);
    LWIP_ASSERT("OSSemDel ",err==OS_ERR_NONE);
    sem = NULL;
}

//查询一个信号量的状态,无效或有效
//sem:信号量指针
//返回值:1,有效.
//      0,无效
int sys_sem_valid(sys_sem_t *sem)
{
    if(sem->NamePtr)
        return (strcmp(sem->NamePtr,"?SEM"))? 1:0;
    else
        return 0;
}

//设置一个信号量无效
//sem:信号量指针
void sys_sem_set_invalid(sys_sem_t *sem)
{
    if(sys_sem_valid(sem))
        sys_sem_free(sem);
}
```

```

}

//arch 初始化
void sys_init(void)
{
    //这里,我们在该函数,不做任何事情
}

extern CPU_STK * TCPIP_THREAD_TASK_STK;//TCP IP 内核任务堆栈,在 lwip_comm 函数定义
//LWIP 内核任务的任务控制块
OS_TCB TcpipthreadTaskTCB;
//创建一个新进程
//*name:进程名称
//thred:进程任务函数
//*arg:进程任务函数的参数
//stacksize:进程任务的堆栈大小
//prio:进程任务的优先级
sys_thread_t sys_thread_new(const char *name, lwip_thread_fn thread, void *arg, int stacksize, int prio)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    if(strcmp(name,TCPIP_THREAD_NAME)==0)//创建 TCP IP 内核任务
    {
        OS_CRITICAL_ENTER(); //进入临界区
        //创建开始任务
        OSTaskCreate((OS_TCB*      )&TcpipthreadTaskTCB,//任务控制块
                    (CPU_CHAR*    )"TCPIPThread task", //任务名字
                    (OS_TASK_PTR  )thread,              //任务函数
                    (void*        )0,                   //传递给任务函数的参数
                    (OS_PRIO      )prio,                //任务优先级
                    (CPU_STK*     )&TCPIP_THREAD_TASK_STK[0],
                    (CPU_STK_SIZE )stacksize/10,        //任务堆栈深度限位
                    (CPU_STK_SIZE )stacksize,           //任务堆栈大小
                    (OS_MSG_QTY   )0,
                    (OS_TICK      )0,
                    (void*        )0,                   //用户补充的存储区
                    (OS_OPT       )OS_OPT_TASK_STK_CHK\ //任务选项
                        OS_OPT_TASK_STK_CLR,
                    (OS_ERR*      )&err);               //存放该函数错误时的返回值
        OS_CRITICAL_EXIT(); //退出临界区
    }
    return 0;
}

```

```

}

//lwip 延时函数
//ms:要延时的 ms 数
void sys_msleep(u32_t ms)
{
    delay_ms(ms);
}

//获取系统时间,LWIP1.4.1 增加的函数
//返回值:当前系统时间(单位:毫秒)
u32_t sys_now(void)
{
    OS_ERR err;
    u32_t ucso_time, lwip_time;
    ucso_time=OSTimeGet(&err);    //获取当前系统时间 得到的是 UCSO 的节拍数
    lwip_time=(ucso_time*1000/OS_CFG_TICK_RATE_HZ+1);//将节拍数转换为 LWIP 的时间
    MS
    return lwip_time;    //返回 lwip_time;
}

```

4) 修改 cpu.h 文件

文件 cpu.h 中有与临界段代码保护有关的宏定义，因此要修改为 UCOSIII 的，修改完以后的临界段代码保护部分代码如下所示：

```

//使用操作系统时的临界区保护，这里以 UCOS III 为例
//当定义了 OS_CRITICAL_METHOD 时就说明使用了 UCOS III
#if CPU_CFG_CRITICAL_METHOD == 1
#define SYS_ARCH_DECL_PROTECT(lev)
#define SYS_ARCH_PROTECT(lev)      CPU_INT_DIS()
#define SYS_ARCH_UNPROTECT(lev)    CPU_INT_EN()
#endif

#if CPU_CFG_CRITICAL_METHOD == 3
#define SYS_ARCH_DECL_PROTECT(lev)  u32_t lev
#define SYS_ARCH_PROTECT(lev)       lev = CPU_SR_Save() //进入临界区
#define SYS_ARCH_UNPROTECT(lev)     CPU_SR_Restore(lev) //退出临界区
#endif

```

5) 修改 lwip_comm.c 文件

文件 lwip_comm.c 中有 DHCP 任务的创建和删除函数，其所使用到的 API 函数要修改为 UCOSIII 的。部分修改后的宏和函数如下：

```

////////////////////////////////////
//lwip 两个任务定义(内核任务和 DHCP 任务)

```

//lwip 内核任务任务堆栈(优先级和堆栈大小在 lwipopts.h 定义了)

```
CPU_STK * TCPIP_THREAD_TASK_STK;
```

//lwip DHCP 任务

//设置任务优先级

```
#define LWIP_DHCP_TASK_PRIO          7
```

//设置任务堆栈大小

```
#define LWIP_DHCP_STK_SIZE          256
```

//任务控制块

```
OS_TCB LwipdhcpTaskTCB;
```

//任务堆栈，采用内存管理的方式控制申请

```
CPU_STK * LWIP_DHCP_TASK_STK;
```

//任务函数

```
void lwip_dhcp_task(void *pdata);
```

至此，UCOSIII+LWIP 的移植就完成了，编译查看还有没有错误，有的话请自行修改。注意：main.c 中默认肯定是针对 UCOSII 编写的测试代码，编译的话肯定会报错，这个先不用管，下一小节编会讲解如何编写测试代码。

2.3 软件设计

移植完带操作系统的 LWIP 以后就可以编写 main()函数来测试移植是否成功，main()函数代码如下。

```
int main(void)
{
    delay_init();           //延时初始化
    NVIC_Configuration();   //中断分组配置
    uart_init(115200);      //串口波特率设置
    usmart_dev.init(84);    //初始化 USMART
    LED_Init();             //LED 初始化
    KEY_Init();             //按键初始化
    LCD_Init();             //LCD 初始化

    FSMC_SRAM_Init();       //SRAM 初始化

    mymem_init(SRAMIN);     //初始化内部内存池
    mymem_init(SRAMEX);     //初始化外部内存池
    mymem_init(SRAMCCM);    //初始化 CCM 内存池

    POINT_COLOR = RED;      //红色字体
    LCD_ShowString(30,30,200,20,16,"Explorer STM32F4");
    LCD_ShowString(30,50,200,20,16,"LWIP+UCOS Test");
    LCD_ShowString(30,70,200,20,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,20,16,"2014/9/1");
```

```

OSInit();           //UCOS 初始化
while(lwip_comm_init()) //lwip 初始化
{
    LCD_ShowString(30,110,200,20,16,"Lwip Init failed!"); //lwip 初始化失败
    delay_ms(500);
    LCD_Fill(30,110,230,150,WHITE);
    delay_ms(500);
}
LCD_ShowString(30,110,200,20,16,"Lwip Init Success!"); //lwip 初始化成功
OSTaskCreate(start_task,(void*)0,(OS_STK*)&\
START_TASK_STK[START_STK_SIZE-1],START_TASK_PRIO);
OSStart(); //开启 UCOS
}

```

在主函数中我们首先完成外设的初始化，然后初始化 UCOS，调用 `lwip_comm_init()` 函数完成 LWIP 的初始化，最后创建开始任务 `start_task`。在 `start_task` 任务中如果使用 DHCP 的话就创建 DHCP 任务，在 `start_task` 任务中还创建了 `led_task` 和 `display_task` 这两个任务，`start_task` 任务代码如下。

```

//start 任务
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr;
    pdata = pdata ;

    OSStatInit();           //初始化统计任务
    OS_ENTER_CRITICAL();    //关中断
#ifdef LWIP_DHCP
    lwip_comm_dhcp_creat(); //创建 DHCP 任务
#endif
    OSTaskCreate(led_task,(void*)0,(OS_STK*)&\
LED_TASK_STK[LED_STK_SIZE-1],LED_TASK_PRIO); //创建 LED 任务

    OSTaskCreate(display_task,(void*)0,(OS_STK*)&\
DISPLAY_TASK_STK[DISPLAY_STK_SIZE-1],DISPLAY_TASK_PRIO); //显示任务

    OSTaskSuspend(OS_PRIO_SELF); //挂起 start_task 任务
    OS_EXIT_CRITICAL();         //开中断
}

```

2.4 下载验证

在代码编译成功之后，我们下载代码到 STM32F407 开发板上，这里我们开启了 DHCP，大家也可以自行尝试一下关闭 DHCP 使用静态 IP 地址，下载完成后 LCD 显示如图 2.4.1 所示，这是串口调试助手上显示如图 2.4.2 所示。

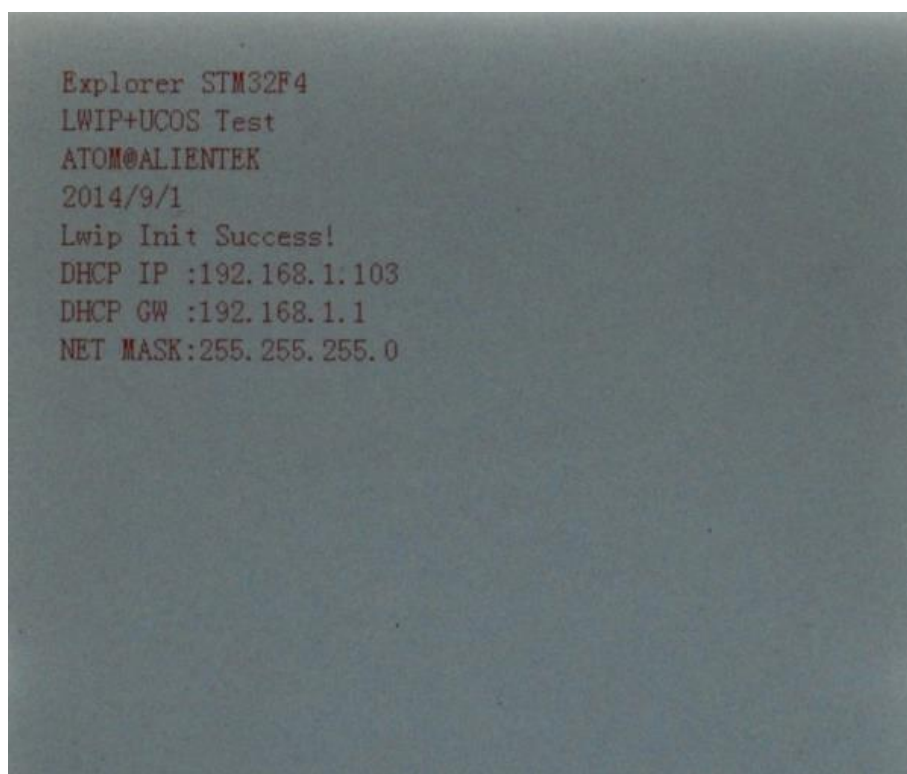


图 2.4.1 LCD 显示界面

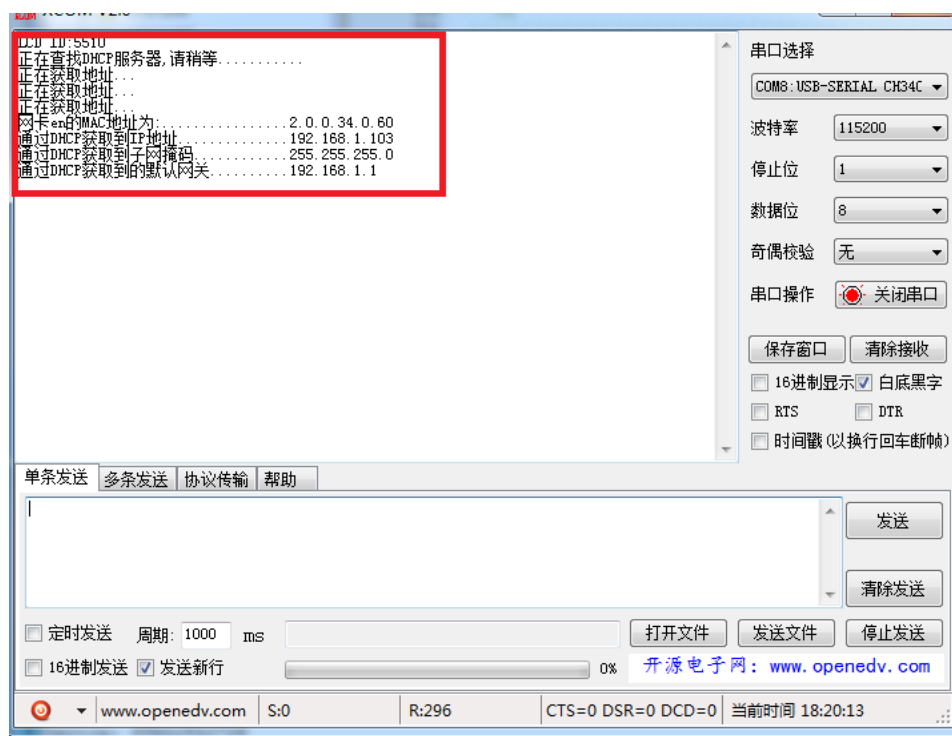


图 2.4.2 串口调试助手输出信息

可以看到此时通过路由器的 DHCP 分配到的 IP 地址为: 192.168.1.103, 默认网关为: 192.168.1.1, 子网掩码为: 255.255.255.0。在电脑上 ping 开发板的 IP 地址, 结果如图 2.4.3 所示。



```
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>ping 192.168.1.103

正在 Ping 192.168.1.103 具有 32 字节的数据:
来自 192.168.1.103 的回复: 字节=32 时间=3ms TTL=255
来自 192.168.1.103 的回复: 字节=32 时间=4ms TTL=255
来自 192.168.1.103 的回复: 字节=32 时间=2ms TTL=255
来自 192.168.1.103 的回复: 字节=32 时间=2ms TTL=255

192.168.1.103 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间<以毫秒为单位>:
        最短 = 2ms, 最长 = 4ms, 平均 = 2ms

C:\Users\Administrator>
```

图 2.4.2ping 测试

第三章 RAW 编程接口 UDP 实验

本章，我们开始学习如何使用 LWIP，本章采用 RAW API 编程接口完成开发板和电脑之间的 UDP 通信。本章实验中我们通过电脑端的网络调试助手给开发板发送数据，开发板接收并在 LCD 上显示接收到的数据，同时也可以通过按键从开发板向网络调试助手发送数据。本章分为如下几个部分：

- 3.1 RAW 编程接口 UDP 简介
- 3.2 软件设计
- 3.3 下载验证

3.1 RAW 编程接口 UDP 简介

1) UDP 协议

UDP 协议是 TCP/IP 协议栈中传输层协议，是一个简单的面向数据报的协议，在传输层中还有另一个重要的协议，那就是 TCP 协议。UDP 不提供数据包分组、组装，不能对数据包进行排序，当报文发送出去后无法知道是否安全、完整的到达。UDP 除了这些缺点外肯定有它自身的优势，由于 UDP 不属于连接型协议，因而消耗资源小，处理速度快，所以通常在音频、视频和普通数据传输时使用 UDP 较多。UDP 数据报结构如图 3.1.1 所示。



图 3.1.1 UDP 数据报结构

端口号表示发送和接收进程，UDP 协议使用端口号为不同的应用保留各自的数据传输通，UDP 和 TCP 协议都是采用端口号对同一时刻内多项应用同时发送和接收数据，而数据接收方则通过目标端口接收数据。有的网络应用只能使用预先为其预留或注册的静态端口；而另外一些网络应用则可以使用未被注册的动态端口。因为 UDP 报头使用两个字节存放端口号，所以端口号的有效范围是从 0 到 65535。一般来说，大于 49151 的端口号都代表动态端口。

数据报的长度是指包括报头和数据部分在内的总字节数。因为报头的长度是固定的，所以该域主要被用来计算可变长度的数据部分（又称为数据负载）。数据报的最大长度根据操作环境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为 65535 字节。

UDP 协议使用报头中的校验和来保证数据的安全。校验和首先在数据发送方通过特殊的算法计算得出，在传递到接收方之后，还需要再重新计算。如果某个数据报在传输过程中被第三方篡改或者由于线路噪音等原因受到损坏，发送和接收方的校验计算和将不会相符，由此 UDP 协议可以检测是否出错。

2) LWIP 中 UDP 协议函数

在 LWIP 源码中 `udp.c` 和 `udp.h` 这两个文件是关于 UDP 协议的，这两个文件中有很多函数，要想对 UDP 有详细的了解的话可以查阅这两个文件。在这里推荐一本我本人认为很好的 LWIP 参考书《嵌入式网络那些事 LWIP 协议栈深度剖析与实战演练》，作者朱升林，在这本书中对于 LWIP 的源码做了非常详细的讲解，想要研究 LWIP 源码的可以看看这本书，不过这本书中采用的是 1.3.2 版本的 LWIP，本教程采用的是 1.4.1 版本的 LWIP，所以有些函数会有差别，这一点大家一定要注意一下。

`udp.c` 中与 UDP 报文处理有关的函数之间的关系如图 3.1.2 所示。

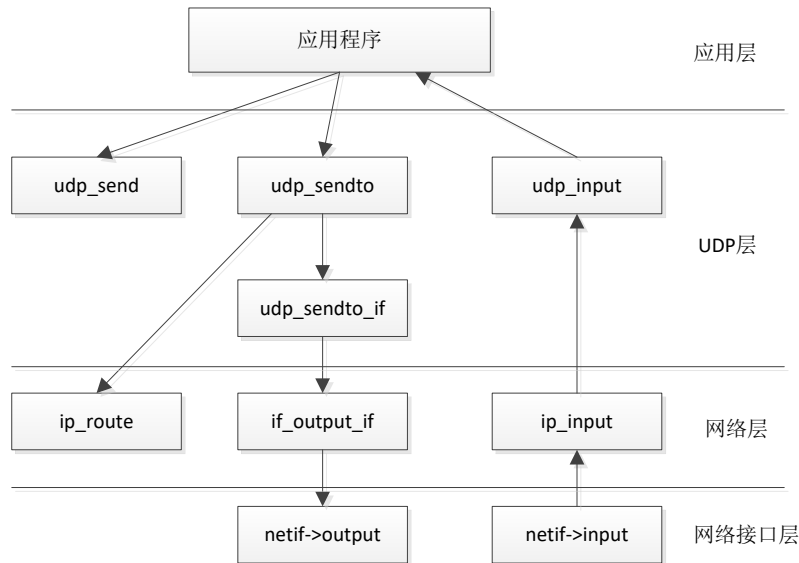


图 3.1.2 UDP 层函数关系

3) LWIP 中 RAW API 编程接口中与 UDP 相关的函数

LWIP 的 RAW API 编程方式是基于回调机制的，当我们初始化应用的时候我们必须为内核中不同的事件注册相应的回调函数，当相应的事件发生的时候这些回调函数就会被调用。表 3.1.1 给出了 UDP 的 RAW API 功能函数，我们使用这些函数来完成 UDP 的数据发送和接收。

RAW API 函数	描述
udp_new	新建一个 UDP 的 PCB 块
udp_remove	将一个 PCB 控制块从链表中删除，并且释放这个控制块的内存
udp_bind	为 UDP 的 PCB 控制块绑定一个本地 IP 地址和端口号
udp_connect	连接到指定 IP 地址主机的指定端口上，其实就是设置 PCB 控制块的 remote_ip 和 remote_port。
udp_disconnect	断开连接，将控制块设置为非连接状态，其实就是清除 remote_ip 和 remote_port 字段
udp_send	通过一个 PCB 控制块发送数据
udp_recv	需要创建的一个回调函数，当接收到数据的时候被调用

表 3.1.1 UDP 的 RAW API 功能函数

表 3.1.1 只是列出了我们在编程时需要使用到的函数，关于 UDP 的函数还有好多，这里就不一一列举，关于 UDP 的其他函数大家可以参考 udp.c 文件。

3.2 软件设计

上一节中我们简单的介绍了几个 LWIP 中关于 UDP 的函数，本节中我们就用这几个函数编写我们本章的例程，本章实验的目标是 PC 端和开发板通过 UDP 协议连接起来，PC 端使用网络调试助手向开发板发送数据，开发板接收并在 LCD 上显示接收到的数据，我们也可以通过开发板上的按键发送数据给 PC。本章实验中我们主要有两个文件 udp_demo.c 和 udp_demo.h，这两个文件在如图 3.2.1 所示的位置，注意文件的路径，以后我们所有关于网络的实验都会放到这个目录下。

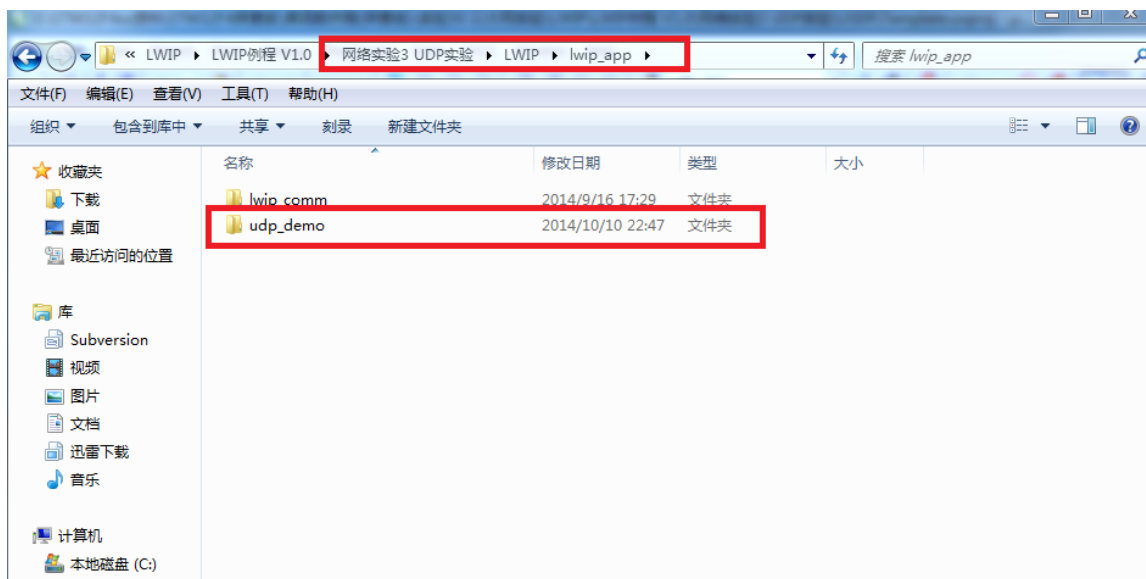


图 3.2.1 UDP 实验文件所在位置

我们打开 `udp_demo.h` 文件, 在这个文件中我们定义了 UDP 接收缓冲区的大小和 UDP 连接的端口号另外就是一些函数的声明, `udp_demo.h` 这个文件很简单。我们需要重点来看一下的是 `udp_demo.c` 这个文件, 在这个文件中我们有 5 个函数, 如表 3.2.1 所示。

函数	描述
<code>udp_demo_set_remoteip()</code>	设置远端 IP 地址, UDP 连接的时候要要用到这个地址
<code>udp_demo_test()</code>	UDP 实验测试函数, 本实验的主要内容就在这个函数里面
<code>udp_demo_recv()</code>	UDP 接收数据的回调函数, 回调函数是要我们自己编写的
<code>udp_demo_senddata()</code>	发送数据, 通过这个函数可以将我们要发送的数据发送出去
<code>udp_demo_connection_close()</code>	关闭 UDP 连接

表 3.2.1 UDP 实验函数

我们首先看一下 `udp_demo_set_remoteip()` 函数, 该函数的代码如下。

```
void udp_demo_set_remoteip(void)
{
    u8 *tbuf;
    u16 xoff;
    u8 key;
    LCD_Clear(WHITE);
    POINT_COLOR=RED;
    LCD_ShowString(30,30,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,50,200,16,16,"UDP Test");
    LCD_ShowString(30,70,200,16,16,"Remote IP Set");
    LCD_ShowString(30,90,200,16,16,"KEY0:+ KEY2:-");
    LCD_ShowString(30,110,200,16,16,"KEY_UP:OK");
    tbuf=mymalloc(SRAMIN,100); //申请内存
    if(tbuf==NULL)return;
    //前三个 IP 保持和 DHCP 得到的 IP 一致
    lwipdev.remoteip[0]=lwipdev.ip[0];
    lwipdev.remoteip[1]=lwipdev.ip[1];
```

```

lwipdev.remoteip[2]=lwipdev.ip[2];
sprintf((char*)tbuf,"Remote
IP:%d.%d.%d.",lwipdev.remoteip[0],lwipdev.remoteip[1],lwipdev.remoteip[2]);//远端 IP
LCD_ShowString(30,150,210,16,16,tbuf);
POINT_COLOR=BLUE;
xoff=strlen((char*)tbuf)*8+30;
LCD_ShowxNum(xoff,150,lwipdev.remoteip[3],3,16,0);
while(1)
{
    key=KEY_Scan(0);
    if(key==WKUP_PRES)break;
    else if(key)
    {
        if(key==KEY0_PRES)lwipdev.remoteip[3]++;//IP 增加
        if(key==KEY2_PRES)lwipdev.remoteip[3]--;//IP 减少
        LCD_ShowxNum(xoff,150,lwipdev.remoteip[3],3,16,0X80);//显示新 IP
    }
}
myfree(SRAMIN,tbuf);
}

```

这个函数首先是在 LCD 上显示一些提示信息，重点是 while() 循环，在这个循环了我们通过按键 KEY0 和 KEY2 来设置需要连接的远端 IP 地址，这个函数很简单。

udp_demo_recv() 函数，这个函数是 UDP 实验的接收回调函数，在这个函数中我们根据参数 p 来做不同的处理，当 p 为 NULL 的时候说明接收到的为空数据，当接收到数据为空的时候我们调用 udp_disconnect() 函数来关闭 UDP 连接，并且标记 UDP 连接断开，既清除 udp_demo_flag 的 bit5，当 p 不为 NULL 的时候，表示接收到数据，我们就进行如下代码所示处理(这里只截取出 udp_demo_recv() 函数的部分代码)

if(p!=NULL) //接收到不为空的数据时

```

{
    memset(udp_demo_recvbuf,0,UDP_DEMO_RX_BUFSIZE); //数据接收缓冲区清零
    for(q=p;q!=NULL;q=q->next) //遍历完整个 pbuf 链表
    {
        //判断要拷贝到 UDP_DEMO_RX_BUFSIZE 中的数据是否大于
        //UDP_DEMO_RX_BUFSIZE 的剩余空间，如果大于
        //的话就只拷贝 UDP_DEMO_RX_BUFSIZE 中剩余长度的数据，否则的话就拷贝
        //所有的数据
        if(q->len > (UDP_DEMO_RX_BUFSIZE-data_len))
            memcpy(udp_demo_recvbuf+data_len,q->payload,\
                (UDP_DEMO_RX_BUFSIZE-data_len)); //拷贝数据
        else memcpy(udp_demo_recvbuf+data_len,q->payload,q->len);
        data_len += q->len;
        if(data_len > UDP_DEMO_RX_BUFSIZE) break; //超出 TCP 客户端接收数组,跳出
    }
}

```

```

    }
    upcb->remote_ip=*addr;           //记录远程主机的 IP 地址
    upcb->remote_port=port;          //记录远程主机的端口号
    lwipdev.remoteip[0]=upcb->remote_ip.addr&0xff;      //IADDR4
    lwipdev.remoteip[1]=(upcb->remote_ip.addr>>8)&0xff;  //IADDR3
    lwipdev.remoteip[2]=(upcb->remote_ip.addr>>16)&0xff; //IADDR2
    lwipdev.remoteip[3]=(upcb->remote_ip.addr>>24)&0xff; //IADDR1
    udp_demo_flag|=1<<6; //标记接收到数据了
    pbuf_free(p); //释放内存
}

```

从上面的代码中可以看出当接收到数据以后我们先清除 UDP 的接收缓冲区数组：udp_demo_recvbuf。LWIP 在接收数据的时候会通过 pbuf_alloc() 函数来申请内存，申请到的内存被组织成一个链表，因此我们这里遍历完整个链表，将链表中的数据拷贝到 udp_demo_recvbuf 数组中。最后我们记录下远端主机的 IP 地址和端口号，标记接收到数据。

udp_demo_senddata() 函数是用来发送数据的，在发送数据前我们先通过 pbuf_alloc() 函数申请内存，当内存申请成功以后我们将发送缓冲区 tcp_demo_sendbuf 的首地址填入到 ptr 的 payload 字段，然后调用 udp_send() 函数将数据发送出去，最后释放申请到的内存，代码如下。
//UDP 服务器发送数据

```

void udp_demo_senddata(struct udp_pcb *upcb)
{
    struct pbuf *ptr;
    //申请内存
    ptr=pbuf_alloc(PBUF_TRANSPORT,strlen((char*)tcp_demo_sendbuf),PBUF_POOL);
    if(ptr)
    {
        pbuf_take(ptr,(char*)tcp_demo_sendbuf,strlen((char*)tcp_demo_sendbuf));
        udp_send(upcb,ptr);    //udp 发送数据
        pbuf_free(ptr); //释放内存
    }
}

```

udp_demo_connection_close() 函数是用来关闭 UDP 连接的，这个函数很简单，通过调用函数 udp_disconnect() 来关闭连接，然后调用 udp_remove() 函数将当前被关闭的连接控制块从当前连接控制块链表中删除，代码如下(为了方便，这里并未给出源码中的 LCD 显示程序)。

```

void udp_demo_connection_close(struct udp_pcb *upcb)
{
    udp_disconnect(upcb);
    udp_remove(upcb);    //断开 UDP 连接
    udp_demo_flag &= ~(1<<5); //标记连接断开
}

```

最后我们重点介绍一下 udp_demo_test() 这个函数，这个函数我们首先调用 udp_demo_set_remoteip() 函数来设置远端 IP 地址，然后就是在 LCD 上显示一些信息。显示完成以后就是我们这个函数的重点，代码如下。

```

udppcb=udp_new();

```

```

if(udppcb)//创建成功
{
    IP4_ADDR(&rmtipaddr,lwipdev.remoteip[0],lwipdev.remoteip[1],\
    lwipdev.remoteip[2],lwipdev.remoteip[3]);
    //UDP 客户端连接到指定 IP 地址和端口号的服务器
    err=udp_connect(udppcb,&rmtipaddr,UDP_DEMO_PORT);
    if(err==ERR_OK)
    {
        err=udp_bind(udppcb,IP_ADDR_ANY,UDP_DEMO_PORT);//绑定本地 IP 地址与端口号
        if(err==ERR_OK) //绑定完成
        {
            udp_recv(udppcb,udp_demo_recv,NULL);//注册接收回调函数
            LCD_ShowString(30,210,210,16,16,"STATUS:Connected  ");
            udp_demo_flag |= 1<<5;           //标记已经连接上
            POINT_COLOR=RED;
            LCD_ShowString(30,230,lcddev.width-30,lcddev.height-190,16,"Receive Data:");
            POINT_COLOR=BLUE;//蓝色字体
        }else res=1;
    }else res=1;
}
while(res==0)
{
    key=KEY_Scan(0);
    if(key==WKUP_PRES)break;
    if(key==KEY0_PRES)//KEY0 按下了,发送数据
    {
        udp_demo_senddata(udppcb);
    }
    if(udp_demo_flag&1<<6)//是否收到数据?
    {
        LCD_Fill(30,250,lcddev.width-1,lcddev.height-1,WHITE);//清上一次数据
        LCD_ShowString(30,250,lcddev.width-30,lcddev.height-230,16,udp_demo_recvbu
        udp_demo_flag&=~(1<<6);//标记数据已经被处理了.
    }
    lwip_periodic_handle();
    delay_ms(2);
    t++;
    if(t==200)
    {
        t=0;
        LED0=!LED0;
    }
}
}

```


在上面代码中我们完成了一下几个功能：

- 1、调用 `udp_new()` 函数创建一个 UDP 控制块 `upcb`。
- 2、当创建成功以后，调用 `udp_connect()` 函数连接到指定的 IP 地址和端口号的主机，如果 UDP 控制块 `upcb` 创建失败的话 `res` 等于 1。
- 3、连接成功以后调用 `udp_bind()` 函数绑定本地 IP 地址和端口号，连接失败的话就让 `res` 等于 1。
- 4、绑定成功以后使用 `udp_recv()` 函数注册 UDP 的接收函数，并且置位 `udp_demo_flag` 的 bit5，表示已经连接上(UDP 是非可靠连接,这里仅仅表示本地 UDP 已经准备好)，如果绑定失败的话令 `res` 等 1。
- 5、如果 `res` 等于 0 的话那么表示以上 4 个步骤都成功了，成功以后就进入 `while` 循环，失败的话就调用 `udp_demo_connection_close()` 函数断开连接，并且释放内存。
- 6、进入 `while` 以后我们通过按键来做不同的处理，当按下 `KWY_UP` 键的时候退出循环，当按下 `KWY0` 键的时候发送数据。通过读取 `udp_demo_flag` 的 bit6 来判断是否接收到数据，当接收到数据的时候就在 LCD 上显示接收到的数据。
- 7、这点特别重要！在 `while` 循环中一定要调用 `lwip_periodic_handle()` 函数，如果不调用这个函数那么 LWIP 的内核就不能运行，那么网络肯定不会工作！

至此，`udp_demo.c` 文件就讲完了，接下来就是编写 `main` 函数，`main` 函数相对来说简单一点，`mian` 函数代码如下，本实验完整工程请参考“[网络实验 4 RAW_UDP 实验](#)”。

```
int main(void)
{
    u8 key;
    delay_init();           //延时初始化
    NVIC_Configuration();   //中断分组配置
    uart_init(115200);      //串口波特率设置
    usmart_dev.init(84);    //初始化 USMART
    LED_Init();             //LED 初始化
    KEY_Init();             //按键初始化
    LCD_Init();             //LCD 初始化
    FSMC_SRAM_Init();       //初始化外部 SRAM

    mymem_init(SRAMIN);     //初始化内部内存池
    mymem_init(SRAMEX);     //初始化外部内存池
    mymem_init(SRAMCCM);    //初始化 CCM 内存池

    POINT_COLOR = RED;      //红色字体
    lwip_test_ui(1);        //加载前半部分 UI
    TIM3_Int_Init(999,839); //100khz 的频率,计数 1000 为 10ms
    while(lwip_comm_init()) //lwip 初始化
    {
        LCD_ShowString(30,150,200,20,16,"LWIP Init Falied!");
        delay_ms(1200);
        LCD_Fill(30,110,230,130,WHITE); //清除显示
        LCD_ShowString(30,110,200,16,16,"Retrying...");
    }
}
```

```

    }
    LCD_ShowString(30,110,200,20,16,"LWIP Init Success!");
    LCD_ShowString(30,130,200,16,16,"DHCP IP configing..."); //等待 DHCP 获取
#endif LWIP_DHCP
    //等待 DHCP 获取成功/超时溢出
    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0XFF))
    {
        lwip_periodic_handle();
    }
#endif
    lwip_test_ui(2);        //加载后半部分 UI
    delay_ms(500);          //延时 1s
    delay_ms(500);
    udp_demo_test();        //TCP 模式
    while(1)
    {
        key = KEY_Scan(0);
        if(key == KEY1_PRES)    //按 KEY1 键建立连接
        {
            if((udp_demo_flag & 1<<5)) printf("UDP 连接已经建立,不能重复连接\r\n"); /
            else udp_demo_test(); //当断开连接后,调用 udp_demo_test()函数
        }
        delay_ms(10);
    }
}

```

main 函数一开始完成外设的初始化，如果开启 DHCP 的话通过 DHCP 获取 IP 地址，IP 地址获取成功以后就调用 `udp_demo_test()` 函数进入 UDP 实验。我们知道在 `udp_demo_test()` 函数中有一个 `while()` 循环，当从这个循环退出来以后就会进入 main 函数的 `while()` 循环中，在 main 函数的 `while()` 循环中当 KEY1 按下并且 UDP 连接已经断开就调用 `udp_demo_test()` 函数重新开始 UDP 实验。

3.3 下载验证

下载完代码后，打开网络调试助手，等待开发板的 LCD 出现如图 3.3.7 所示界面，在这个界面上我们通过按键 KEY2 和 KEY0 设置远端 IP 地址，也就是电脑的 IP 地址，设置好以后按 KEY_UP 确认，确认完了以后 LCD 就如图 3.3.8 所示的数据接收界面。

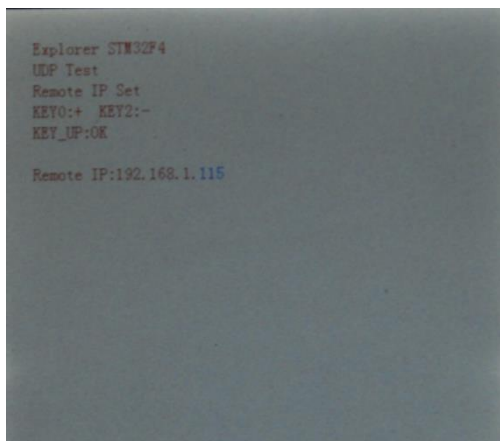


图 3.3.7 远端 IP 地址设置

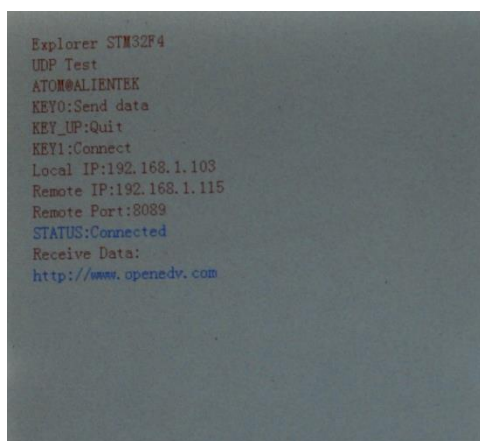


图 3.3.8 数据接收界面

接下来设置电脑端的网络调试助手，设置完成后点击网络调试助手的“连接”，操作完后的网络调试助手如图 3.3.9 所示，

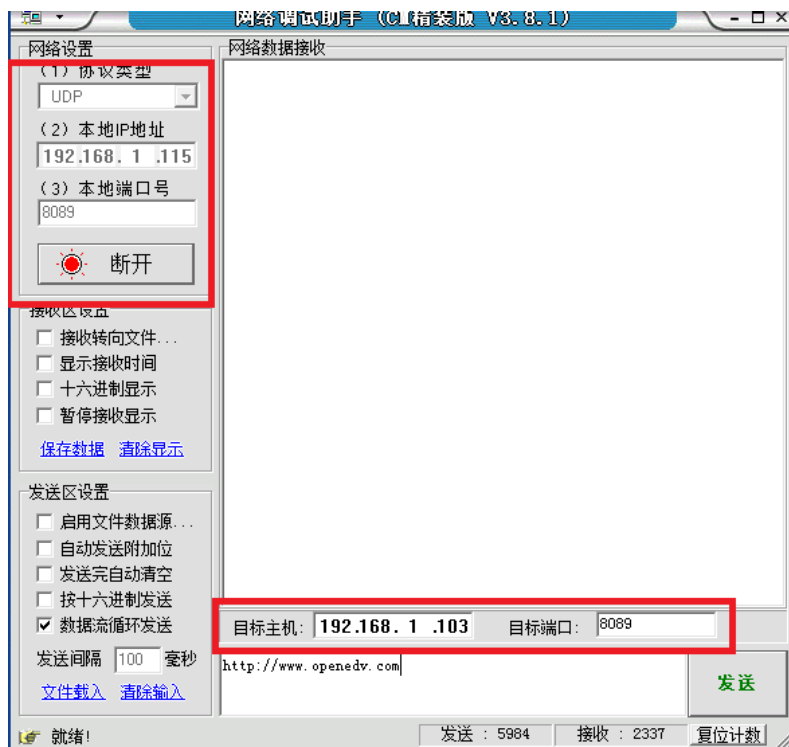


图 3.3.9 网络调试助手设置

设置完网络调试助手后在发送填入要发送的数据，这里填写了我们的开源电子网论坛地址: <http://www.openedv.com>，然后点击发送，这时我们开发板的 LCD 上显示如图 3.3.10 所示，我们可以看到在 LCD 上显示出了电脑端发送过来的数据，我们通过按下 KEY0，向电脑端发送数据: Explorer STM32F407 UDP demo send data。我们可以在网络调试助手看到图 3.3.10 所示，表明网络调试助手接收到开发板发送的数据，这里我们按了 11 次 KEY0，因此在网络调试助手上有 11 行数据。

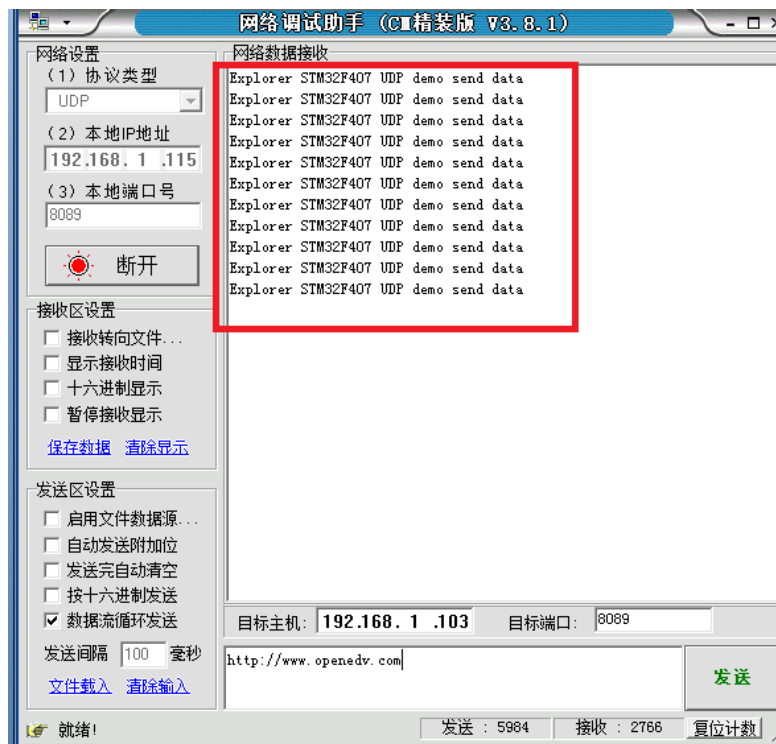


图 3.3.10 UDP 测试

第四章 RAW 编程接口 TCP 客户端实验

本章采用 RAW API 编程接口完成开发板和电脑之间的 TCP 通信。在本章中开发板做 TCP 客户端，网络调试助手做 TCP 服务器，实验中我们通过电脑端的网络调试助手给开发板发送数据，开发板接收并在 LCD 上显示接收到的数据，同时也可以通过按键从开发板向网络调试助手发送数据。本章分为如下几个部分：

- 4.1 RAW 编程接口 TCP 简介
- 4.2 软件设计
- 4.3 下载验证

4.1 RAW 编程接口 TCP 简介

1) TCP 协议简介

TCP 是一种面向连接的、可靠的、基于 IP 的传输层协议，面向连接意味着两个使用 TCP 的应用在彼此交换数据之前必须先建立一个 TCP 连接。当应用层向 TCP 层发送用于网间传输的、用 8 位字节表示的数据流，TCP 则把数据流分割成适当长度的报文段，最大传输段大小(MSS)通常受该计算机连接的网路的数据链路层的最大传送单元 (MTU) 限制。之后 TCP 把数据包传给 IP 层，由它来通过网络将包传送给接收端的 TCP 层。

TCP 为了保证报文传输的可靠，就给每个包一个序号，同时序号也保证了传送到接收端的数据包能被按序接收。然后接收端对已成功收到的字节发回一个相应的确认(ACK)；如果发送端在合理的往返时延(RTT)内未收到确认，那么对应的数据（假设丢失了）将会被重传。

在数据正确性与合法性上，TCP 用一个校验和函数来检验数据是否有错误，在发送和接收时都要计算校验和。在保证可靠性上，采用超时重传和捎带确认机制。在流量控制上，采用滑动窗口协议，协议中规定，对于窗口内未经确认的分组需要重传。在拥塞控制上，采用 TCP 拥塞控制算法。

2) TCP 首部

TCP 数据被封装在一个 IP 数据报中，如图 4.1.1 所示。

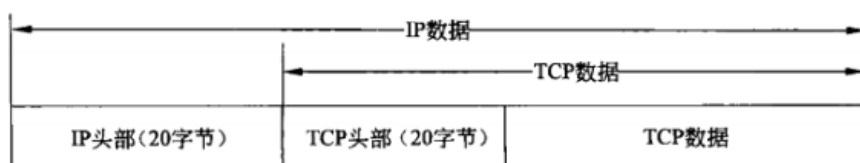


图 4.1.1 TCP 数据在 IP 数据报中的封装

图 4.1.2 所示为 TCP 报文数据格式，在没有选项的情况下，它通常是 20 个字节。

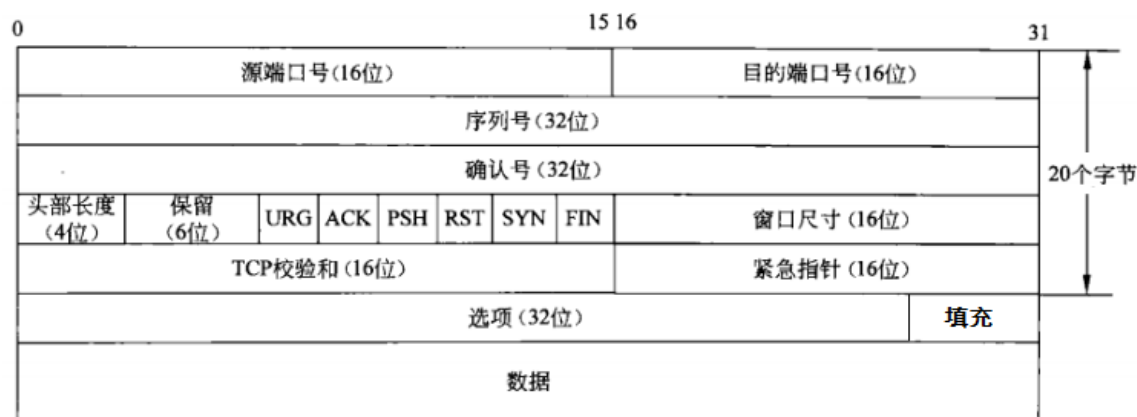


图 4.1.2 TCP 报文数据格式

源端口号和目的端口号用于寻找发送端和接收端的应用进程，这个和 UDP 报文相同，这两个值加上 IP 首部中的源 IP 地址和目的 IP 地址唯一确定一个 TCP 连接。

序列号字段用来标识从 TCP 发送端向 TCP 接收端发送的数据字节流，它表示在这个报文段中的第一个数据字节。当建立一个新的连接时，握手报文中的 SYN 标志置 1，这个握手报文中的序号字段为随机选择的初始序号 ISN(Initial Sequence Number)，当连接建立好以后发送方要发送的第一个字节序号为 ISN+1。

确认号字段只有在 ACK 为 1 的时候才有用，确认号中包含发送确认的一方所期望收到的下一个序号，确认号是在上一次成功接收到的数据字节序列号上加一，例如上次接收成功接收

到对方发过来的数据序号为 X ，那么返回的确认号就应该为 $X+1$ 。

头部长度也叫首部长度，首部长度中给出了首部的长度，以 32bit 也就是 4 字节为单位，这个字段有 4bit，因此 TCP 最多有 60 字节的首部，如果没有任何的选项字段，正常的首部长度是 20 字节。

TCP 首部中还有 6 个标志比特，这 6 个标志位的说明如表 4.1.1 所示。

表 4.1.1 TCP 首部标志位说明

标志位	说明
URG	该位置 1 时表示紧急指针有效
ACK	该位置 1 表示确认序号字段有效
PSH	改为置 1 表示接收方应该尽快将这个报文段交给应用层
RST	该位置 1 表示重建连接
SYN	用来发起连接
FIN	发送端完成发送任务，终止连接

窗口尺寸也就窗口大小，其中填写相应的值以通知对方自己期望接收的字节数，窗口大小字段是 TCP 流量控制的关键字段，窗口大小是一个 16bit 的字段，因此窗口大小最大为 65535 字节。

16 位的校验和和 UDP 的校验和的计算过程和原理相同，这是一个强制性的字段，校验和覆盖了整个 TCP 报文段：TCP 首部和 TCP 数据。

紧急指针只有在 URG 置 1 时有效，紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。

3) LWIP 中 RAW API 编程接口中与 TCP 相关的函数

tcp.c、tcp.h、tcp_in.c 和 tcp_out.c 是 LWIP 中关于 TCP 协议的文件，TCP 层中函数的关系如图 4.1.3 所示。

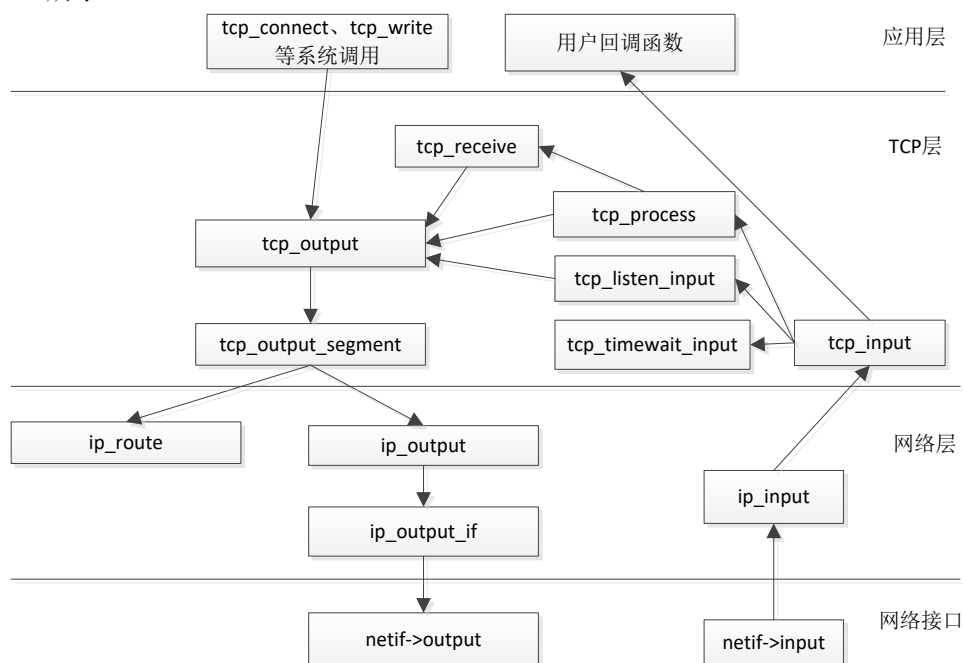


图 4.1.3 TCP 层函数关系图

LWIP 提供了很多关于 TCP 的 RAW API 编程函数,我们可以使用这些函数来完成有关 TCP 的实验,我们在表 4.1.2 列出了一部分函数。

函数分组	API 函数	函数功能描述
TCP 连接建立	tcp_new()	创建一个 TCP 的 PCB 控制块
	tcp_bind()	为 TCP 的 PCB 控制块绑定一个本地 IP 地址和端口号
	tcp_listen()	开始 TCP 的 PCB 监听
	tcp_accept()	控制块 accept 字段注册的回调函数,侦听到连接时被调用
	tcp_accepted()	通知 LWIP 协议栈一个 TCP 连接被接受了
	tcp_connect()	连接远端主机
发送 TCP 数据	tcp_write()	构造一个报文并放到控制块的发送缓冲队列中
	tcp_sent()	控制块 sent 字段注册的回调函数,数据发送成功后被回调
	tcp_output()	将发送缓冲队列中的数据发送出去
接收 TCP 数据	tcp_recv()	控制块 recv 字段注册的回调函数,当接收到新数据时被调用
	tcp_recved()	当程序处理完数据后一定要调用这个函数,通知内核更新接收窗口
轮询函数	tcp_poll()	控制块 poll 字段注册的回调函数,该函数周期性调用
关闭和中止连接	tcp_close()	关闭一个 TCP 连接
	tcp_err()	控制块 err 字段注册的回调函数,遇到错误时被调用
	tcp_abort()	中断 TCP 连接

表 4.1.2 TCP RAW API 函数

4.2 软件设计

上一节中我们简单的介绍了几个 LWIP 中关于 TCP 的函数,本节中我们就用这几个函数编写我们本章的例程,本章实验的目标是 PC 端和开发板通过 TCP 协议连接起来,开发板做 TCP 客户端,PC 端的网络调试助手配置成服务器。开发板接收服务器发送类的数据并在 LCD 上显示接收到的数据,我们也可以通过开发板上的按键发送数据给 PC。本章实验中我们主要有两个文件 tcp_client_demo.c 和 tcp_client_demo.h。

tcp_client_demo.h 文件很简单,这里就不讲解,我们重点讲解一下 tcp_client_demo.c 这个文件,在 tcp_client_demo.c 文件中我们一共定义了 9 个函数,如表 4.2.1 所示。

TCP 客户端函数	函数说明
tcp_client_set_remoteip()	设置远端 IP 地址,也就是要连接的服务器 IP 地址
tcp_client_test()	TCP 客户端测试程序
tcp_client_connected()	TCP 连接建立后调用的回调函数
tcp_client_recv()	接收到数据时的回调函数
tcp_client_error()	接收错误时的回调函数
tcp_client_poll()	轮询函数
tcp_client_sent()	发送回调函数
tcp_client_senddata()	发送数据
tcp_client_connection_close()	关闭 TCP 连接

表 4.2.1 TCP 客户端函数

tcp_client_set_remoteip()函数就是设置要连接的服务器的 IP 地址,基本和 UDP 实验的一样,

tcp_client_connected()函数是当 TCP 连接建立以后调用的回调函数，函数源码如下。

//lwIP TCP 连接建立后调用回调函数

```
err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t err)
{
    struct tcp_client_struct *es=NULL;
    if(err==ERR_OK)
    {
        es=(struct tcp_client_struct*)mem_malloc(sizeof(struct tcp_client_struct)); //申请内存
        if(es) //内存申请成功
        {
            es->state=ES_TCPCLIENT_CONNECTED;//状态为连接成功
            es->pcb=tpcb;
            es->p=NULL;
            tcp_arg(tpcb,es);           //使用 es 更新 tpcb 的 callback_arg
            tcp_recv(tpcb,tcp_client_recv); //初始化 LwIP 的 tcp_recv 回调功能
            tcp_err(tpcb,tcp_client_error); //初始化 tcp_err()回调函数
            tcp_sent(tpcb,tcp_client_sent); //初始化 LwIP 的 tcp_sent 回调功能
            tcp_poll(tpcb,tcp_client_poll,1); //初始化 LwIP 的 tcp_poll 回调功能
            tcp_client_flag|=1<<5;         //标记连接到服务器了
            err=ERR_OK;
        }else
        {
            tcp_client_connection_close(tpcb,es);//关闭连接
            err=ERR_MEM; //返回内存分配错误
        }
    }else
    {
        tcp_client_connection_close(tpcb,0);//关闭连接
    }
    return err;
}
```

这个函数的重点是注册 TCP 控制块中相关字段的回调函数，这样当有事件发生时就会调用相应的回调函数来处理这个事件。

tcp_client_recv()函数是当接收到数据时的回调函数，在这个函数中我们根据不同的状态有不同的处理，这里最重要的就是当处于连接状态并且接收到数据时的处理，这个时候我们将遍历完接收数据的 pbuf 链表，将链表中的所有数据拷贝到 tcp_client_recvbuf 中这个过程和 UDP 的接收处理过程相似。数据接收成功以后我们将 tcp_client_flag 的 bit5 置 1，表示接收到数据，tcp_client_recv()函数代码如下。

//lwIP tcp_recv()函数的回调函数

```
err_t tcp_client_recv(void *arg,struct tcp_pcb *tpcb,struct pbuf *p,err_t err)
{
    u32 data_len = 0;
    struct pbuf *q;
```

```

struct tcp_client_struct *es;
err_t ret_err;
LWIP_ASSERT("arg != NULL",arg != NULL);
es=(struct tcp_client_struct *)arg;
if(p==NULL)//如果从服务器接收到空的数据帧就关闭连接
{
    es->state=ES_TCPCLIENT_CLOSING;//需要关闭 TCP 连接了
    es->p=p;
    ret_err=ERR_OK;
}else if(err!= ERR_OK)//当接收到一个非空的数据帧,但是 err!=ERR_OK
{
    if(p)pbuf_free(p);//释放接收 pbuf
    ret_err=err;
}else if(es->state==ES_TCPCLIENT_CONNECTED)//当处于连接状态时
{
    if(p!=NULL)//当处于连接状态并且接收到的数据不为空时
    {
        //数据接收缓冲区清零
        memset(tcp_client_recvbuf,0,TCP_CLIENT_RX_BUFSIZE);
        for(q=p;q!=NULL;q=q->next) //遍历完整个 pbuf 链表
        {
            //判断要拷贝到 TCP_CLIENT_RX_BUFSIZE 中的数据是否大于
            //TCP_CLIENT_RX_BUFSIZE 的剩余空间, 如果大于的话就只拷贝
            //TCP_CLIENT_RX_BUFSIZE 中剩余长度的数据, 否则拷贝所有的数据
            if(q->len>(TCP_CLIENT_RX_BUFSIZE-data_len))
            //拷贝数据
            memcpy(tcp_client_recvbuf+data_len,q->payload, \
            (TCP_CLIENT_RX_BUFSIZE-data_len));
            else memcpy(tcp_client_recvbuf+data_len,q->payload,q->len);
            data_len += q->len;
            if(data_len > TCP_CLIENT_RX_BUFSIZE) break; //超出 TCP 客户端接收数组,跳出
        }
        tcp_client_flag|=1<<6; //标记接收到数据了
        tcp_recved(tpcb,p->tot_len);//用于获取接收数据,通知 LWIP 可以获取更多数据
        pbuf_free(p); //释放内存
        ret_err=ERR_OK;
    }
}else //接收到数据但是连接已经关闭,
{
    tcp_recved(tpcb,p->tot_len);//用于获取接收数据,通知 LWIP 可以获取更多数据
    es->p=NULL;
    pbuf_free(p); //释放内存
    ret_err=ERR_OK;
}

```

```

    }
    return ret_err;
}

```

tcp_client_error()函数是控制块中 errf 字段的回调函数,当出现知名错误的时候就会被调用,这里我们没有实现这个函数,用户可以根据自己的实际情况来实现这个函数。

tcp_client_poll()函数为控制块中 poll 字段的回调函数,这个函数会被周期调用,因此在这个函数中我们可以将要发送的数据发送出去。通过 tcp_client_flag 的 bit7 来判断是否有数据要发送,因为 LWIP 中处理数据用的是 pbuf 结构体组成的链表,因此如果有数据要发送的话就将发送缓冲区 tcp_client_sendbuf 中的待发送数据放进 pbuf 链表中,这个我们使用 pbuf_take 来实现这个过程,然后我们调用 tcp_client_senddata()函数将数据发送出去,发送完成以后记得将 tcp_client_flag 的 bit7 清零。

```

err_t tcp_client_poll(void *arg, struct tcp_pcb *tpcb)
{
    err_t ret_err;
    struct tcp_client_struct *es;
    es=(struct tcp_client_struct*)arg;
    if(es!=NULL) //连接处于空闲可以发送数据
    {
        if(tcp_client_flag&(1<<7)) //判断是否有数据要发送
        {
            es->p=pbuf_alloc(PBUF_TRANSPORT,
strlen((char*)tcp_client_sendbuf),PBUF_POOL); //申请内存
            //将 tcp_client_sendbuf[]中的数据拷贝到 es->p_tx 中
            pbuf_take(es->p,(char*)tcp_client_sendbuf,strlen((char*)tcp_client_sendbuf));
            //将 tcp_client_sendbuf[]里面复制给 pbuf 的数据发送出去
            tcp_client_senddata(tpcb,es);
            tcp_client_flag&=~(1<<7); //清除数据发送标志
            if(es->p)pbuf_free(es->p); //释放内存
        }else if(es->state==ES_TCPCLIENT_CLOSING)
        {
            tcp_client_connection_close(tpcb,es);//关闭 TCP 连接
        }
        ret_err=ERR_OK;
    }else
    {
        tcp_abort(tpcb);//终止连接,删除 pcb 控制块
        ret_err=ERR_ABRT;
    }
    return ret_err;
}

```

tcp_client_sent()函数为控制块中的 sent 字段的回调函数,这个函数中主要调用了我们下面要讲的 tcp_client_senddata()这个函数, tcp_client_sent()函数代码如下。

//lwip tcp_sent 的回调函数(当从远端主机接收到 ACK 信号后发送数据)

```
err_t tcp_client_sent(void *arg, struct tcp_pcb *tpcb, u16_t len)
{
    struct tcp_client_struct *es;
    LWIP_UNUSED_ARG(len);
    es=(struct tcp_client_struct*)arg;
    if(es->p)tcp_client_senddata(tpcb,es);//发送数据
    return ERR_OK;
}
```

tcp_client_senddata()函数用来发送数据，在这个函数中我们使用 tcp_write()函数将要发送的数据加入到发送缓冲队列中，最后调用 tcp_output()函数将发送缓冲队列中的数据发送出去，这个函数的代码如下。

//此函数用来发送数据

```
void tcp_client_senddata(struct tcp_pcb *tpcb, struct tcp_client_struct * es)
{
    struct pbuf *ptr;
    err_t wr_err=ERR_OK;
    while((wr_err==ERR_OK)&&es->p&&(es->p->len<=tcp_sndbuf(tpcb)))
    {
        ptr=es->p;
        //将要发送的数据加入到发送缓冲队列中
        wr_err=tcp_write(tpcb,ptr->payload,ptr->len,1);
        if(wr_err==ERR_OK)
        {
            es->p=ptr->next;          //指向下一个 pbuf
            if(es->p)pbuf_ref(es->p);  //pbuf 的 ref 加一
            pbuf_free(ptr);          //释放 ptr
        }else if(wr_err==ERR_MEM)es->p=ptr;
        tcp_output(tpcb);           //将发送缓冲队列中的数据立即发送出去
    }
}
```

tcp_client_connection_close()函数的功能是关闭与服务器的连接，通过调用 tcp_abort()函数来关闭与服务器的连接，然后注销掉控制块中的回调函数，将 tcp_client_flag 的 bit5 置 1，标记连接断开，tcp_client_connection_close()函数代码如下。

//关闭与服务器的连接

```
void tcp_client_connection_close(struct tcp_pcb *tpcb, struct tcp_client_struct * es)
{
    //移除回调
    tcp_abort(tpcb);//终止连接,删除 pcb 控制块
    tcp_arg(tpcb,NULL);
    tcp_recv(tpcb,NULL);
    tcp_sent(tpcb,NULL);
    tcp_err(tpcb,NULL);
    tcp_poll(tpcb,NULL,0);
}
```

```

if(es)mem_free(es);
tcp_client_flag&=~(1<<5);//标记连接断开了
}

```

最后我们来重点介绍一下 `tcp_client_test()` 函数，本实验的主要内容就是这个函数来实现的，这个函数和我们上一章讲的 `udp_demo_test()` 函数功能类似。首先我们通过调用 `tcp_client_set_remoteip()` 来设置远端 IP 地址，然后就是在 LCD 上显示一些提示信息，接下来就是本函数的重点，代码如下(`tcp_client_test()`的一部分)。

```

tcppcb=tcp_new(); //创建一个新的 pcb
if(tcppcb)          //创建成功
{
    IP4_ADDR(&rmtipaddr,lwipdev.remoteip[0],lwipdev.remoteip[1],\
    lwipdev.remoteip[2],lwipdev.remoteip[3]);
    //连接到目的地址的指定端口上,当连接成功后回调 tcp_client_connected()函数
    tcp_connect(tcppcb,&rmtipaddr,TCP_CLIENT_PORT,tcp_client_connected);
}
else res=1;
while(res==0)
{
    key=KEY_Scan(0);
    if(key==WKUP_PRES)break;
    if(key==KEY0_PRES)//KEY0 按下了,发送数据
    {
        tcp_client_flag|=1<<7;//标记要发送数据
    }
    if(tcp_client_flag&1<<6)//是否收到数据?
    {
        LCD_Fill(30,250,lcddev.width-1,lcddev.height-1,WHITE);//清上一次数据
        //显示接收到的数据
        LCD_ShowString(30,250,lcddev.width-30,lcddev.height-230,16,tcp_client_recvbuf);
        tcp_client_flag&=~(1<<6);//标记数据已经被处理了.
    }
    if(tcp_client_flag&1<<5)//是否连接上?
    {
        //提示消息
        LCD_ShowString(30,210,lcddev.width-30,lcddev.height-190,16,\
        "STATUS:Connected  ");
        POINT_COLOR=RED;
        //提示消息
        LCD_ShowString(30,230,lcddev.width-30,lcddev.height-190,16,"Receive Data:");
        POINT_COLOR=BLUE;//蓝色字体
    }
    else if((tcp_client_flag&1<<5)==0)
    {
        LCD_ShowString(30,210,190,16,16,"STATUS:Disconnected");
        LCD_Fill(30,230,lcddev.width-1,lcddev.height-1,WHITE);//清屏
    }
}

```

```

    }
    lwip_periodic_handle();
    delay_ms(2);
    t++;
    if(t==200)
    {
        if( (tcp_client_flag&1<<5)==0)//未连接上,则尝试重连
        {
            tcp_client_connection_close(tcpcb,0);//关闭连接
            tcpcb=tcp_new(); //创建一个新的 pcb
            if(tcpcb)          //创建成功
            {
                //连接到目的地址的指定端口上,当连接成功后回调 tcp_client_connected()
                tcp_connect(tcpcb,&rmtipaddr,TCP_CLIENT_PORT,tcp_client_connected);
            }
        }
        t=0;
        LED0=!LED0;
    }
}
tcp_client_connection_close(tcpcb,0);//关闭 TCP Client 连接

```

在上面代码中我们完成了一下几个功能:

1、通过 tcp_new() 创建一个 TCP 的 PCB 控制块

2、当 pcb 控制块创建成功以后就调用 tcp_connect() 连接到目的地址的指定端口上, 如果 pcb 控制块创建失败的话就让 res 为 1。

3、当 res 为 0 的话就进入 while() 循环中, 按下 KEY_UP 键退出循环, 按下 KEY0 键向电脑发送数据。通过判断 tcp_client_flag 的 bit6 来确定是否接收到数据, 如果接收到数据的话就在 LCD 上显示出来, 在 while() 循环中如果检测到 TCP 客户端未连接上服务器的话就通过调用 tcp_connect() 函数来尝试连接服务器。

4、这点特别重要! 在 while 循环中一定要调用 lwip_periodic_handle() 函数, 如果不调用这个函数那么 LWIP 的内核就不能运行, 那么网络肯定不会工作!

至此, tcp_client_demo.c 文件就讲完了, 接下来就是编写 main 函数, main 函数基本和 UDP 实验的相同, main 函数代码如下, 实验完整工程为“[网络实验 5 RAW_TCP 客户端实验](#)”。

```

int main(void)
{
    u8 key;
    delay_init();          //延时初始化
    NVIC_Configuration();  //中断分组配置
    uart_init(115200);     //串口波特率设置
    usmart_dev.init(84);   //初始化 USMART
    LED_Init();            //LED 初始化
    KEY_Init();            //按键初始化
    LCD_Init();            //LCD 初始化
}

```



```

FSMC_SRAM_Init();      //初始化外部 SRAM

mymem_init(SRAMIN);    //初始化内部内存池
mymem_init(SRAMEX);    //初始化外部内存池
mymem_init(SRAMCCM);   //初始化 CCM 内存池

POINT_COLOR = RED;     //红色字体
lwip_test_ui(1);        //加载前半部分 UI
TIM3_Int_Init(999,839); //100khz 的频率,计数 1000 为 10ms
while(lwip_comm_init()) //lwip 初始化
{
    LCD_ShowString(30,150,200,20,16,"LWIP Init Falied!");
    delay_ms(1200);
    LCD_Fill(30,110,230,130,WHITE); //清除显示
    LCD_ShowString(30,110,200,16,16,"Retrying...");
}
LCD_ShowString(30,110,200,20,16,"LWIP Init Success!");
LCD_ShowString(30,130,200,16,16,"DHCP IP configing..."); //等待 DHCP 获取
#if LWIP_DHCP
    //等待 DHCP 获取成功/超时溢出
    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0XFF)) {
        lwip_periodic_handle();
    }
#endif
lwip_test_ui(2);        //加载后半部分 UI
delay_ms(500);
delay_ms(500);
tcp_client_test();      //TCP Client 模式
while(1)
{
    key = KEY_Scan(0);
    if(key == KEY1_PRES) //按 KEY1 键建立连接
    {
        if((tcp_client_flag & 1<<5)) printf("TCP 客户端连接已经建立,不能重复连接\r\n");
        //如果连接成功,不做任何处理
        else tcp_client_test(); //当断开连接后,调 tcp_client_test()函数
    }
    delay_ms(10);
}
}

```

4.3 下载验证

在代码编译成功以后，我们下载代码到开发板中，通过网线连接开发板到路由器上，如果

没有路由器的话就连接到电脑端的 RJ45 上,电脑端还要进行设置,设置过程和 UDP 实验一样。打开网络调试助手软件设置为如图 4.3.1 所示。

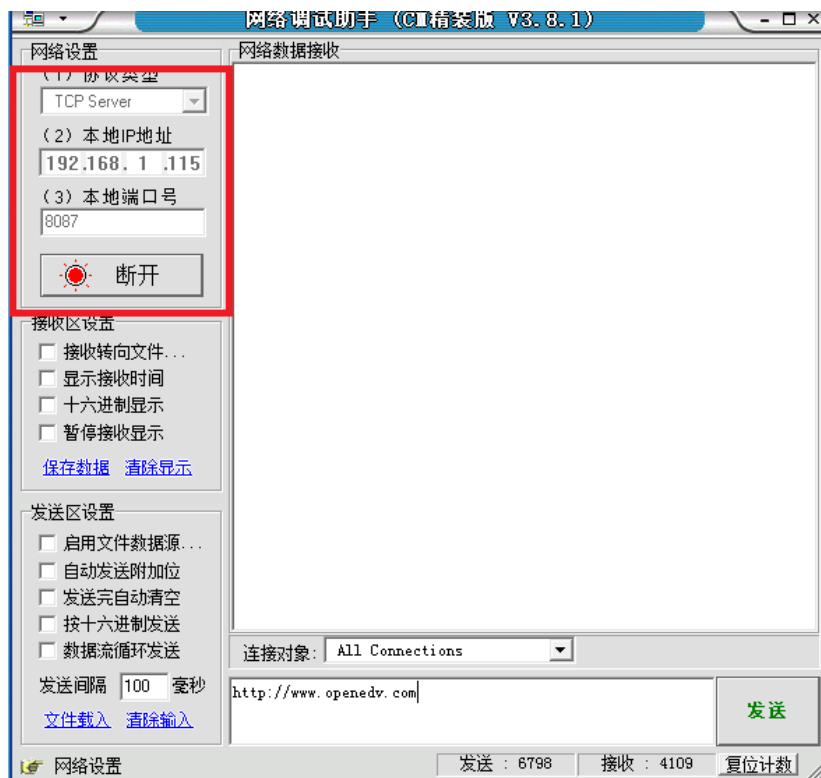


图 4.3.1 网络调试助手设置

开发板上电,等待出现 4.3.2 所示画面,我们设置远端 IP 地址为电脑的 IP 地址,也就是图 4.3.1 中的本地 IP 地址,设置好以后按 KEY_UP 键确认,确认后进入图 4.3.3 所示界面,当 STATUS 为 Connected 的时候就可以和网络调试助手互相发送数据了。

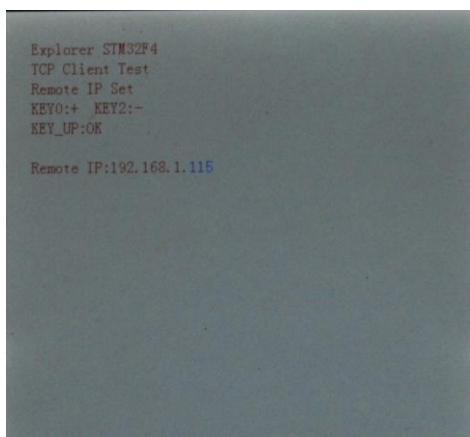


图 4.3.2 设置服务器 IP 地址

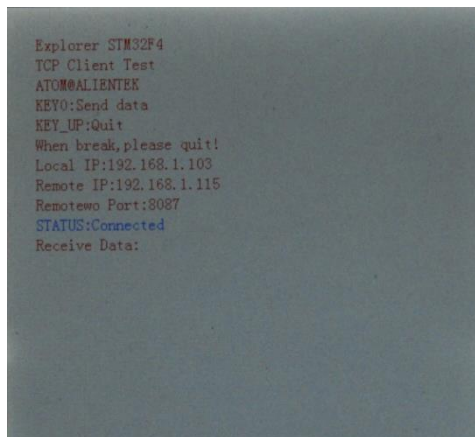


图 4.3.3 连接到服务器

我们通过网络调试助手向开发板发送: <http://www.openedv.com>, 此时开发板 LCD 上显示接收到的数据如图 4.3.4 所示, 按下 KEY0 键向网络调试助手发送数据, 网络调试助手接收到数据如图 4.3.5 所示。

```
Explorer STM32F4
TCP Client Test
ATOM@ALIENTEK
KEY0:Send data
KEY_UP:Quit
When break,please quit!
Local IP:192.168.1.103
Remote IP:192.168.1.115
Remotewo Port:8087
STATUS:Connected
Receive Data:
http://www.openedv.com
```

图 4.3.4 LCD 显示接收到的数据

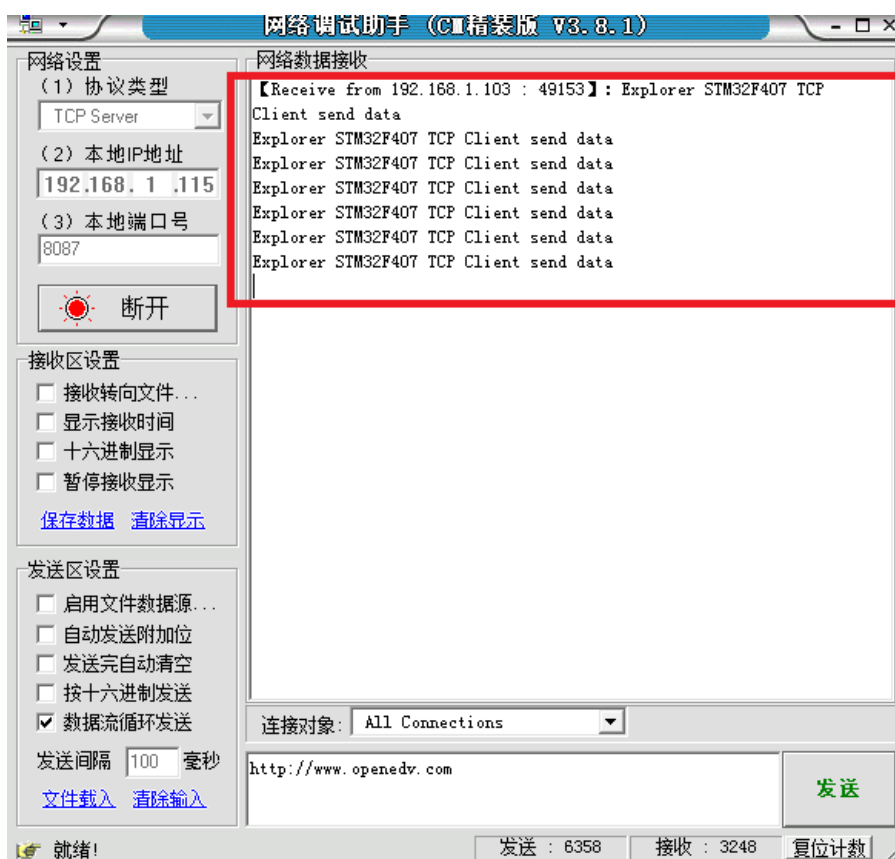


图 4.3.5 网络调试助手显示接收到的数据

第五章 RAW 编程接口 TCP 服务器实验

本章采用 RAW API 编程接口完成开发板和电脑之间的 TCP 通信。在本章中开发板做 TCP 服务器，网络调试助手做 TCP 客户端，实验中我们通过电脑端的网络调试助手给开发板发送数据，开发板接收并在 LCD 上显示接收到的数据，同时也可以通过按键从开发板向网络调试助手发送数据。本章分为如下几个部分：

- 5.1 RAW 编程接口 TCP 简介
- 5.2 软件设计
- 5.3 下载验证

5.1 RAW 编程接口 TCP 简介

在上一章 RAW 编程接口的 TCP 客户端实验中我们已经讲解过了 TCP 的基础知识，这里就不做讲解。

5.2 软件设计

上一章中我们简单的介绍了几个 LWIP 中关于 TCP 的函数，本节中我们就用这几个函数编写我们本章的例程，本章实验的目标是 PC 端和开发板通过 TCP 协议连接起来，开发板做 TCP 服务器，PC 端的网络调试助手配置成客户端。网络调试助手连接到开发板服务器，网络调试助手向开发板发送数据并在 LCD 上显示接收到的数据，我们也可以通过开发板上的按键发送数据给 PC。本章实验中我们主要有两个文件 tcp_server_demo.c 和 tcp_server_demo.h。

tcp_server_demo.h 文件很简单，这里就不讲解，我们重点讲解一下 tcp_server_demo.c 这个文件，在 tcp_server_demo.c 文件中我们一共定义了 9 个函数，如表 5.2.1 所示。

TCP 客户端函数	函数说明
tcp_server_test()	TCP 服务器测试程序
tcp_server_accept()	控制块 accept 字段的回调函数
tcp_server_recv()	控制块 recv 字段的回调函数
tcp_server_error()	控制块 errf 字段的回调函数
tcp_server_poll(p)	控制块 poll 字段的回调函数
tcp_server_sent()	控制块 sent 字段的回调函数
tcp_server_senddata()	发送数据
tcp_server_connection_close()	关闭 TCP 连接
tcp_server_remove_timewait()	删除处于等待状态的 pcb 控制块

表 5.2.1 TCP 服务器函数

tcp_server_accept()函数为控制块 accept 字段的回调函数，当一个侦听和其他主机连接上以后调用，在这个函数中我们主要是为控制块的相应字段注册回调函数，函数的代码如下。

//lwIP tcp_accept()的回调函数

```
err_t tcp_server_accept(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    err_t ret_err;
    struct tcp_server_struct *es;
    LWIP_UNUSED_ARG(arg);
    LWIP_UNUSED_ARG(err);
    tcp_setprio(newpcb, TCP_PRIO_MIN); //设置新创建的 pcb 优先级
    es = (struct tcp_server_struct *) mem_malloc(sizeof(struct tcp_server_struct)); //分配内存
    if (es != NULL) //内存分配成功
    {
        es->state = ES_TCPSERVER_ACCEPTED; //接收连接
        es->pcb = newpcb;
        es->p = NULL;

        tcp_arg(newpcb, es);
        tcp_recv(newpcb, tcp_server_recv); //初始化 tcp_recv()的回调函数
```

```

tcp_err(newpcb,tcp_server_error);    //初始化 tcp_err()回调函数
tcp_poll(newpcb,tcp_server_poll,1);  //初始化 tcp_poll 回调函数
tcp_sent(newpcb,tcp_server_sent);    //初始化发送回调函数

tcp_server_flag|=1<<5;                //标记有客户端连上了
lwipdev.remoteip[0]=newpcb->remote_ip.addr&0xff;    //IADDR4
lwipdev.remoteip[1]=(newpcb->remote_ip.addr>>8)&0xff;    //IADDR3
lwipdev.remoteip[2]=(newpcb->remote_ip.addr>>16)&0xff;    //IADDR2
lwipdev.remoteip[3]=(newpcb->remote_ip.addr>>24)&0xff;    //IADDR1
ret_err=ERR_OK;
}else ret_err=ERR_MEM;
return ret_err;
}

```

tcp_server_recv()函数和上一章 TCP 客户端实验的 tcp_client_recv()函数功能基本差不多,大家可以对照这两个函数的源码看一下,这里就不做讲解了。

tcp_server_error()为当出现重大错误的时候的回调函数,在 TCP 客户端实验中我们没有实现这个函数,在本章实验中我们将这个函数的参数通过串口打印出来,当然也可以根据自己的实际情况来实现这个函数。

tcp_server_poll(), tcp_server_sent()和 tcp_server_senddata()这三个函数分别和上一章 TCP 客户端实验中的 tcp_client_poll(), tcp_client_sent()和 tcp_client_senddata()函数功能类似,大家可以参考 TCP 客户端实验中关于这三个函数的讲解。

tcp_server_connection_close()函数用来关闭 TCP 连接,我们通过调用 tcp_close()函数来关闭连接,注意这里和 TCP 客户端实验的不同,然后就是注销掉控制块中的回调函数,最后释放内存,清零 tcp_server_flag 的 bit5, 标记连接断开,函数代码如下。

```

//关闭 tcp 连接
void tcp_server_connection_close(struct tcp_pcb *tpcb, struct tcp_server_struct *es)
{
    tcp_close(tpcb);
    tcp_arg(tpcb,NULL);
    tcp_sent(tpcb,NULL);
    tcp_recv(tpcb,NULL);
    tcp_err(tpcb,NULL);
    tcp_poll(tpcb,NULL,0);
    if(es)mem_free(es);
    tcp_server_flag&=~(1<<5);//标记连接断开了
}

```

最后一个函数是 tcp_server_remove_timewait()这个函数用来强制删除处于 TIME-WAITI 状态的控制块,函数代码如下。

```

//强制删除 TCP Server 主动断开时的 time wait
void tcp_server_remove_timewait(void)
{
    struct tcp_pcb *pcb,*pcb2;
    while(tcp_active_pcbs!=NULL)

```

```

{
    lwip_periodic_handle();//继续轮询
    delay_ms(10);//等待 tcp_active_pcbs 为空
}
pcb=tcp_tw_pcbs;
while(pcb!=NULL)//如果有等待状态的 pcbs
{
    tcp_pcb_purge(pcb);
    tcp_tw_pcbs=pcb->next;
    pcb2=pcb;
    pcb=pcb->next;
    memp_free(MEMP_TCP_PCB,pcb2);
}
}

```

接下来是很实验最重要的函数：`tcp_server_test()`，同 UDP 实验和 TCP 客户端实验一样，这个函数一开始也是显示一些提示信息，不过不同的是 `tcp_server_test()` 函数不用设置需要连接的远端主机的 IP 地址，因为服务器是等待其他主机来连接的。显示完提示信息以后就是本函数的重点了，代码如下。

```

tcppcbnew=tcp_new(); //创建一个新的 pcb
if(tcppcbnew)          //创建成功
{
    //将本地 IP 与指定的端口号绑定在一起,IP_ADDR_ANY 为绑定本地所有的 IP 地址
    err=tcp_bind(tcppcbnew,IP_ADDR_ANY,TCP_SERVER_PORT);
    if(err==ERR_OK) //绑定完成
    {
        tcppcbconn=tcp_listen(tcppcbnew);          //设置 tcppcb 进入监听状态
        tcp_accept(tcppcbconn,tcp_server_accept); //初始化 LWIP 的 tcp_accept 的回调函数
    }else res=1;
}
POINT_COLOR=BLUE;//蓝色字体
while(res==0)
{
    key=KEY_Scan(0);
    if(key==WKUP_PRES)break;
    if(key==KEY0_PRES)//KEY0 按下了,发送数据
    {
        tcp_server_flag|=1<<7;//标记要发送数据
    }
    if(tcp_server_flag&1<<6)//是否收到数据?
    {
        LCD_Fill(30,210,lcddev.width-1,lcddev.height-1,WHITE);//清上一次数据
        //显示接收到的数据
        LCD_ShowString(30,210,lcddev.width-30,lcddev.height-210,16,tcp_server_recvbuf);
    }
}

```



```

        tcp_server_flag&=~(1<<6);//标记数据已经被处理了.
    }
    if(tcp_server_flag&1<<5)//是否连接上?
    {
        sprintf((char*)tbuf,"ClientIP:%d.%d.%d.%d",lwipdev.remoteip[0],\
        lwipdev.remoteip[1],lwipdev.remoteip[2],lwipdev.remoteip[3]);//客户端 IP
        LCD_ShowString(30,170,230,16,16,tbuf);
        POINT_COLOR=RED;
        //提示消息
        LCD_ShowString(30,190,lcddev.width-30,lcddev.height-190,16,"Receive Data:");
        POINT_COLOR=BLUE;//蓝色字体
    }else if((tcp_server_flag&1<<5)==0)
    {
        LCD_Fill(30,170,lcddev.width-1,lcddev.height-1,WHITE);//清屏
    }
    lwip_periodic_handle();
    delay_ms(2);
    t++;
    if(t==200)
    {
        t=0;
        LED0=!LED0;
    }
}
tcp_server_connection_close(tcpcbnew,0);//关闭 TCP Server 连接
tcp_server_connection_close(tcpcbconn,0);//关闭 TCP Server 连接
tcp_server_remove_timewait();

```

在上面代码中主要完成一下几个功能。

1、通过调用 `tcp_new` 函数创建一个 `tcp` 控制块 `tcpcbnew`，这个控制块用来进行监听，如果未创建成功的话就令 `res` 等于 1。

2、当控制块 `tcpcbnew` 创建成功以后就将其绑定到指定的 IP 地址和端口号上，绑定成功以后将控制块设置为监听状态，并且注册控制块 `accept` 字段的回调函数，如果绑定未成功的话就让 `res` 等于 1。

3、当 `res` 等于 0 的话就进入 `while()` 循环，在 `while` 循环的处理过程基本和 TCP 客户端的一样。

4、当从 `while` 循环退出来后，我们就关闭 TCP 连接，这里我们要关闭两个：`tcpcbnew` 和 `tcpcbconn`，最后还要调用 `tcp_server_remove_timewait()` 函数将处于 TIME—WAIT 状态的 `pcb` 控制块删除。

到这里 `tcp_server_demo.c` 文件中的函数已经讲完，接下来就是编写 `main` 函数了，`main` 函数的代码如下，实验完整工程为“[网络实验 6 RAW_TCP 服务器实验](#)”。

```

int main(void)
{
    u8 key;

```

```

delay_init();           //延时初始化
NVIC_Configuration();   //中断分组配置
uart_init(115200);       //串口波特率设置
usmart_dev.init(84);     //初始化 USMART
LED_Init();             //LED 初始化
KEY_Init();             //按键初始化
LCD_Init();             //LCD 初始化
FSMC_SRAM_Init();       //初始化外部 SRAM

mymem_init(SRAMIN);      //初始化内部内存池
mymem_init(SRAMEX);      //初始化外部内存池
mymem_init(SRAMCCM);     //初始化 CCM 内存池

POINT_COLOR = RED;      //红色字体
lwip_test_ui(1);         //加载前半部分 UI
TIM3_Int_Init(999,839);  //100khz 的频率,计数 1000 为 10ms
while(lwip_comm_init())  //lwip 初始化
{
    LCD_ShowString(30,150,200,20,16,"LWIP Init Falied!");
    delay_ms(1200);
    LCD_Fill(30,110,230,130,WHITE); //清除显示
    LCD_ShowString(30,110,200,16,16,"Retrying...");
}
LCD_ShowString(30,110,200,20,16,"LWIP Init Success!");
LCD_ShowString(30,130,200,16,16,"DHCP IP configing..."); //等待 DHCP 获取
#ifdef LWIP_DHCP
    //等待 DHCP 获取成功/超时溢出
    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0XFF)) {
        lwip_periodic_handle();
    }
#endif
lwip_test_ui(2);         //加载后半部分 UI
delay_ms(500);           //延时 1s
delay_ms(500);
tcp_server_test();       //TCP Server 模式
while(1)
{
    key = KEY_Scan(0);
    if(key == KEY1_PRES)  //按 KEY1 键建立连接
    {
        //如果连接成功,不做任何处理
        if((tcp_server_flag & 1<<6)) printf("UDP 连接已经建立,不能重复连接\r\n");
        else tcp_server_test(); //当断开连接后,调用 udp_demo_test()函数
    }
}

```

```
}  
    delay_ms(10);  
}  
}
```

main 函数的代码很简单，和前面 UDP，TCP 客户端实验的差不多，这里就不做讲解了。

5.3 下载验证

在代码编译成功以后，我们下载代码到开发板中，通过网线连接开发板到路由器上，如果没有路由器的话就连接到电脑端的 RJ45 上，电脑端还要进行设置，设置过程请参照第一章。开发板上电，等待出现 5.3.1 所示画面，打开网络调试助手按图 5.3.2 所示设置好以后点击“连接”按钮。

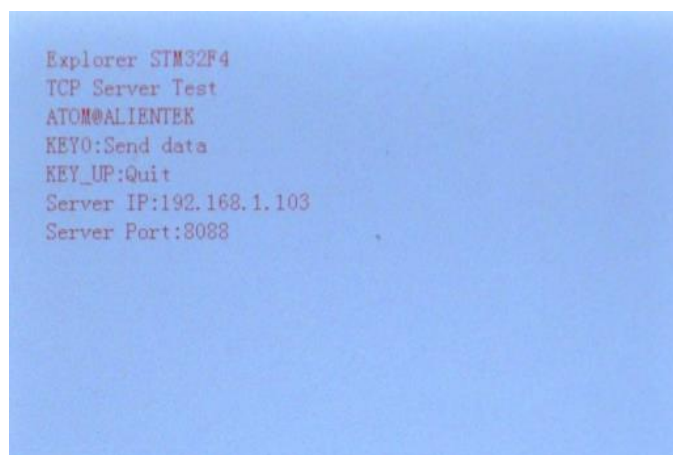


图 5.3.1 开机 LCD 显示画面

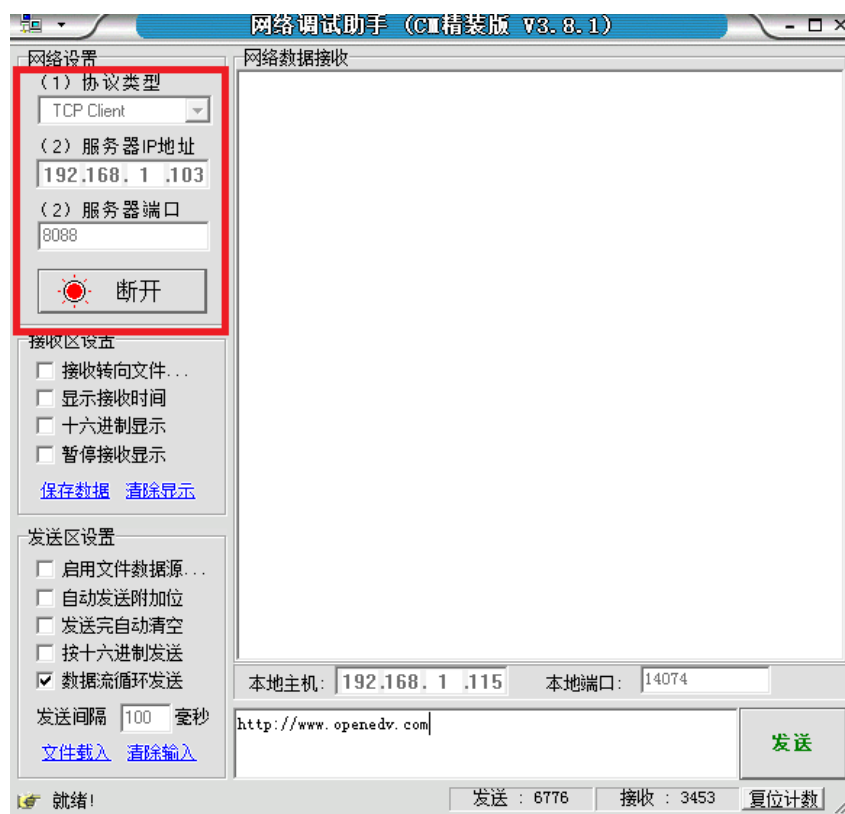


图 5.3.2 网络调试助手设置

当网络调试助手连接上开发板以后，开发板 LCD 上显示如图 5.3.3 所示，表明网络调试助手已经连接上服务器。

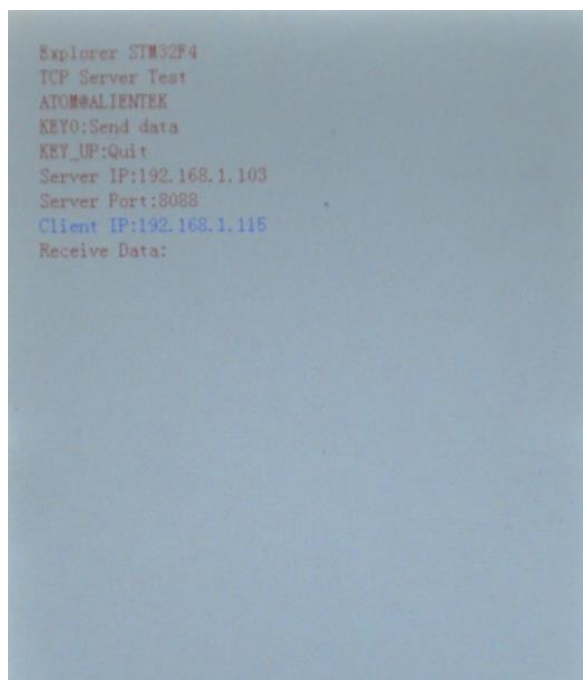


图 5.3.3 连接上以后的 LCD 显示

我们通过网络调试助手向开发板发送：<http://www.openedv.com>，此时开发板 LCD 上显示接收到的数据如图 5.3.4 所示，按下开发板上的 KEY0 键向网络调试助手发送数据，网络调试助手接收到数据如图 5.3.5 所示。

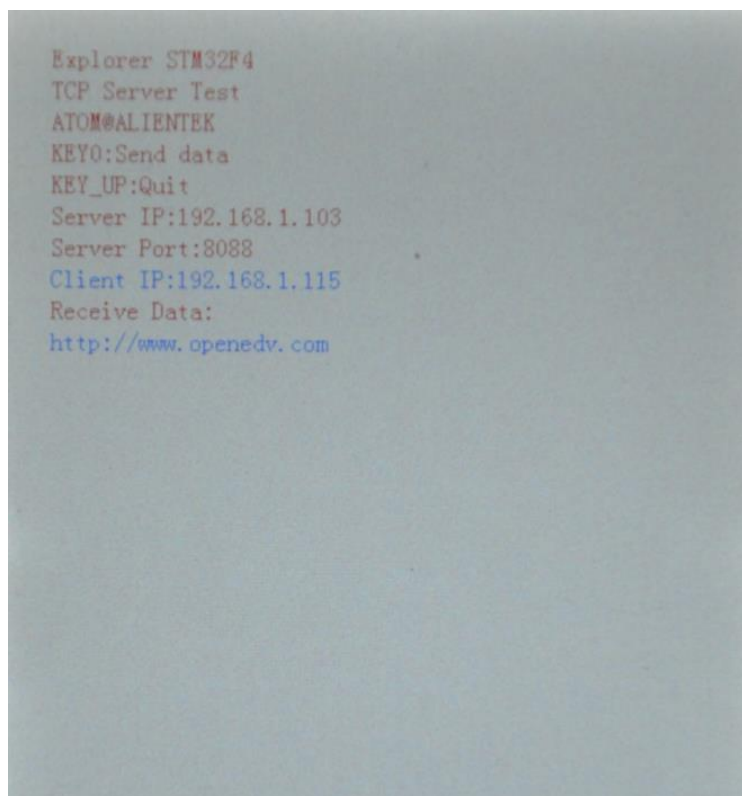


图 5.3.4 LCD 显示接收到的数据

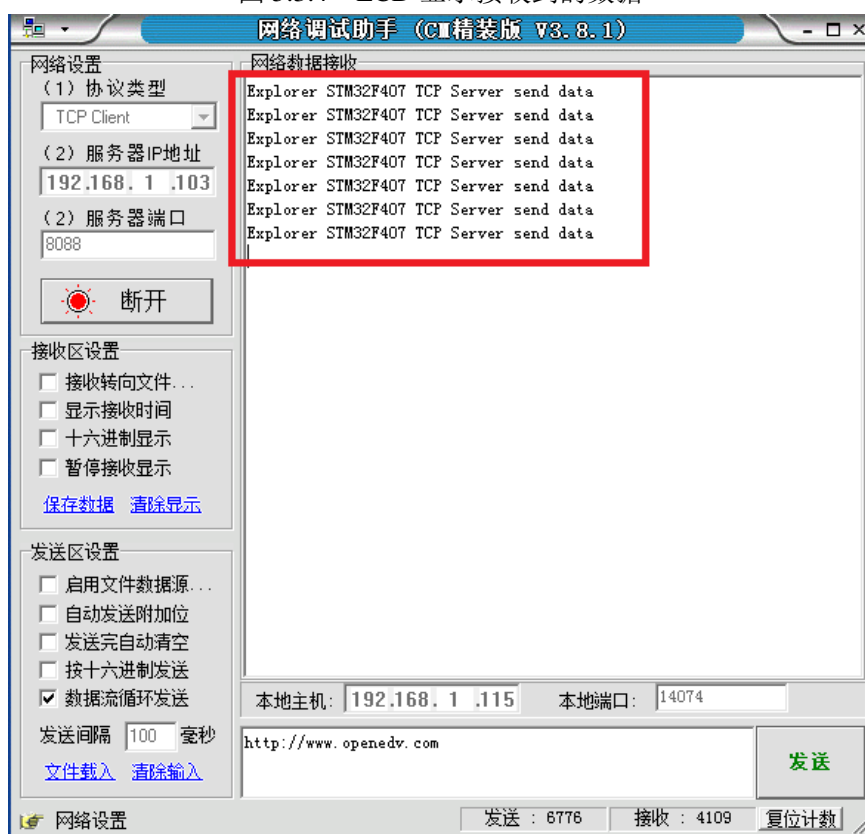


图 5.3.5 网络调试助手显示接收到的数据

第六章 RAW 编程接口 Web Server 实验

本章采用 RAW API 编程接口在开发板上实现一个 Web Server。在本章中我们通过移植并修改 ST 官方的一个 Web Server 的例程来展示如何在 STM32F4 开发板上实现 Web Server 服务器。我们可以通过在浏览器中输入开发板的 IP 地址来访问开发板，这是开发板会返回一个网页，我们可以通过网页控制开发板的 LED 灯和蜂鸣器，可以查看 ADC 和内部温度传感器的值，也可以查看 RTC 时钟，本章分为如下几个部分：

- 6.1 Web Server 文件以及相关技术简介
- 6.2 软件设计
- 6.3 下载验证

6.1 Web Server 文件以及相关技术简介

1) 实验相关文件简介

本章我们在 ST 官方的 Web Server 例程的基础上完成本章的实验，打开我们的 Web Server 实验文件夹下的 web_server_demo 文件，如图 6.1.1 所示。

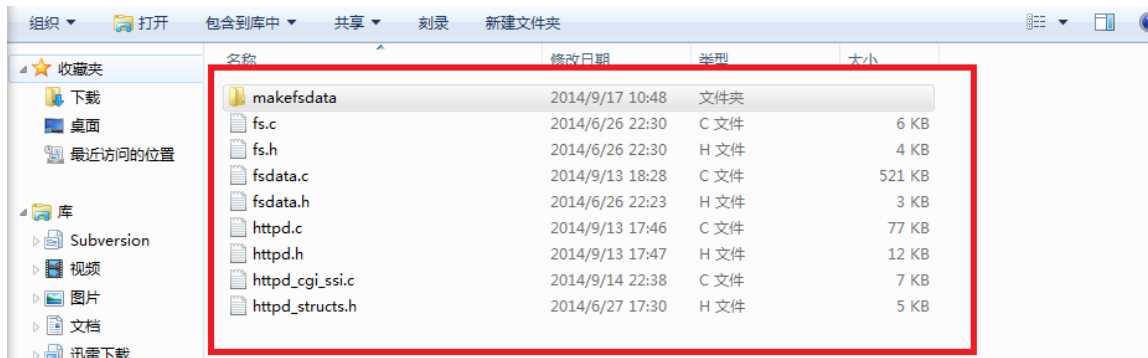


图 6.1.1 Web Server 实验所需文件

图 6.1.1 中的各个文件的讲解见表 6.1.1。

表 6.1.1 Web Server 各文件说明

文件	说明
makefsdata	这个文件中包含有原始网页文件和将原始的网页文件转换成网页数组的工具 makefsdata.exe。
fs.c	这两个文件用来管理生成的网页数组
fs.h	
fsdata.c	生成的网页数组
fsdata.h	
httpd.c	Http Server 的源代码，本实验的核心文件，这两个文件完成了将开发板配置成 Web Serve 的工作
httpd.h	
httpd_cgi_ssi.c	CGI 和 SSI 源文件，我们通过网页和开发板交互主要是这个文件中的函数完成的

2) 网页数组制作

在 makefsdata 文件夹下的 fs 文件为网页源文件，如图 6.1.2 所示，这里大家可以根据自己实际情况制作网页，关于网页的制作不属于本教程的内容，大家自行查阅相关教材。

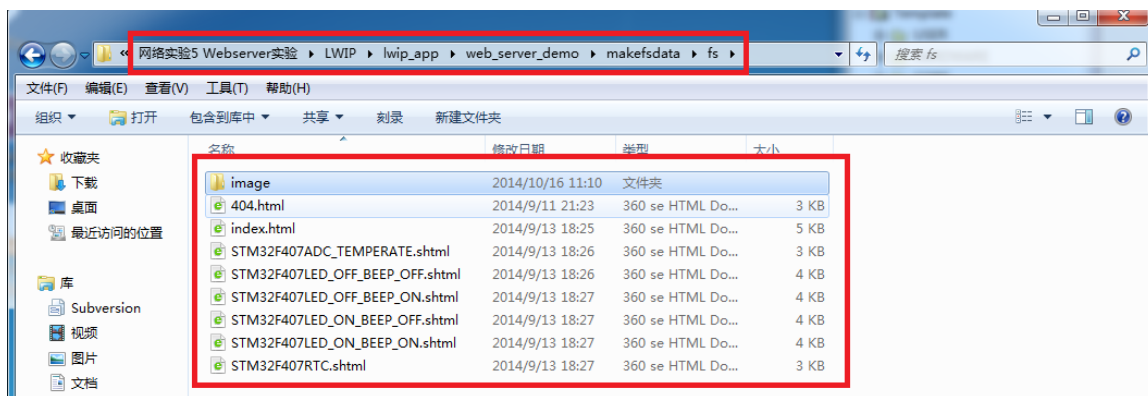


图 6.1.2 本实验网页源文件

图 6.1.2 中的是网页源文件，显然这个文件不能直接放到 STM32 里面，我们要做一个转换，

使其可以放到 STM32 里面。这里我们通过 makefsdata.exe 这个工具将原始网页文件转换成.c 格式的网页数组,这样就可以添加到我们的工程中了,下面我们讲解 makefsdata 工具的使用。

3) makefsdata 工具的使用

makefsdata 工具是用来将我们编辑网页文件等信息转换成二进制数的一个工具。接下来我们讲解一下这个工具的使用方法

1、我们新建一个名为 fs 的文件夹,将我们编辑好的网页源文件放到 fs 文件夹下,fs 文件夹内容如图 6.1.3 所示,里面包含了我们编辑好的 html 和.shtml 等网页文件,其中 image 文件夹里面是我们使用到的图片。

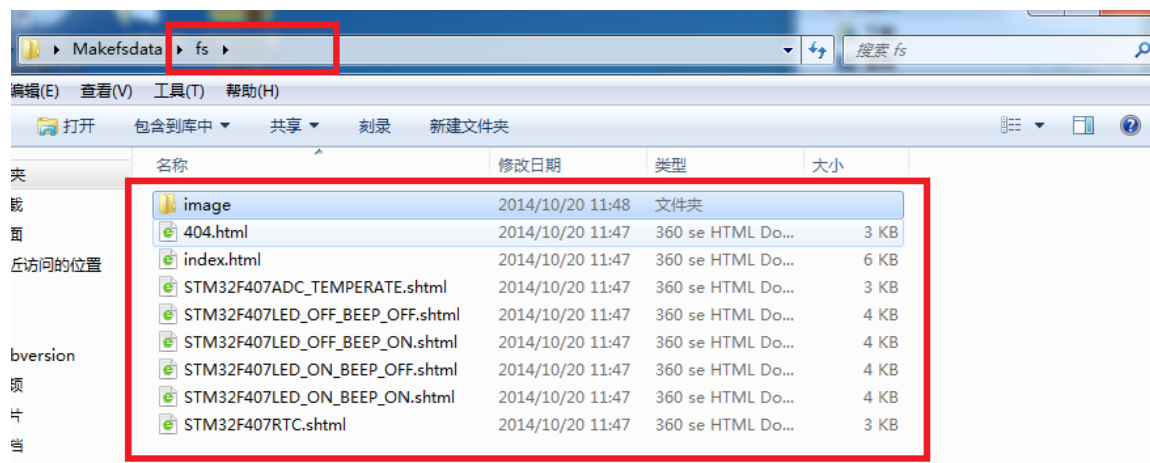


图 6.1.3fs 文件内容

2、将 fs 文件夹和 makefsdata.exe 工具放到同一文件夹下,此处为 makefsdata 文件夹,打开 makefsdata 文件夹,如图 6.1.4 所示。其中图 6.1.4 中的 echotool.exe 和 Tftpd32-3.51-setup.exe 为其他工具,这里没有使用到,cmd.reg 稍后我们会讲到。



图 6.1.4 makefsdata 文件内容

3、在 makefsdata 文件夹上点击鼠标右键,然后点击”在此位置打开 CMD”选项,打开后结果如图 6.1.5 所示。此时会打开一个 CMD 命令窗口。如果点击右键没有”在此位置打开 CMD”选型的话,请使用我们提供的 cmd.reg 文件导入注册表注册,双击打开 cmd.reg,然后一路确定下去就可以了。

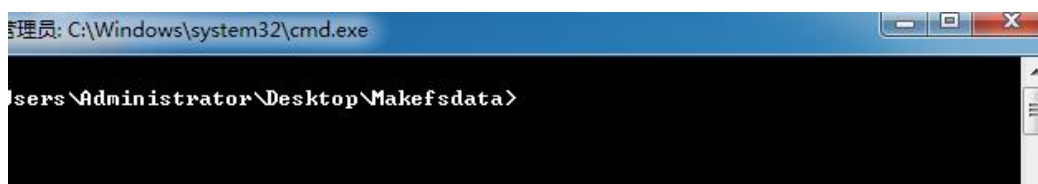


图 6.1.5 点击”在此位置打开 CMD”

4、在打开的 CMD 命令窗口中输入：makefsdata -i 命令，如图 6.1.6 所示。

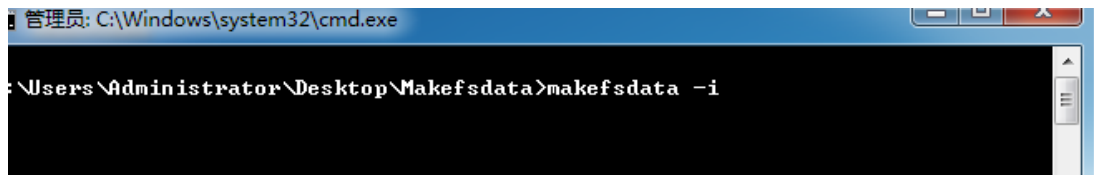


图 6.1.6 输入 makefsdata -i 命令

5、按回车键。命令窗如图 6.1.7 所示。

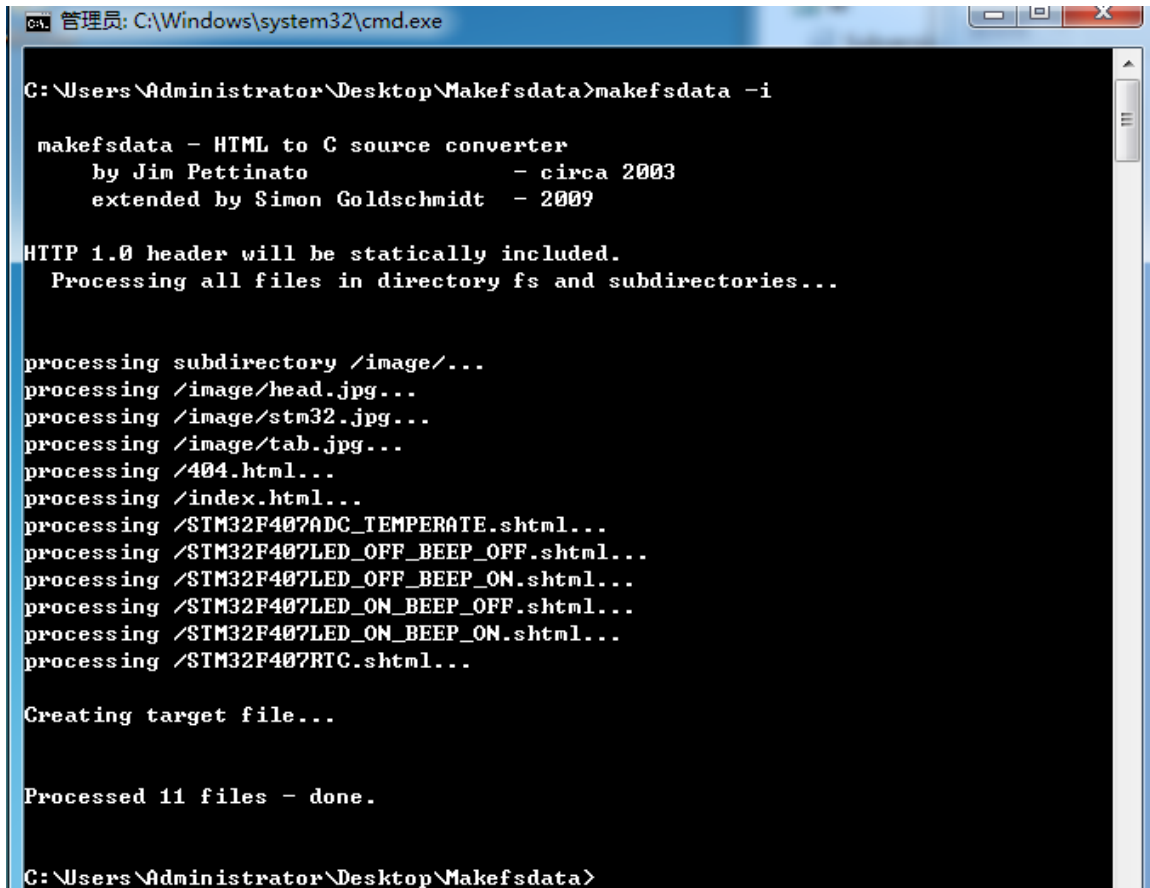


图 6.1.7 按回车键后的命令窗

6、打开 makefsdata 文件夹，打开后如图 6.1.8 所示，我们发现在 makefsdata 文件夹下多了一个 fsdata.c 的 C 文件，这个 fsdata.c 文件就是我们转换后的二进制网页文件，到此 makefsdata 工具的用法介绍完毕



图 6.1.8 生成的 fsdata.c 文件

4) CGI 技术简介

公共网关接口 CGI(Common Gateway Interface) 是 WWW 技术中最重要的技术之一, 有着不可替代的重要地位。CGI 是外部应用程序与 Web 服务器之间的接口标准, 是在 CGI 程序和 Web 服务器之间传递信息的规程。CGI 规范允许 Web 服务器执行外部程序, 并将它们的输出发送给 Web 浏览器, CGI 在物理上是一段程序, 运行在服务器上, 提供同客户端 HTML 页面的接口。

绝大多数的 CGI 程序被用来解释处理来自表单的输入信息, 并在服务器产生相应的处理, 或将相应的信息反馈给浏览器, CGI 程序使网页具有交互功能。在我们本章实验中我们通过浏览器控制开发板上的 LED 和蜂鸣器就是使用的 CGI 技术。

5) SSI 技术简介

服务器端嵌入: Server Side Include, 是一种类似于 ASP 的基于服务器的网页制作技术。大多数的 WEB 服务器等均支持 SSI 命令。将内容发送到浏览器之前, 可以使用“服务器端包含 (SSI)”指令将文本、图形或应用程序信息包含到网页中。例如, 可以使用 SSI 包含时间/日期戳、版权声明或供客户填写并返回的表单。对于在多个文件中重复出现的文本或图形, 使用包含文件是一种简便的方法。将内容存入一个包含文件中即可, 而不必将内容输入所有文件。通过一个非常简单的语句即可调用包含文件, 此语句指示 Web 服务器将内容插入适当网页。而且, 使用包含文件时, 对内容的所有更改只需在一个地方就能完成。因为包含 SSI 指令的文件要求特殊处理, 所以必须为所有 SSI 文件赋予 SSI 文件扩展名。默认扩展名是 .stm、.shtm 和 .shtml。

SSI 是为 WEB 服务器提供的一套命令, 这些命令只要直接嵌入到 HTML 文档的注释内容之中即可。如: `<!--#include file="info.htm"-->` 就是一条 SSI 指令, 其作用是将“info.htm”的内容拷贝到当前的页面中, 当访问者来浏览时, 会看到其它 HTML 文档一样显示 info.htm 其中的内容。其它的 SSI 指令使用形式基本同刚才的举例差不多, 可见 SSI 使用只是插入一点代码而已, 使用形式非常简单。`<!-- -->` 是 HTML 语法中表示注释, 当 WEB 服务器不支持 SSI 时, 会忽略这些信息。

在本实验中我们可以通过网页查看开发板的 ADC, 内部温度传感器和 RTC 的值就是通过 SSI 来实现的。

6.2 软件设计

本实验对应的网络例程为“网络实验 7 RAW_WebServer 实验”、“网络实验 14 NETCONN_Webserver 实验 (UCOSII 版本)”和“网络实验 15 NETCONN_WebServer 实验 (UCOSIII 版本)”, 不同之处是网络实验 7 是无操作系统的, 网络实验 14 和 15 分别为带 UCOSII 和 UCOSIII 操作系统的, 但是 WebServer 的实现方法都是相同的。

我们打开 Web Server 实验的工程, 如图 6.2.1 所示, 其中 fs.c 文件管理生成的网页数组文件这个文件由 ST 提供。httpd.c 文件是本章实验的重点, 这个文件将开发板配置为 Web Server, 这个文件也由 ST 官方提供的, 阅读这个文件需要有网页相关的知识, 这里对这个文件不做讲解。

我们在浏览器中输入网址, 服务器就会返回给我们相应的网页, 然后浏览器解析并呈现给我们。同样的, 当我们通过浏览器访问开发板的时候, 开发板这时是作为服务器的, 服务器针对不同的 URL 在 fsdata.c 文件中找出相应的网页, 并且返回给浏览器, 在 fsdata.c 文件中查找网页的过程就需要 fs.c 里面的函数。接收浏览器发送的数据并且将网页返回给浏览器的过程都是由 httpd.c 文件里面的函数来完成的。

fs.c 和 httpd.h 文件本章不做讲解, 感兴趣的朋友可以去看一下, 本章中我们主要讲解的是

httpd_cgi_ssi.c 这个文件, 这个文件中讲解了如何使用 CGI 和 SSI 技术完成浏览器与服务器的交互。

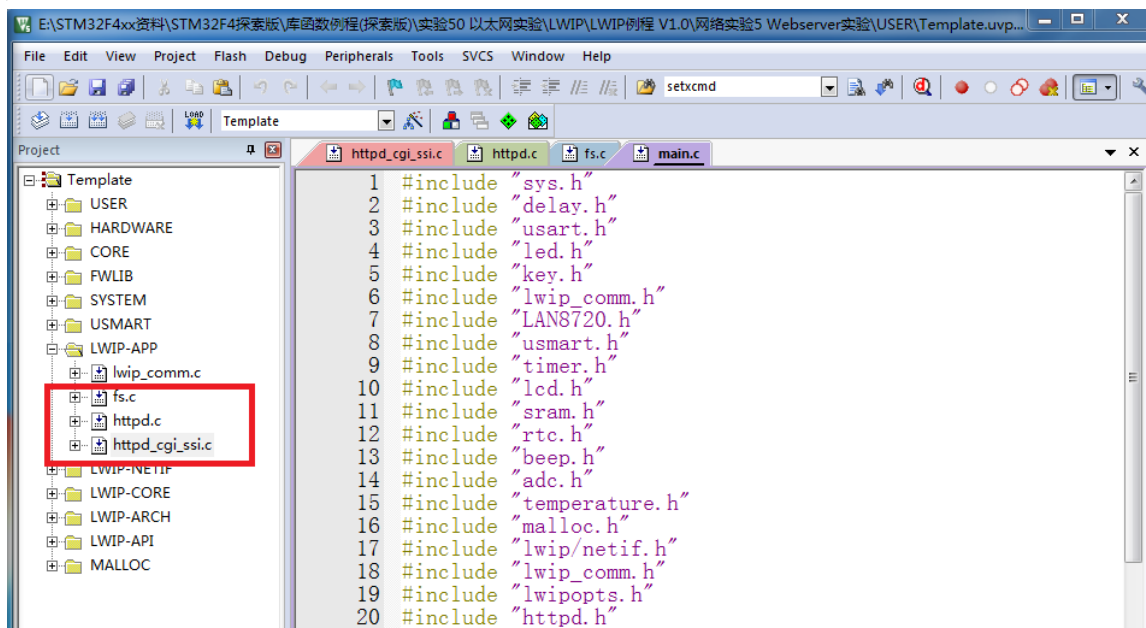


图 6.2.1 Web Server 工程文件

1) CGI 实现

我们通过浏览器控制开发板上的 LED 灯和蜂鸣器就是使用的 CGI 技术, 我们在开发板上对浏览器发送过来的 URL 进行分析, 然后根据不同的 URL 调用不同的程序就可以了, 下图 6.2.2 中我们控制 LED1 灯亮, 注意图中的 URL。



图 6.2.2 打开 LED1

上图中的 URL 为: `http://192.168.1.101/leds.cgi?LED1=LED1ON&button2=SEND`。我们就是分析这一串字符串来做出相应的处理, 其中 `leds.cgi` 表示为控制 LED 灯的 CGI, 后面的“LED1”为变量, LED1ON 为变量“LED1”的值。我们根据字符串“leds.cgi”调用处理 LED 灯的程序,

然后根据后面的变量和变量值来决定是打开还是关闭 LED 灯。在 `httpd_cgi_ssi.c` 中我们定义了一个数组 `ppcURLs`，数组如下，在这个数组中 `leds.cgi` 对应的 `LEDS_CGI_Handler` 处理程序，`beep.cgi` 对应的是 `BEEP_CGI_Handler` 处理程序。

```
static const tCGI ppcURLs[] = //cgi 程序
{
    {"/leds.cgi",LEDS_CGI_Handler},
    {"/beep.cgi",BEEP_CGI_Handler},
};
```

`LEDS_CGI_Handler` 是一个函数，代码如下，从下面代码中可以看出我们是根据变量“LED1”的值来做相应的处理，当为 `LED1ON` 的时候就打开 LED1，当为 `LED1OFF` 的时候就关闭 LED1。那么蜂鸣器 `BEEP` 的处理过程也一样的，这里就不做讲解。最后我们还要初始化 CGI 句柄，初始化函数为 `httpd_cgi_init()`，这个函数很简单。

```
//CGI LED 控制句柄
const char* LEDS_CGI_Handler(int iIndex, int iNumParams, char *pcParam[], char *pcValue[])
{
    u8 i=0; //注意根据自己的 GET 的参数多少来选择 i 值范围
    iIndex = FindCGIPParameter("LED1",pcParam,iNumParams); //找到 led 的索引号
    //只有一个 CGI 句柄 iIndex=0
    if (iIndex != -1)
    {
        LED1=1; //关闭 LED1 灯
        for (i=0; i<iNumParams; i++) //检查 CGI 参数
        {
            if (strcmp(pcParam[i], "LED1")==0) //检查参数"led" 属于控制 LED1 灯的
            {
                if(strcmp(pcValue[i], "LED1ON")==0) //改变 LED1 状态
                    LED1=0; //打开 LED1
                else if(strcmp(pcValue[i], "LED1OFF") == 0)
                    LED1=1; //关闭 LED1
            }
        }
    }
    if(LED1 == 0 && BEEP == 0)          return "/STM32F407LED_ON_BEEP_OFF.shtml";
    //LED1 开,BEEP 关

    else if(LED1 == 0 && BEEP == 1) return "/STM32F407LED_ON_BEEP_ON.shtml";
    else if(LED1 == 1 && BEEP == 1) return "/STM32F407LED_OFF_BEEP_ON.shtml";
    else return "/STM32F407LED_OFF_BEEP_OFF.shtml";
}
```

2) SSI 实现

我们通过网页查看开发板上的 ADC 值，内部温度传感器和 RTC 时间的时候就是用的 SSI，每隔 1s 刷新一次网页，然后通过 SSI 技术将这些值嵌入到网页中，这样我们就看到时钟在动态的跟新，如图 6.2.3 所示。

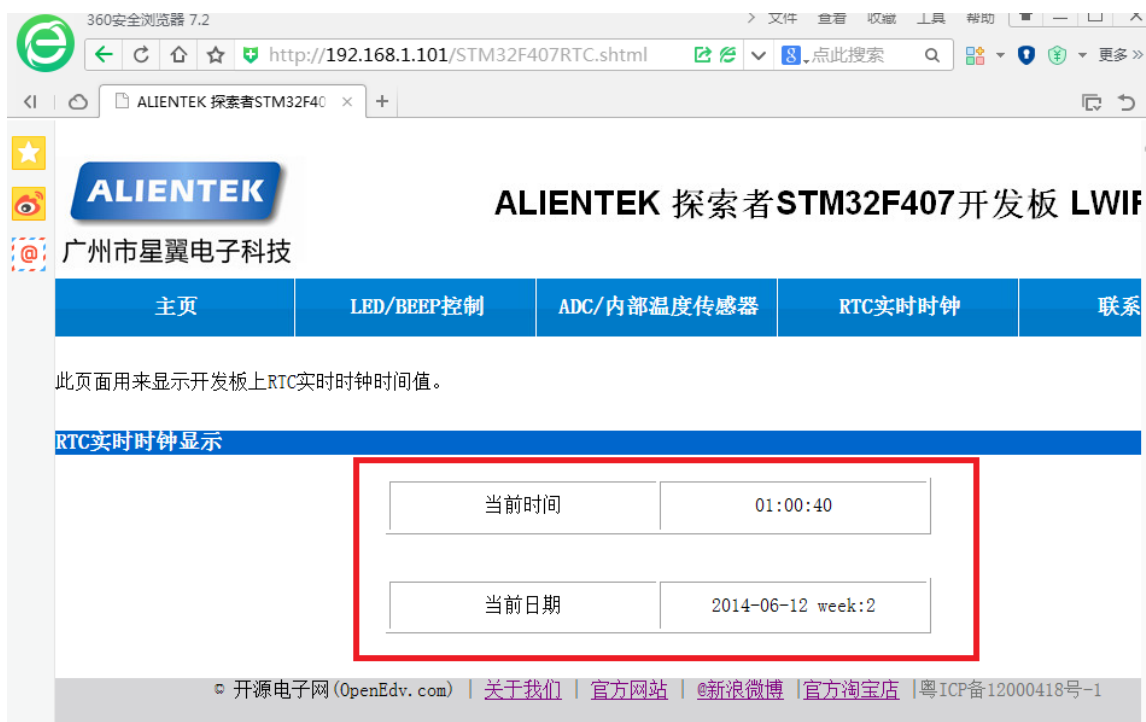


图 6.2.3 网页显示 RTC 时间

SSIHandler 函数为 SSI 的句柄函数，函数代码如下，在这个函数中我们根据参数 iIndex 调用不同的函数来完成向网页中添加数据，这几个函数比较简单，大家自行查阅一下。最后我们还要初始化 SSI 句柄，初始化函数为 httpd_ssi_init()，这个函数很简单。

//SSI 的 Handler 句柄

```
static u16_t SSIHandler(int iIndex, char *pcInsert, int iInsertLen)
{
    switch(iIndex)
    {
        case 0:
            ADC_Handler(pcInsert);
            break;
        case 1:
            Temperate_Handler(pcInsert);
            break;
        case 2:
            RTCTime_Handler(pcInsert);
            break;
        case 3:
            RTCdate_Handler(pcInsert);
            break;
    }
    return strlen(pcInsert);
}
```

最后就是 main 函数的编写，main 函数代码如下，在主函数中我们调用 httpd_init() 函数初

始化 Web Server，在 while() 循环中我们一定要调用 lwip_periodic_handle() 函数。

```
int main(void)
{
    u32 i;
    delay_init();           //延时初始化
    NVIC_Configuration();   //中断分组配置
    uart_init(115200);      //串口波特率设置
    usmart_dev.init(84);    //初始化 USMART
    LED_Init();             //LED 初始化
    KEY_Init();             //按键初始化
    LCD_Init();             //LCD 初始化
    BEEP_Init();            //蜂鸣器初始化
    RTC_Timer_Init();       //RTC 初始化
    Adc_Init();             //ADC1_CH5 初始化
    Adc_Temperate_Init();   //内部温度传感器初始化
    FSMC_SRAM_Init();       //初始化外部 SRAM

    mymem_init(SRAMIN);     //初始化内部内存池
    mymem_init(SRAMEX);     //初始化外部内存池
    mymem_init(SRAMCCM);    //初始化 CCM 内存池

    POINT_COLOR = RED;     //红色字体
    lwip_test_ui(1);        //加载前半部分 UI
    TIM3_Int_Init(999,839); //100khz 的频率,计数 1000 为 10ms
    while(lwip_comm_init()) //lwip 初始化
    {
        LCD_ShowString(30,150,200,20,16,"LWIP Init Falied!");
        delay_ms(1200);
        LCD_Fill(30,110,230,130,WHITE); //清除显示
        LCD_ShowString(30,110,200,16,16,"Retrying...");
    }
    LCD_ShowString(30,110,200,20,16,"LWIP Init Success!");
    LCD_ShowString(30,130,200,16,16,"DHCP IP configing..."); //等待 DHCP 获取
#ifdef LWIP_DHCP
    //等待 DHCP 获取成功/超时溢出
    while((lwipdev.dhcpstatus!=2)&&(lwipdev.dhcpstatus!=0XFF)) {
        lwip_periodic_handle();
    }
#endif
    lwip_test_ui(2);        //加载后半部分 UI
    delay_ms(500);          //延时 1s
    delay_ms(500);
    httpd_init();           //Web Server 模式
}
```



```
while(1)
{
    lwip_periodic_handle();
    i++;
    if(i%5000==0)LED0=!LED0;
    delay_us(100);
}
}
```

6.3 下载验证

在代码编译成功以后，我们下载代码到开发板中，通过网线连接开发板到路由器上，如果没有路由器的话就连接到电脑端的 RJ45 上，电脑端还要进行设置，设置过程和 UDP 实验一样。下载完成后等待开发板 LCD 出现如图 6.3.1 所示，然后我们在浏览器里面输入开发板的 IP 地址，我的开发板 IP 地址为：192.168.1.101，大家根据自己的实际情况输入就行了，按回车，服务器将网页返回给浏览器并显示出来，如图 6.3.2 所示。

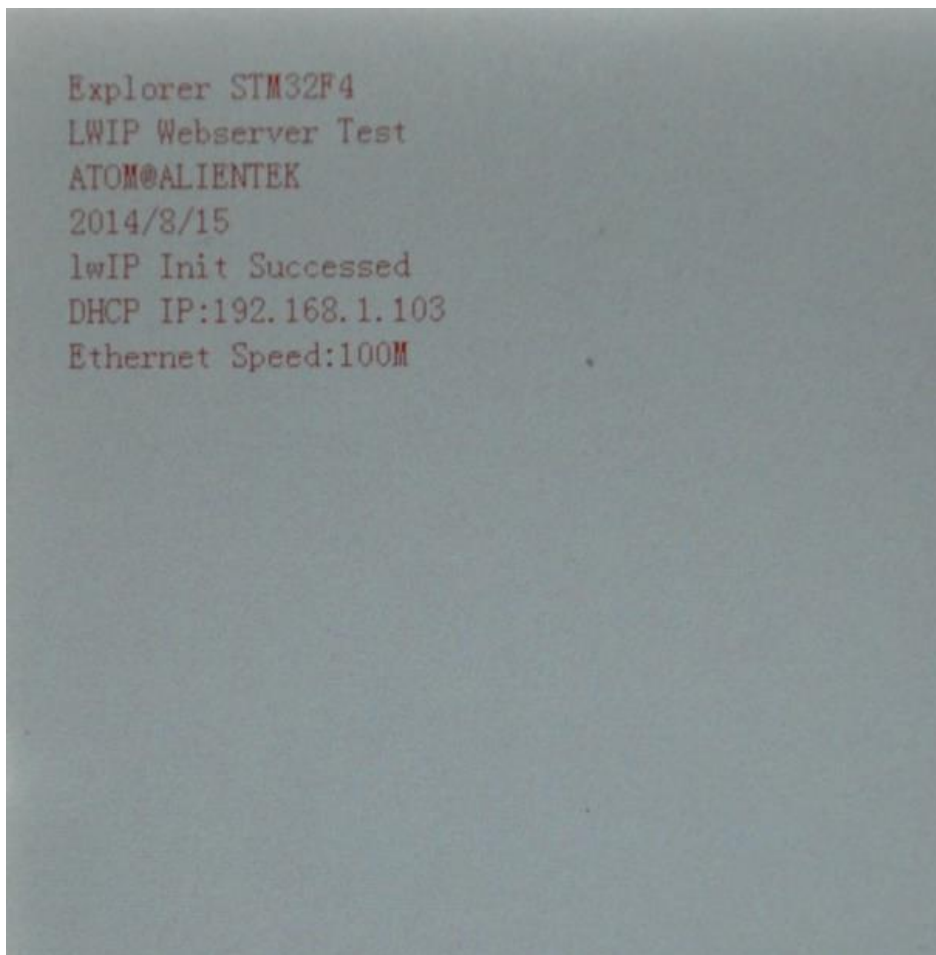


图 6.3.1 LCD 显示界面



图 6.3.2 WEB Server 主页

最后大家可以测试一下其他的功能,比如控制开发板的 LED1 灯,蜂鸣器等,可以查看 ADC 的值和内部温度传感器,查看 RTC 时间值。

第七章 NETCONN 编程接口简介

前几章实验我们都没有使用操作系统，因此采用的是 RAW 编程接口，RAW 编程接口使得程序效率高，但是需要对 LWIP 有深入的了解，而且不适合大数据量等场合。本章我们就讲解一下 NETCONN 编程接口，使用 NETCONN API 时需要有操作系统的支持，我们使用的是 UCOSII 操作系统，本章我们分为以下几部分：

7.1 netbuf 数据缓冲区

7.2 netconn 连接结构

7.3 netconn api 函数

声明：本章内容参考自《嵌入式网络那些事 LWIP 协议深度剖析与实战演练》，作者朱升林！如果有读者想深入了解 LWIP 协议栈内部，建议阅读这本参考书。

7.1 netbuf 数据缓冲区

在前面我们讲过了描述数据包的 pbuf 结构，netbuf 是 NETCONN API 编程接口使用的描述数据包的结构，我们可以使用 netbuf 来管理发送数据、接收数据的缓冲区。有关 netbuf 的详细描述在 netbuf.c 和 netbuf.h 这两个文件中，netbuf 是一个结构体，在 netbuf.h 中定义了这个结构体，代码如下，这里我们去掉了条件编译的代码。

```
struct netbuf
{
    struct pbuf *p, *ptr;
    ip_addr_t addr;
    u16_t port;
};
```

我们可以从上面的代码中看出，p 和 ptr 指向 pbuf 链表，其中 p 和 ptr 都指向 pbuf 链表，不同的是 p 一直指向 pbuf 链表的第一个 pbuf 结构，而 ptr 可能指向链表中其他的位置，netbuf_next() 和 netbuf_first() 操作 ptr 字段。addr 和 port 字段用来记录数据发送方的 IP 地址和端口号，netbuf_fromaddr 和 netbuf_fromport 这两个宏定义用于返回 addr 和 port 这两个字段。netbuf 和 pbuf 之间的关系如图 7.1.1 所示。

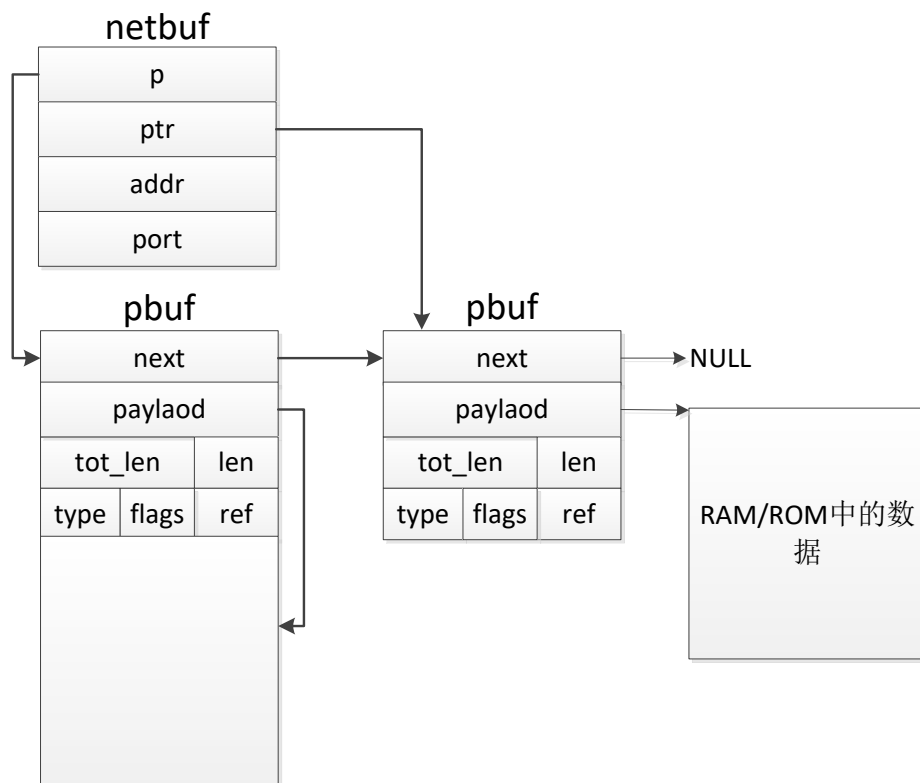


图 7.1.1 用户数据缓冲区

不管是 TCP 连接还是 UDP 连接，接收到数据包后会将数据封装在一个 netbuf 中，然后将这个 netbuf 交给应用程序去处理。在数据发送时，根据不同的连接有不同的处理：对于 TCP 连接，用户只需要提供待发送数据的起始地址和长度，内核会根据实际情况将数据封装在合适大

小的数据包中，并放入发送队列中，对于 UDP 来说，用户需要自行将数据封装在 netbuf 结构中，当发送函数被调用的时候内核直接将数据包中的数据发送出去。

在 netbuf.c 中提供了几个操作 netbuf 的函数，如表 7.1.1 所示。

函数	描述
netbuf_new()	申请一个 netbuf 空间，但是不会分配任何数据空间(不指向任何的 pbuf)，真正的数据存储区域需要调用 netbuf_alloc()函数来分配。
netbuf_delete()	释放一个 netbuf 的内存，如果 netbuf 中的 p 指针上还记录有数据，则相应的 pbuf 也会被释放掉。
netbuf_alloc()	为 netbuf 结构分配指定大小的数据空间，数据空间是通过 pbuf 的形式表现的，函数返回成功分配的数据空间起始地址(pbuf 的 payload 指向的地址)
netbuf_free()	释放 netbuf 中的 p 指向的 pbuf 数据空间内存。
netbuf_ref()	和 netbuf_alloc 类似，不过只是分配一个 pbuf 首部结构(首部结构中包含协议的首部空间)，pbuf 中的 payload 指向参数 dataptr，这种描述数据包的方式在静态数据包的发送时经常用到。
netbuf_chain()	将参数 tail 的 pbuf 连接到参数 head 的 pbuf 的后面，调用此函数后参数 tail 结构会被删除掉。
netbuf_data()	将 netbuf 结构中的 ptr 指针记录的 pbuf 数据起始地址填入参数 dataptr 中，同时将该 pbuf 中的数据长度填入到参数 len 中。
netbuf_next()	将 netbuf 结构的 ptr 指针指向 pbuf 链表中的下一个 pbuf 结构。
netbuf_first()	将 netbuf 结构的 ptr 指针指向 pbuf 链表中的第一个 pbuf 结构。

表 7.1.1 netbuf 操作函数

7.2 netconn 连接结构

我们前面在使用 RAW 编程接口的时候，对于 UDP 和 TCP 连接使用的是两种不同的编程函数：udp_xxx 和 tcp_xxx。NETCONN 对于这两种连接提供了统一的编程接口，用于可以使用同一的连接结构和编程函数，在 api.h 中定义了 netconn 结构体，代码如下。

```
struct netconn
{
    enum netconn_type type;           //连接类型，TCP UDP 或者 RAW
    enum netconn_state state;         //当前连接状态
    union {
        struct ip_pcb *ip;           //IP 控制块
        struct tcp_pcb *tcp;         //TCP 控制块
        struct udp_pcb *udp;         //UDP 控制块
        struct raw_pcb *raw;         //RAW 控制块
    } pcb;
    err_t last_err;                   //此连接上最新的错误
    sys_sem_t op_completed;           //用于两部分 API 同步的信号量
    sys_mbox_t recvmbox;              //接收数据的邮箱
#ifdef LWIP_TCP
    sys_mbox_t acceptmbox;            //用于 TCP 服务器端，连接请求的缓冲队列
#endif
#ifdef LWIP_SOCKET
```

```

    int socket;                //socket 描述符，用于 socket API
#endif
#if LWIP_SO_SNDTIMEO
    s32_t send_timeout;        //发送数据时的超时时间
#endif
#if LWIP_SO_RCVTIMEO
    int rcv_timeout;           //接收数据时的超时时间
#endif
#if LWIP_SO_RCVBUF
    int rcv_bufsize;           //接收消息队列长度
    s16_t rcv_avail;           //数据邮箱 rcvmbox 中已缓存的数据长度
#endif
    u8_t flags;                //标识符
#if LWIP_TCP
//当调用 netconn_write 发送数据但缓存不足的时候，数据会暂时存放在 current_msg 中,等待下一次数据发送，write_offset 记录下一次发送时的索引
    size_t write_offset;
    struct api_msg_msg *current_msg;
#endif
    netconn_callback callback;  //连接相关回调函数，实现 socket API 时使用
};

```

在 api.h 文件中还定义了连接状态和连接类型，这两个都是枚举类型。

```

//枚举类型，用于描述连接类型
enum netconn_type {
    NETCONN_INVALID    = 0,        //无效类型
    NETCONN_TCP         = 0x10,     //TCP
    NETCONN_UDP         = 0x20,     //UDP
    NETCONN_UDPLITE     = 0x21,     //UDPLite
    NETCONN_UDPNOCHKSUM = 0x22,     //无校验 UDP
    NETCONN_RAW         = 0x40      //原始链接
};
//枚举类型，用于描述连接状态，主要用于 TCP 连接中
enum netconn_state
{
    NETCONN_NONE,           //不处于任何状态
    NETCONN_WRITE,          //正在发送数据
    NETCONN_LISTEN,         //侦听状态
    NETCONN_CONNECT,        //连接状态
    NETCONN_CLOSE           //关闭状态
};

```

7.3 netconn api 函数

本节我们就讲解一下 NETCONN 的 API 函数，在文件 api_lib.c 中实现了 NETCONN 的各

个函数，在 `api_lib.c` 中有很多 `netconn` 的函数，其中大部分是 LWIP 内部调用的，我们最后使用的有 9 个函数，如表 7.3.1 所示。

函数	描述
<code>netconn_new()</code>	这个函数其实就是一个宏定义，用来为新连接申请一个连接结构 <code>netconn</code> 空间。
<code>netconn_delete()</code>	该函数的功能是删除一个 <code>netconn</code> 连接结构。
<code>netconn_getaddr()</code>	该函数用来获取一个 <code>netconn</code> 连接结构的源 IP 地址和源端口号或者目的 IP 地址和目的端口号。
<code>netconn_bind()</code>	将一个连接结构与本地 IP 地址和端口号进行绑定。
<code>netconn_connect()</code>	将一个连接结构与目的 IP 地址和端口号进行绑定。
<code>netconn_disconnect()</code>	只能用在 UDP 连接结构中，用来断开与服务器的连接。
<code>netconn_listen()</code>	这个函数就是一个宏定义，只在 TCP 服务器程序中使用，用来将连接结构 <code>netconn</code> 置为侦听状态。
<code>netconn_accept()</code>	这个函数也只用与 TCP 服务器程序中，服务器调用此函数可以获得一个新的连接。
<code>netconn_recv()</code>	从连接的 <code>recvmbx</code> 邮箱中接收数据包，可用于 TCP 连接，也可用于 UDP 连接。
<code>netconn_send()</code>	用于在已建立的 UDP 连接上发送数据。
<code>netconn_write()</code>	用于在稳定的 TCP 连接上发送数据。
<code>netconn_close()</code>	关闭一个 TCP 连接。

图 7.3.1 NETCONN API 函数

`netconn_new()`函数是函数 `netconn_new_with_proto_and_callback()`的宏定义，此函数用来为新连接申请一个 `netconn` 空间，参数为新连接的类型，连接类型在上一节已经讲过了，常用的值是 `NETCONN_UDP` 和 `NETCONN_TCP`，分别代表 UDP 连接和 TCP 连接。

`netconn_delete()`函数用来删除一个 `netconn` 连接结构，如果函数调用时双方仍然处于连接状态，则相应连接将被关闭：对于 UDP 连接，连接立即被关闭，UDP 控制块被删除；对于 TCP 连接，则函数执行主动关闭，内核完成剩余的断开握手过程。

`netconn_getaddr()`函数用来获取一个 `netconn` 连接结构的源 IP 地址和源端口号或者目的 IP 地址和目的端口号，IP 地址保存在 `addr` 中，端口信息保存在 `port` 中，参数 `local` 指明是获取源地址还是目的地址，当 `local` 为 1 时表示本地地址，此函数原型如下。

```
err_t netconn_getaddr(struct netconn *conn, ip_addr_t *addr, u16_t *port, u8_t local);
```

`netconn_bind()`函数将一个连接结构与本地 IP 地址 `addr` 和端口号 `port` 进行绑定，服务器端程序必须执行这一步，服务器必须与指定的端口号绑定才能接受客户端的连接请求，函数原型如下。

```
err_t netconn_bind(struct netconn *conn, ip_addr_t *addr, u16_t port);
```

`netconn_connect()`函数的功能是连接服务器，将指定的连接结构与目的 IP 地址 `addr` 和目的端口号 `port` 进行绑定，当作为 TCP 客户端程序时，调用此函数会产生握手过程，函数原型如下。

```
err_t netconn_connect(struct netconn *conn, ip_addr_t *addr, u16_t port);
```

`netconn_disconnect()`函数只能使用在 UDP 连接中，功能是断开与服务器的连接。对于 UDP 连接来说就是将 UDP 控制块中的 `remote_ip` 和 `remote_port` 字段值清零，函数原型如下。

```
err_t netconn_disconnect (struct netconn *conn);
```

`netconn_listen()`函数只有在 TCP 服务器程序中使用，将一个连接结构 `netconn` 设置为侦听状

态，既将 TCP 控制块的状态设置为 LISTEN 状态。

`netconn_accept()`函数也只用于 TCP 服务器程序，服务器调用此函数可以从 `acceptmbox` 邮箱中获取一个新建立的连接，若邮箱为空，则函数会一直阻塞，直到新连接的到来。服务器端调用此函数前必须先调用 `netconn_listen()`函数将连接设置为侦听状态，函数原型如下。

```
err_t netconn_accept(struct netconn *conn, struct netconn **new_conn);
```

`netconn_recv()`函数是从连接的 `recvmbox` 邮箱中接收数据包，可用于 TDP 连接，也可用于 UDP 连接，函数会一直阻塞，直到从邮箱中获得数据消息，数据被封装在 `netbuf` 中。如果从邮箱中接收到一条空消息，表示对方已经关闭当前的连接，应用程序也应该关闭这个无效的连接，函数原型如下。

```
err_t netconn_recv(struct netconn *conn, struct netbuf **new_buf);
```

`netconn_send()`函数用于在 UDP 连接上发送数据，参数 `conn` 指出了要操作的连接，参数 `buf` 为要发送的数据，数据被封装在 `netbuf` 中。如果 IP 层分片功能未使能，则 `netbuf` 中的数据不能太长，不能超过 MTU 的值，最好不要超过 1000 字节。如果 IP 层分片功能使能的情况下就可以忽略此细节，函数原型如下。

```
err_t netconn_send(struct netconn *conn, struct netbuf *buf);
```

`netconn_write()`函数用于在稳定的 TCP 连接上发送数据，参数 `dataptr` 和 `size` 分别指出了待发送数据的起始地址和长度，函数并不要求用户将数据封装在 `netbuf` 中，对于数据长度也没有限制，内核会直接处理这些数据，将他们封装在 `pbuf` 中，并挂接到 TCP 的发送队列中。

`netconn_close()`函数用来关闭一个 TCP 连接，该函数会产生一个 FIN 握手包的发送，成功后函数便返回，而后剩余的断开握手操作由内核自动完成，用户程序不用关心，该函数只是断开一个连接，但不会删除连接结构 `netconn`，用户需要调用 `netconn_delete()`函数来删除连接结构，否则会造成内存泄漏，函数原型如下。

```
err_t netconn_close(struct netconn *conn);
```

第八章 NETCONN 编程接口 UDP 实验

本章，我们开始学习 NETCONN API 函数的使用，本章实验中我们通过电脑端的网络调试助手给开发板发送数据，开发板接收数据并通过串口将接收到的数据发送到串口调试助手上，也可以通过按键从开发板向网络调试助手发送数据。本章分为如下几个部分：

8.1 软件设计

8.2 下载验证

8.1 软件设计

8.1.1 UCOSII 版本

关于 UDP 的基础知识我们在第三章讲 RAW 编程接口的 UDP 实验时已经讲过了，这里就不重复讲解了。

本章对应着我们网络例程的网络实验 7 例程，打开我们的例程，在 LWIP->lwip_app 下有 udp_demo.c 和 udp_demo.h 两个文件，这两个文件就是我本章实验的源码，在 udp_demo.c 中我们实现了两个函数 udp_thred() 和 udp_demo_init()，在有操作系统的支持下，udp 可以作为一个线程来处理。

udp_thred() 函数为任务函数，udp_demo_init() 为创建一个 UDP 线程，udp_thred() 函数代码如下。

//udp 任务函数

```
static void udp_thread(void *arg)
```

```
{
```

```
    OS_CPU_SR cpu_sr;
```

```
    err_t err;
```

```
    static struct netconn *udpconn;
```

```
    static struct netbuf *recvbuf;
```

```
    static struct netbuf *sendbuf;
```

```
    struct ip_addr destipaddr;
```

```
    u32 data_len = 0;
```

```
    struct pbuf *q;
```

```
    LWIP_UNUSED_ARG(arg);
```

```
① udpconn = netconn_new(NETCONN_UDP); //创建一个 UDP 链接
```

```
② udpconn->recv_timeout = 10;
```

```
    if(udpconn != NULL) //创建 UDP 连接成功
```

```
    {
```

```
③ err = netconn_bind(udpconn,IP_ADDR_ANY,UDP_DEMO_PORT);
```

```
    IP4_ADDR(&destipaddr,lwipdev.remoteip[0],lwipdev.remoteip[1],
```

```
    lwipdev.remoteip[2],lwipdev.remoteip[3]); //构造目的 IP 地址
```

```
④ netconn_connect(udpconn,&destipaddr,UDP_DEMO_PORT); //连接到远端主机
```

```
    if(err == ERR_OK) //绑定完成
```

```
    {
```

```
        while(1)
```

```
        {
```

```
            if((udp_flag & LWIP_SEND_DATA) == LWIP_SEND_DATA) //有数据要发送
```

```
            {
```

```
⑤ sendbuf = netbuf_new();
```

```
    netbuf_alloc(sendbuf,strlen((char *)udp_demo_sendbuf));
```

```
    //指 udp_demo_sendbuf 组
```

```
    sendbuf->p->payload = (char *)udp_demo_sendbuf;
```

```

⑥      err = netconn_send(udpconn,sentbuf);      //将 netbuf 中的数据发送出去
      if(err != ERR_OK)
      {
          printf("发送失败\r\n");
          netbuf_delete(sentbuf);      //删除 buf
      }
      udp_flag &= ~LWIP_SEND_DATA;  //清除数据发送标志
      netbuf_delete(sentbuf);      //删除 buf
    }

⑦      netconn_recv(udpconn,&recvbuf); //接收数据
      if(recvbuf != NULL)      //接收到数据
      {
          OS_ENTER_CRITICAL(); //关中断
          //数据接收缓冲区清零
          memset(udp_demo_recvbuf,0,UDP_DEMO_RX_BUFSIZE);
          for(q=recvbuf->p;q!=NULL;q=q->next) //遍历完整个 pbuf 链表
          {
              //判断要拷贝到 UDP_DEMO_RX_BUFSIZE 中的数据是否大于
              //UDP_DEMO_RX_BUFSIZE 的剩余空间，如果大于
              //的话就只拷贝 UDP_DEMO_RX_BUFSIZE 中剩余长度的数据，
              //否则的话就拷贝所有的数据
              if(q->len > (UDP_DEMO_RX_BUFSIZE-data_len)) \
              memcpy(udp_demo_recvbuf+data_len,q->payload,\
              (UDP_DEMO_RX_BUFSIZE-data_len)); //拷贝数据
              else memcpy(udp_demo_recvbuf+data_len,q->payload,q->len);
              data_len += q->len;
              //超出 TCP 客户端接收数组,跳出
              if(data_len > UDP_DEMO_RX_BUFSIZE) break;
          }
          data_len=0; //复制完成后 data_len 要清零。
          OS_EXIT_CRITICAL(); //开中断
          printf("%s\r\n",udp_demo_recvbuf); //打印接收到的数据
          netbuf_delete(recvbuf);      //删除 buf
      }else OSTimeDlyHMSM(0,0,0,5); //延时 5ms
    }
  }else printf("UDP 绑定失败\r\n");
}else printf("UDP 连接创建失败\r\n");
}

```

①调用 netconn_new()函数申请一个新的 netconn 结构，申请类型为 UDP 连接类型，申请成功后的 netconn 结构为 udpconn。

②netconn_recv()函数会阻塞线程，而我们在 while()循环中还要发送数据，所以不能阻塞线程，因此我们令 udpconn 的 recv_timeout(超时时间)为 10，这样就不会阻塞线程，但是要能使

recv_timeout 字段要在 lwipopts.h 中定义 LWIP_SO_RCVTIMEO，否则不能 recv_timeout 字段。

③将 udpconn 连接结构与本地 IP 地址和端口号绑定, IP_ADDR_ANY 代表任何一个网络接口的 IP 地址。

④将 udpconn 连接结构连接到远端 IP 地址和端口号上。

⑤如果要发送数据的话我们需要申请一个 netbuf 结构来存放数据。

⑥调用 netconn_send()函数在指定的连接上发送数据

⑦调用 netconn_recv()函数接收数据，接收到的数据放进 recvbuf 中。

⑧如果 recvbuf 不为空的话我们就将 recvbuf 中的数据拷贝到数组 udp_demo_recvbuf 中，然后将 udp_demo_recvbuf 中的数据通过串口发送给串口调试助手。

我们定义了一个全局变量 udp_flag 来标记是否有数据发送，当有数据发送的时候 udp_flag 的 bit8 就为 1，通过与 LWIP_SEND_DATA 进行与运算就知道是否有数据要发送，有数据要发送时就调用 netconn_send()函数将 udp_demo_sendbuf 中的数据发送出去，发送失败的话就关闭连接。

udp_demo_init()为创建 UDP 线程，这个函数比较简单，代码如下。

```
//创建 UDP 线程
//返回值:0 UDP 创建成功
//      其他 UDP 创建失败
INT8U udp_demo_init(void)
{
    INT8U res;
    OS_CPU_SR cpu_sr;
    OS_ENTER_CRITICAL(); //关中断
    res=OSTaskCreate(udp_thread,(void*)0,(OS_STK*)&\
        UDP_TASK_STK[UDP_STK_SIZE-1],UDP_PRIO); //创建 UDP 线程
    OS_EXIT_CRITICAL(); //开中断
    return res;
}
```

接下来就是 main 函数的编写，main 函数代码如下。

```
int main(void)
{
    delay_init(168); //延时初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //中断分组配置
    uart_init(115200); //串口波特率设置
    usmart_dev.init(84); //初始化 USMART
    LED_Init(); //LED 初始化
    KEY_Init(); //按键初始化
    LCD_Init(); //LCD 初始化
    FSMC_SRAM_Init(); //SRAM 初始化
    mymem_init(SRAMIN); //初始化内部内存池
    mymem_init(SRAMEX); //初始化外部内存池
    mymem_init(SRAMCCM); //初始化 CCM 内存池

    POINT_COLOR = RED; //红色字体
}
```

```

LCD_ShowString(30,30,200,20,16,"Explorer STM32F4");
LCD_ShowString(30,50,200,20,16,"UDP NETCONN Test");
LCD_ShowString(30,70,200,20,16,"ATOM@ALIENTEK");
LCD_ShowString(30,90,200,20,16,"KEY1:Send data");
LCD_ShowString(30,110,200,20,16,"2014/9/1");
POINT_COLOR = BLUE; //蓝色字体

OSInit();           //UCOS 初始化
while(lwip_comm_init()) //lwip 初始化
{
    LCD_ShowString(30,130,200,20,16,"Lwip Init failed!"); //lwip 初始化失败
    delay_ms(500);
    LCD_Fill(30,130,230,150,WHITE);
    delay_ms(500);
}
LCD_ShowString(30,130,200,20,16,"Lwip Init Success!"); //lwip 初始化成功
while(udp_demo_init()) //初始化 udp_demo(创建 udp_demo 线程)
{
    LCD_ShowString(30,150,200,20,16,"UDP failed!!"); //udp 创建失败
    delay_ms(500);
    LCD_Fill(30,150,230,170,WHITE);
    delay_ms(500);
}
LCD_ShowString(30,150,200,20,16,"UDP Success!"); //udp 创建成功
OSTaskCreate(start_task,(void*)0,(OS_STK*)&\
START_TASK_STK[START_STK_SIZE-1],START_TASK_PRIO);
OSStart(); //开启 UCOS
}

```

在 main 函数中我们首先完成外设的初始化，接下来一次初始化 UCOS，LWIP。然后调用 `udp_demo_init()` 函数来创建一个 UDP 线程，最后创建了一个 `start_task()` 任务，`start_task()` 任务主要用来创建其他任务的。

在 `start_task()` 任务中我们一共创建了 3 个任务，`display_task`、`key_task` 和 `led_task`。`display_task` 任务用来在 LCD 上显示地址和端口号等信息。`key_task` 任务用来发送数据，当按下 KEY1 时令 `udp_flag` 的 bit8 置 1，标记有数据发送，UDP 线程会通过判断 `udp_flag` 的 bit8 状态来发送数据。`led_task` 任务就是简单的让 LED0 闪烁，提示系统正在运行。

最后最重要的一点!!!!

我们需要设置要连接的主机 IP 地址，在前面 RAW 实验中我们通过按键来设置需要连接的远端 IP 地址，在本实验我们没有采用这种方法，因此我们需要在代码中设置，设置很简单，在 `lwip_comm.c` 文件下有一个 `lwip_comm_default_ip_set()` 函数，此函数用来设置地址信息。我们通过这个函数来设置要连接的远端 IP 地址，`lwip_comm_default_ip_set()` 函数代码如下。

```

//lwip 默认 IP 设置
//lwipx:lwip 控制结构体指针
void lwip_comm_default_ip_set(__lwip_dev *lwipx)

```

```

{
    u32 sn0;
    sn0=(vu32*)(0x1FFF7A10);//获取 STM32 的唯一 ID 的前 24 位作为 MAC 地址后三字节
    //默认远端 IP 为:192.168.1.115
    lwipx->remoteip[0]=192;
    lwipx->remoteip[1]=168;
    lwipx->remoteip[2]=1;
    lwipx->remoteip[3]=115;
    //MAC 地址设置(高三字节固定为:2.0.0,低三字节用 STM32 唯一 ID)
    lwipx->mac[0]=2;//高三字节(IEEE 称之为组织唯一 ID,OUI)地址固定为:2.0.0
    lwipx->mac[1]=0;
    lwipx->mac[2]=0;
    lwipx->mac[3]=(sn0>>16)&0XFF;//低三字节用 STM32 的唯一 ID
    lwipx->mac[4]=(sn0>>8)&0XFFF;;
    lwipx->mac[5]=sn0&0XFF;
    //默认本地 IP 为:192.168.1.30
    lwipx->ip[0]=192;
    lwipx->ip[1]=168;
    lwipx->ip[2]=1;
    lwipx->ip[3]=30;
    //默认子网掩码:255.255.255.0
    lwipx->netmask[0]=255;
    lwipx->netmask[1]=255;
    lwipx->netmask[2]=255;
    lwipx->netmask[3]=0;
    //默认网关:192.168.1.1
    lwipx->gateway[0]=192;
    lwipx->gateway[1]=168;
    lwipx->gateway[2]=1;
    lwipx->gateway[3]=1;
    lwipx->dhcpstatus=0;//没有 DHCP
}

```

上面代码中红色字段就是设置远端 IP 地址的，将自己电脑的 IP 地址填写进去就行了，例如我现在电脑 IP 地址就是 192.168.1.115，这点一定要注意！

8.1.2 UCOSIII 版本

UCOSIII 版本的代码和 UCOSII 版本的代码基本一致的，只是其中的某些 API 函数命名不同，原理上都是相通的。具体代码查看例程“[网络实验 9 NETCONN_UDP 实验\(UCOSIII 版本\)](#)”，实验的原理详解请查看 8.1.1 小节。

8.2 下载验证

代码编译完成后就可以下载到 STM32F407 开发板中，开发板连接路由器，没有路由器的话就连接到电脑上，然后按照我们在第一章中讲解的方法设置电脑。我们需要打开网络调试助

手，串口调试助手。复位开发板，等待开发板 LCD 显示如图 8.2.1 所示信息，在图 8.2.1 中显示了开发板的 IP 地址，子网掩码，默认网关，端口号等信息。

```
Explorer STM32F4
UDP NETCONN Test
ATOM@ALIENTEK
KEY1:Send data
2014/9/1
Lwip Init Success!
UDP Success!
DHCP IP :192.168.1.103
DHCP GW :192.168.1.1
NET MASK:255.255.255.0
Port:8089!
```

图 8.2.1LCD 显示。

我们在来看一下串口调试助手如图 8.2.2 所示，在串口调试助手上也输出了我们开发板的 IP 地址，子网掩码、默认网关等信息。

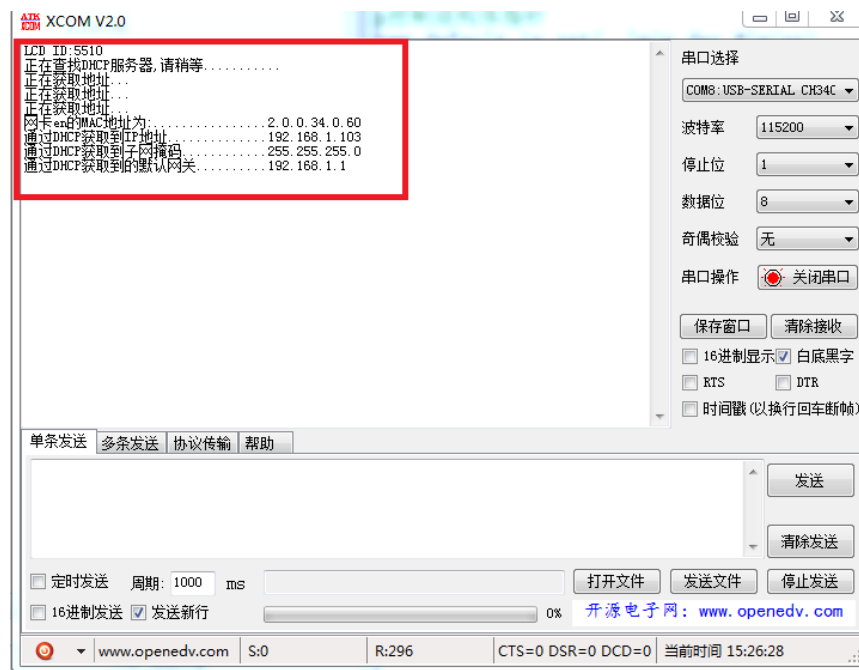


图 8.2.2 串口调试助手

我们通过网络调试助手和开发板互相发送数据，结果如图 8.2.3 所示。从图中我们可以看出开发板接收到网络调试助手发送过来的数据: <http://www.openedv.com>。开发板接收到数据后通过串口将接收到的数据发送给串口调试助手。网络调试助手接收到开发板发送来数据: Explorer

STM32F407 NETCONN UDP demo send data。

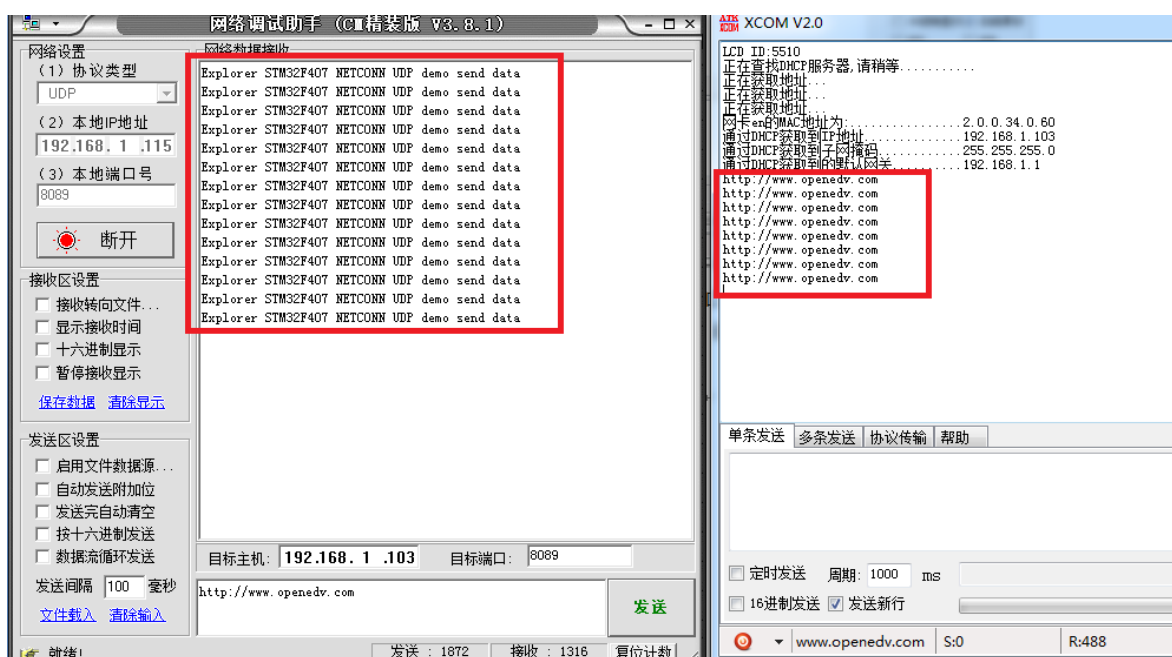


图 8.2.3 开发板和网络调试助手互相发送数据

第九章 NETCONN 编程接口 TCP 客户端实验

本章实验中开发板做 TCP 客户端，网络调试助手为 TCP 服务器。开发板连接到 TCP 服务器(网络调试助手)，网络调试助手给开发板发送数据，开发板接收数据并通过串口将接收到的数据发送到串口调试助手上，也可以通过按键从开发板向网络调试助手发送数据。本章分为如下几个部分：

9.1 软件设计

9.2 下载验证

9.1 软件设计

9.1.1 UCOSII 版本

关于 TCP 的基础知识我们在第四章讲 RAW 编程接口的 TCP 客户端实验时已经讲过了，这里就不重复讲解了。

本章对应着我们网络例程的网络实验 8 例程，打开我们的例程，在 LWIP->lwip_app 下有 tcp_client_demo.c 和 tcp_client_demo.h 两个文件，这两个文件就是我本章实验的源码，在 tcp_client_demo.c 中我们实现了两个函数 tcp_client_thread()和 tcp_client_init()，同上一章一样，在有操作系统的支持下，TCP 客户端可以作为一个线程来处理。

tcp_client_thread()函数为 TCP 客户端任务函数，tcp_client_init()创建一个 TCP 客户端线程，tcp_client_thread ()函数代码如下。

```
static void tcp_client_thread(void *arg)
{
    OS_CPU_SR cpu_sr;
    u32 data_len = 0;
    struct pbuf *q;
    err_t err,recv_err;
    static ip_addr_t server_ipaddr,loca_ipaddr;
    static u16_t      server_port,loca_port;

    LWIP_UNUSED_ARG(arg);
    server_port = REMOTE_PORT;
    IP4_ADDR(&server_ipaddr,lwipdev.remoteip[0],lwipdev.remoteip[1],\
    lwipdev.remoteip[2],lwipdev.remoteip[3]);

    while (1)
    {
        ① tcp_clientconn = netconn_new(NETCONN_TCP); //创建一个 TCP 链接
        ② err = netconn_connect(tcp_clientconn,&server_ipaddr,server_port);//连接服务器
        //返回值不等于 ERR_OK,删除 tcp_clientconn 连接
        if(err != ERR_OK) netconn_delete(tcp_clientconn);
        else if (err == ERR_OK) //处理新连接的数据
        {
            struct netbuf *recvbuf;
            tcp_clientconn->recv_timeout = 10;
            //获取本地 IP 主机 IP 地址和端口号
            netconn_getaddr(tcp_clientconn,&loca_ipaddr,&loca_port,1);
            printf("连接上服务器%d.%d.%d.%d,本机端口号为:%d\r\n",lwipdev.remoteip[0],\
            lwipdev.remoteip[1], lwipdev.remoteip[2],lwipdev.remoteip[3],loca_port);
            while(1)
            {
                //有数据要发送
                ③ if((tcp_client_flag & LWIP_SEND_DATA) == LWIP_SEND_DATA)
```

```

    {
        //发送 tcp_server_sendbuf 中的数据

        err=netconn_write(tcp_clientconn,\
            tcp_client_sendbuf,strlen((char*)tcp_client_sendbuf),\ NETCONN_COPY);
        if(err != ERR_OK)printf("发送失败\r\n");
        tcp_client_flag &= ~LWIP_SEND_DATA;
    }

    //接收到数据
④ if((recv_err = netconn_recv(tcp_clientconn,&recvbuf)) == ERR_OK)
    {
        OS_ENTER_CRITICAL(); //关中断
        //数据接收缓冲区清零
        memset(tcp_client_recvbuf,0,TCP_CLIENT_RX_BUFSIZE);
        for(q=recvbuf->p;q!=NULL;q=q->next) //遍历完整个 pbuf 链表
        {
            //判断要拷贝到 TCP_CLIENT_RX_BUFSIZE 中的数据是否大于
            //TCP_CLIENT_RX_BUFSIZE 的剩余空间,如果大于的话就只拷贝
            //TCP_CLIENT_RX_BUFSIZE 中剩余长度的数据,否则的话就拷贝
            //所有的数据
            if(q->len>(TCP_CLIENT_RX_BUFSIZE-data_len))memcpy(\
                tcp_client_recvbuf+data_len,q->payload,\
                    (TCP_CLIENT_RX_BUFSIZE-data_len)); //拷贝数据
            else memcpy(tcp_client_recvbuf+data_len,q->payload,q->len);
            data_len += q->len;
            //超出 TCP 客户端接收数组,跳出
            if(data_len > TCP_CLIENT_RX_BUFSIZE) break;
        }
        OS_EXIT_CRITICAL(); //开中断
        data_len=0; //复制完成后 data_len 要清零。
        printf("%s\r\n",tcp_client_recvbuf);
        netbuf_delete(recvbuf);
⑤ }else if(recv_err == ERR_CLSD) //关闭连接
    {
⑥     netconn_close(tcp_clientconn);
⑦     netconn_delete(tcp_clientconn);
        printf("服务器%d.%d.%d.%d 断开连接\r\n",lwipdev.remoteip[0],\
            lwipdev.remoteip[1], lwipdev.remoteip[2],lwipdev.remoteip[3]);
        break;
    }
}
}

```

```

}
}

```

①调用 `netconn_new()` 函数申请一个 TCP 连接结构，申请成功后的 `netconn` 结构为 `tcp_clientconn`。

②调用 `netconn_connect()` 函数将 `tcp_clientconn` 结构连接到目的 IP 地址和端口号上，在这里就是我们电脑的 IP 地址和相应的端口号。

③如果有数据要发送的话就发送数据，通过调用 `netconn_write()` 函数将待发送数据从 `tcp_clientconn` 连接结构上发送出去。

④调用 `netconn_recv()` 函数来接收数据，接收处理过程和上一章 UDP 实验中的处理方法类似，接收成功以后就将接收到的数据通过串口发送串口调试助手。

⑤如果数据接收错误，即在④中调用 `netconn_recv()` 函数后的返回值 `recv_err` 为 `ERR_CLSD` 时就关闭 TCP 连接，并且释放连接结构 `tcp_clientconn` 的内存。

⑥调用 `netconn_close()` 函数关闭 `tcp_clientconn` 连接。

⑦调用 `netconn_delete()` 函数释放 `tcp_clientconn` 结构的内存。

同样上一章实验一样，我们定义了一个全局变量 `tcp_client_flag` 来标记是否有数据发送，当有数据发送的时候 `tcp_client_flag` 的 bit8 就为 1，通过与 `LWIP_SEND_DATA` 进行与运算就知道是否有数据要发送，有数据要发送时就调用 `netconn_write()` 函数将 `tcp_client_sendbuf` 中的数据发送出去。

`tcp_client_init()` 为创建 TCP 客户端线程，这个函数比较简单，代码如下。

```

//创建 TCP 客户端线程
//返回值:0 TCP 客户端创建成功
//      其他 TCP 客户端创建失败
INT8U tcp_client_init(void)
{
    INT8U res;
    OS_CPU_SR cpu_sr;
    OS_ENTER_CRITICAL(); //关中断
    //创建 TCP 客户端线程
    res=OSTaskCreate(tcp_client_thread,(void*)0,(OS_STK*)&\
    TCPCLIENT_TASK_STK[TCPCLIENT_STK_SIZE-1],TCPCLIENT_PRIO);
    OS_EXIT_CRITICAL(); //开中断
    return res;
}

```

接下来就是 `main` 函数的编写，`main` 函数和上一章 UDP 实验的 `main` 函数几乎一模一样，关于 `main` 函数可以看一下上一章关于 `main` 函数的讲解，唯一不同的是 `key_task` 任务函数，按下 `KEY1` 键时是对全局变量 `tcp_client_flag` 的 bit1 置 1。

最后同样是通过 `lwip_comm_default_ip_set()` 函数设置远端 IP 地址。

9.1.2 UCOSIII 版本

UCOSIII 版本的代码和 UCOSII 版本的代码基本一致的，只是其中的某些 API 函数命名不同，原理上都是相通的。具体代码查看例程“[网络实验 11 NETCONN_TCP 客户端实验\(UCOSIII 版本\)](#)”，实验的原理详解请查看 9.1.1 小节。

9.2 下载验证

代码编译完成后就可以下载到 STM32F407 开发板中，开发板连接路由器，没有路由器的话就连接到电脑上，然后按照我们在第一章中讲解的方法设置电脑。我们需要打开网络调试助手，串口调试助手。复位开发板，等待开发板 LCD 显示如图 9.2.1 所示信息，在图 9.2.1 中显示了开发板的 IP 地址，子网掩码，默认网关，端口号等信息。

```
Explorer STM32F4
TCP CLIENT NETCONN Test
ATOM@ALIENTEK
KEY1:Send data
2014/9/1
Lwip Init Success!
TCP Client Success!
DHCP IP :192.168.1.103
DHCP GW :192.168.1.1
NET MASK:255.255.255.0
Port:8087!
```

图 9.2.1 LCD 显示。

我们来看一下串口调试助手，在串口调试助手上也输出了我们开发板的 IP 地址，子网掩码、默认网关等信息。将网络调试助手设置为 TCP 服务器，开发板会自动连接到 TCP 服务器(网络调试助手)，这是会在串口调试助手上输出信息提示连接到 TCP 服务器，如图 9.2.2 所示，从图中的绿色方框中可以看出此时开发板已经连接到 TCP 服务器。

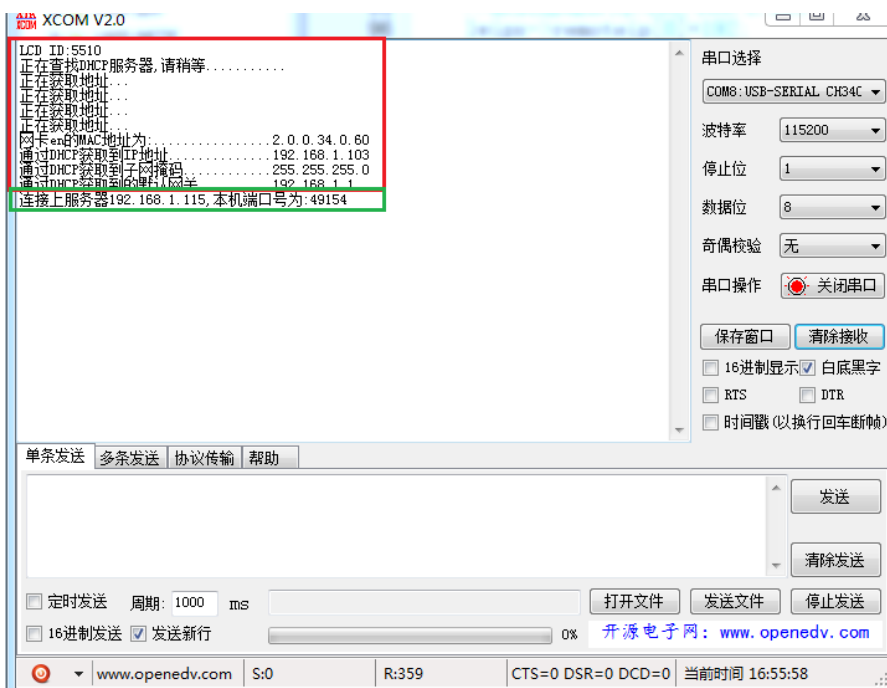


图 9.2.2 串口调试助手

我们通过网络调试助手和开发板互相发送数据，结果如图 9.2.3 所示。从图中我们可以看出开发板接收到网络调试助手发送过来的数据: <http://www.openedv.com>。开发板接收到数据后通过串口将接收到的数据发送给串口调试助手。网络调试助手接收到开发板发过来数据: Explorer STM32F407 NETCONN TCP Client send data。注意图 9.2.3 中网络调试助手的设置。

第十章 NETCONN 编程接口 TCP 服务器实验

本章实验中开发板做 TCP 服务器，网络调试助手做客户端，网络调试助手连接 TCP 服务器(开发板)。连接成功后网络调试助手可以给开发板发送数据，开发板接收数据并通过串口将接收到的数据发送到串口调试助手上，也可以通过按键从开发板向网络调试助手发送数据。本章分为如下几个部分：

10.1 软件设计

10.2 下载验证

10.1 软件设计

10.1.1 UCOSII 版本

本章对应着我们网络例程的网络实验 9 例程，打开我们的例程，在 LWIP->lwip_app 下有 tcp_server_demo.c 和 tcp_server_demo.h 两个文件，这两个文件就是我本章实验的源码，在 tcp_server_demo.c 中我们实现了两个函数 tcp_server_thread()和 tcp_server_init()，在有操作系统的支持下，TCP 服务器可以作为一个线程来处理。

tcp_server_thread()函数为 TCP 服务器任务函数，tcp_server_init()创建一个 TCP 服务器线程，tcp_server_thread()函数代码如下。

//tcp 服务器任务

```
static void tcp_server_thread(void *arg)
```

```
{
```

```
    OS_CPU_SR cpu_sr;
```

```
    u32 data_len = 0;
```

```
    struct pbuf *q;
```

```
    err_t err,recv_err;
```

```
    u8 remot_addr[4];
```

```
    struct netconn *conn, *newconn;
```

```
    static ip_addr_t ipaddr;
```

```
    static u16_t          port;
```

```
    LWIP_UNUSED_ARG(arg);
```

① conn = netconn_new(NETCONN_TCP); //创建一个 TCP 链接

② netconn_bind(conn,IP_ADDR_ANY,TCP_SERVER_PORT); //绑定端口 8 号端口

③ netconn_listen(conn); //进入监听模式

```
conn->recv_timeout = 10; //禁止阻塞线程等待 10ms
```

```
while (1)
```

```
{
```

④ err = netconn_accept(conn,&newconn); //接收连接请求

```
newconn->recv_timeout = 10;
```

```
if (err == ERR_OK) //处理新连接的数据
```

```
{
```

```
    struct netbuf *recvbuf;
```

```
netconn_getaddr(newconn,&ipaddr,&port,0); //获取远端 IP 地址和端口号
```

```
remot_addr[3] = (uint8_t)(ipaddr.addr >> 24);
```

```
remot_addr[2] = (uint8_t)(ipaddr.addr >> 16);
```

```
remot_addr[1] = (uint8_t)(ipaddr.addr >> 8);
```

```
remot_addr[0] = (uint8_t)(ipaddr.addr);
```

```
printf("主机%d.%d.%d.%d 连接上服务器,主机端口号为:%d\r\n",remot_addr[0],\
remot_addr[1],remot_addr[2],remot_addr[3],port);
```

```

while(1)
{
    //有数据要发送
    if((tcp_server_flag & LWIP_SEND_DATA) == LWIP_SEND_DATA)
    {
        //发送 tcp_server_sendbuf 中的数据
        err= netconn_write(newconn ,tcp_server_sendbuf,\
        strlen((char*)tcp_server_sendbuf),NETCONN_COPY);
        if(err != ERR_OK)printf("发送失败\r\n");
        tcp_server_flag &= ~LWIP_SEND_DATA;
    }

    //接收到数据
    if((recv_err = netconn_recv(newconn,&recvbuf)) == ERR_OK)
    {
        OS_ENTER_CRITICAL(); //关中断
        //数据接收缓冲区清零
        memset(tcp_server_recvbuf,0,TCP_SERVER_RX_BUFSIZE);
        for(q=recvbuf->p;q!=NULL;q=q->next) //遍历完整个 pbuf 链表
        {
            //判断要拷贝到 TCP_SERVER_RX_BUFSIZE 中的数据是否大于
            //TCP_SERVER_RX_BUFSIZE 的剩余空间,如果大于的话就只拷贝
            //TCP_SERVER_RX_BUFSIZE 中剩余长度的数据,否则的话就拷贝
            所有的数据
            if(q->len > (TCP_SERVER_RX_BUFSIZE-data_len)) memcpy(=\
            tcp_server_recvbuf+data_len,q->payload,\
            (TCP_SERVER_RX_BUFSIZE-data_len)); //拷贝数据
            else memcpy(tcp_server_recvbuf+data_len,q->payload,q->len);
            data_len += q->len;
            //超出 TCP 客户端接收数组,跳出
            if(data_len > TCP_SERVER_RX_BUFSIZE) break;
        }
        OS_EXIT_CRITICAL(); //开中断
        data_len=0; //复制完成后 data_len 要清零。
        printf("%s\r\n",tcp_server_recvbuf); //通过串口发送接收到的数据
        netbuf_delete(recvbuf);
    }else if(recv_err == ERR_CLSD) //关闭连接
    {
        netconn_close(newconn);
        netconn_delete(newconn);
        printf("主机:%d.%d.%d.%d 断开与服务器的连接\r\n",\
        remot_addr[0], remot_addr[1],remot_addr[2],remot_addr[3]);
    }
}

```

```

        break;
    }
}
}
}
}
}

```

①调用 `netconn_new()` 函数申请一个 TCP 连接结构，申请成功后的 `netconn` 结构为 `conn`。

②调用 `netconn_bind()` 函数将 `conn` 结构连接到目的 IP 地址和端口号上，在这里就是我们电脑的 IP 地址和相应的端口号。

③这里就是和我们上一章 TCP 客户端实验的不同支持，这里通过调用 `netconn_listen()` 函数将 `conn` 结构设置为侦听状态。

④调用 `netconn_accept()` 函数来接收和处理新连接请求，建立好的新连接为 `newconn`，而 `conn` 还是侦听状态。

`tcp_server_thread()` 函数的其他内容和上一章 TCP 客户端中的 `tcp_client_thread()` 函数类似，我们在本章中也定义了一个全局变量 `tcp_server_flag` 来标记是否有数据发送，当有数据发送的时候 `tcp_server_flag` 的 bit8 就为 1，通过与 `LWIP_SEND_DATA` 进行与运算就知道是否有数据要发送，有数据要发送时就调用 `netconn_write()` 函数将 `tcp_server_sendbuf` 中的数据发送出去。

`tcp_server_init()` 为创建 TCP 服务器线程，这个函数比较简单，代码如下。

```

//创建 TCP 服务器线程
//返回值:0 TCP 服务器创建成功
//      其他 TCP 服务器创建失败
INT8U tcp_server_init(void)
{
    INT8U res;
    OS_CPU_SR cpu_sr;
    OS_ENTER_CRITICAL(); //关中断
    //创建 TCP 服务器线程
    res = OSTaskCreate(tcp_server_thread,(void*)0,(OS_STK*)&\
        TCPSERVER_TASK_STK[TCPSERVER_STK_SIZE-1],TCPSERVER_PRIO);
    OS_EXIT_CRITICAL(); //开中断
    return res;
}

```

`main` 函数代码和前两章的 `main` 函数类似，这里就不做讲解了。

10.1.2 UCOSIII 版本

UCOSIII 版本的代码和 UCOSII 版本的代码基本一致的，只是其中的某些 API 函数命名不同，原理上都是相通的。具体代码查看例程“[网络实验 13 NETCONN_TCP 服务器实验\(UCOSIII 版本\)](#)”，实验的原理详解请查看 10.1.1 小节。

10.2 下载验证

代码编译完成后就可以下载到 STM32F407 开发板中，开发板连接路由器，没有路由器的话就连接到电脑上，然后按照我们在第一章中讲解的方法设置电脑。我们需要打开网络调试助手，串口调试助手。复位开发板，等待开发板 LCD 显示如图 10.2.1 所示信息，在图 10.2.1 中显

示了开发板的 IP 地址, 子网掩码, 默认网关, 端口号等信息。

```
Explorer STM32F4
TCP Server NETCONN Test
ATOM@ALIENTEK
KEY1:Send data
2014/9/1
Lwip Init Success!
TCP Server Success!
DHCP IP :192.168.1.103
DHCP GW :192.168.1.1
NET MASK:255.255.255.0
Port:8088!
```

图 10.2.1 LCD 显示。

我们在来看一下串口调试助手, 在串口调试助手上也输出了我们开发板的 IP 地址, 子网掩码、默认网关等信息。将网络调试助手设置为 TCP 客户端, 网络调试助手会自动连接到 TCP 服务器(开发板), 这是会在串口调试助手上输出信息提示有主机连接到 TCP 服务器上(开发板), 如图 10.2.2 所示, 从图中的绿色方框中可以看出此时网络调试助手连接上开发板。

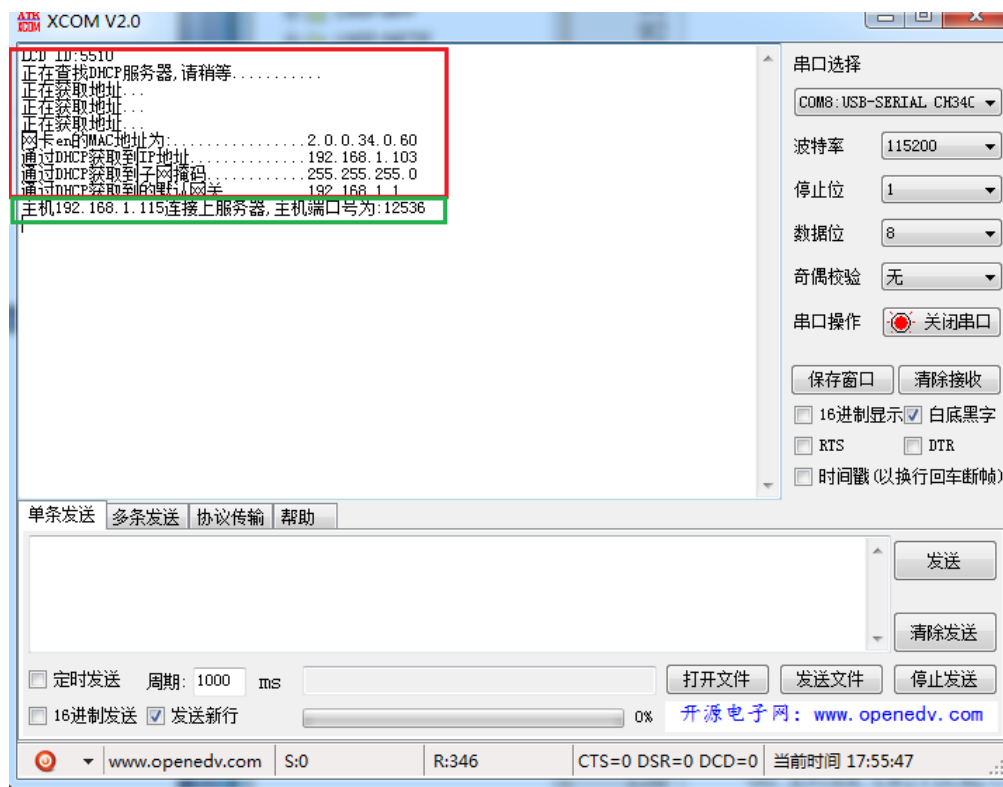


图 9.2.2 串口调试助手

我们通过网络调试助手和开发板互相发送数据，结果如图 10.2.3 所示。从图中我们可以看出开发板接收到网络调试助手发送过来的数据: `http://www.openedv.com`。开发板接收到数据后通过串口将接收到的数据发送给串口调试助手。网络调试助手接收到开发板发送来数据: `Explorer STM32F407 NETCONN TCP Server send data`。注意图 10.2.3 中网络调试助手的设置。

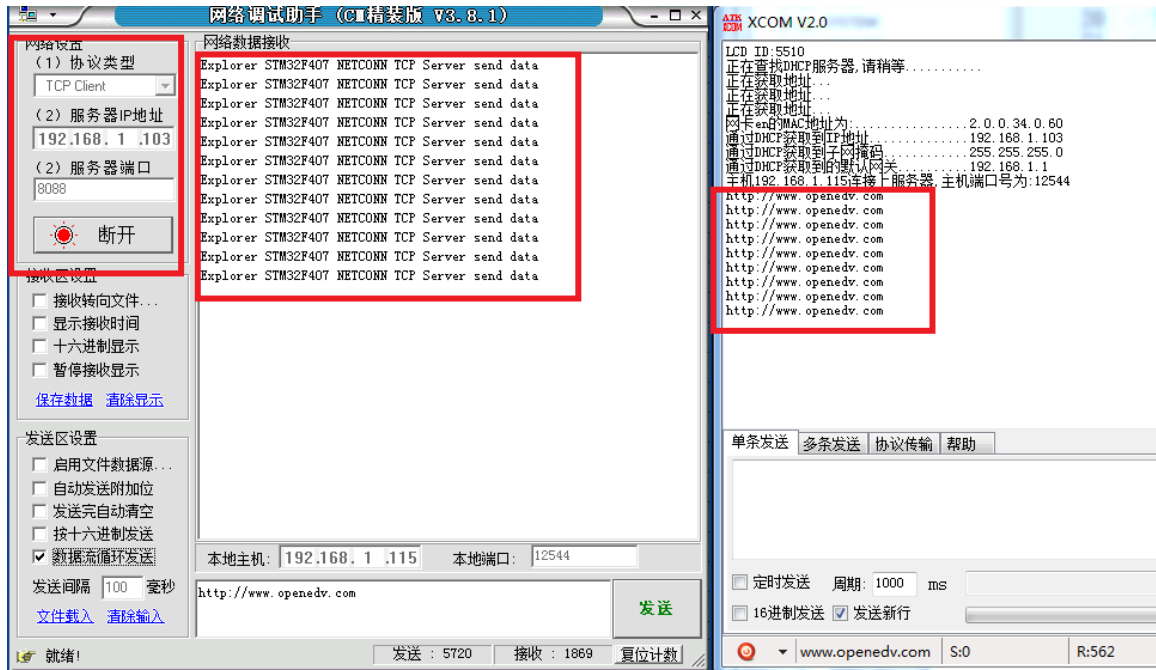


图 10.2.3 开发板和网络调试助手互相发送数据