

1	下载LwIP	2
2	建立一个最基本的工程	2
3	把LwIP加入工程	2
4	编写操作系统模拟层相关代码	3
4.1	操作系统模拟层移植说明——中文翻译	3
4.2	编写操作系统模拟层	6
4.2.1	准备工作——建立文件、定义数据类型及其它	6
4.2.2	信号量操作函数	8
4.2.3	邮箱操作函数	13
4.2.4	实现sys_thread_new() 函数	20
4.2.5	实现sys_arch_timeouts() 函数	22
4.2.6	实现临界保护函数	25
4.2.7	扫尾——结束操作系统模拟层的编写	26
5	LwIP接口——初始设置及网络驱动	28
5.1	准备工作——建立LwIP入口函数文件	28
5.2	__ilvInitLwIP()	29
5.3	__ilvSetLwIP()	30
5.4	ethernetif_init()——初始化底层接口	35
5.4.1	ethernetif_init() 函数分析	35
5.4.2	low_level_output()——链路层发送函数	36
5.4.3	low_level_init()——网卡初始化函数	38
5.4.4	EMACInit()——网卡初始化工作的实际完成者	40
5.4.5	ethernetif_input()——实现接收线程	47
5.4.6	low_level_input()——得到一整帧数据	49
5.4.7	GetInputPacketLen()——获得帧长	50
5.4.8	EMACReadPacket()——复制，从接收缓冲区到pbuf	53
5.4.9	EMACSendPacket()——发送一帧数据	55
5.4.10	编译——ethernetif.c及lib_emac.c	56
6	ping——结束LwIP的移植	57
6.1	编译、链接整个工程	57
6.2	ping测试	59
	后记	62

本文将指导读者一步步完成 LwIP 在 ADS1.2 开发环境下的移植工作，包括底层驱动的编写。本文使用的硬件平台是 AT91SAM7X256 + RTL8201BL (PHY)，至于软件平台，读者从本文标题即可看出。我们使用 uC/OS-II 作为底层操作系统，而 LwIP 的移植亦将主要围绕 uC/OS-II 展开。好了，不再多说，开始吧……

1 下载 LwIP

很简单，到 LwIP 的官方网站即可：<http://savannah.nongnu.org/projects/lwip/>。如果你不想看看其它内容（可能对你很重要），就只是想得到源码，好的，直接到这个地址下载：<http://download.savannah.nongnu.org/releases/lwip/>。目前官方发布的最新版本是 1.1.1，找到 lwip-1.1.1.zip，然后下载、解压缩，第一项工作完成。

2 建立一个最基本的工程

要想完成移植工作，我们必须要有个包含 uC/OS-II 的工程才行，这一步我们就是要建立这个工程。工程建立完毕后，编译链接没有问题，那么，第二项工作也完成了。关于如何建立一个包含 uC/OS-II 的 ADS 工程的问题，不在本文描述范围之内，这里不做讲述。随本笔记一同发布的源码文档中 LwIPPortingTest_2 文件夹下包含了这个最基本工程的源码，读者可以直接使用。我的基本工程建立的路径是 D:\work\LwIPPortingTest，下文将以相对路径进行讲述，不再提供绝对路径。

3 把 LwIP 加入工程

首先，在 \src\ 文件夹下，建立 LwIP 文件夹，即：\src\LwIP；然后将下载的 LwIP 源码文件中 api、core、include、netif 文件复制到 \src\LwIP\ 文件夹下，如下图所示：

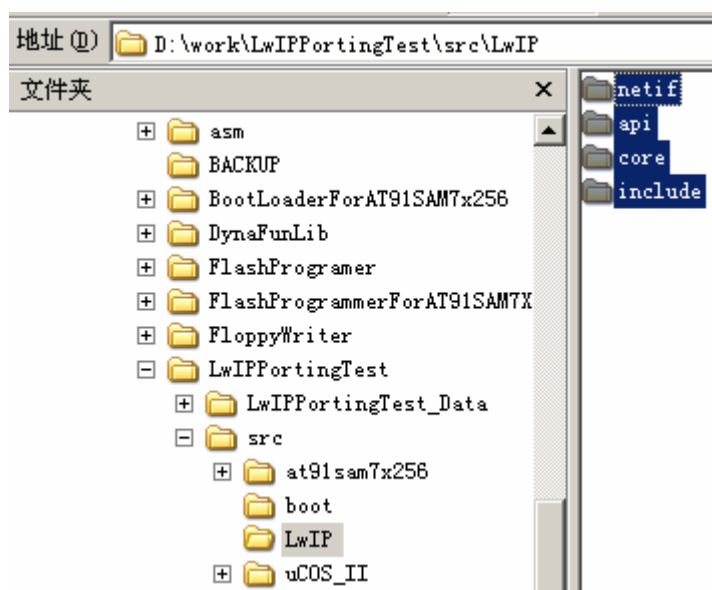


图 3.1

然后，用 ADS 打开工程文件，按照 LwIP 源码文件的实际存放路径建立 LwIP 的工程结构，如下图所示：

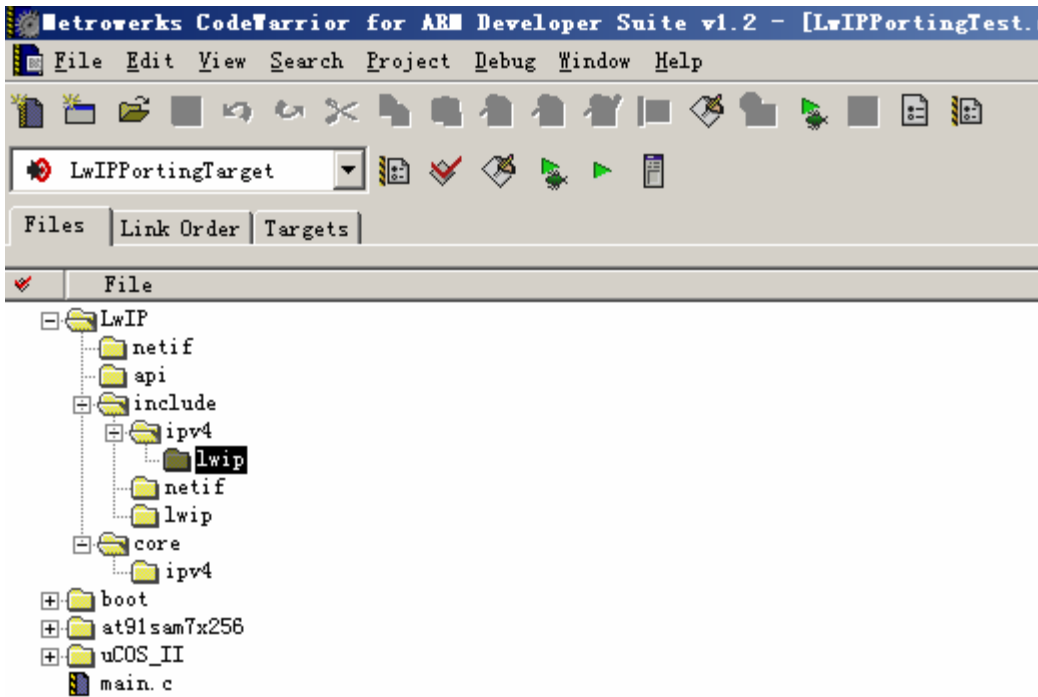


图 3.2

这里需要特别说明的是，源码中的 IP V6、SLIP 及 PPP 部分我们没有添加进来，主要是考虑我及大多数读者的网络还是 V4，而 SLIP、PPP 暂时不在我的考虑范围之内。另外，在移植层面 V6 也和 V4 相差不多，这里就不再讲解这部分内容了。现在基础工程结构建立完毕，可以把 LwIP 源码添加进来了。这一步很容易，按照文件存放路径，将源码文件添加到相应的工程结构下即可。源码添加完成后的工程参见所附源码文件的 LwIPPortingTest_3 文件夹。

4 编写操作系统模拟层相关代码

LwIP 的作者为操作系统模拟层提供了较为详细的说明，文件名为 sys_arch.txt，在 LwIP 的 doc 文件夹下。我们的编写工作根据这个说明进行。

4.1 操作系统模拟层移植说明——中文翻译

事先声明，之所以笔者要翻译该文档，主要是笔者在撰写这篇笔记时亦没有通读该文档。笔者先前使用的模拟层源码是杨晔大侠的。为了真正弄懂 LwIP，笔者决定自己重新实现 LwIP 的移植，本笔记是跟随移植同步进行的，因此，翻译的文档也放在了这篇笔记中，使读者能够真正了解笔者的移植历程。另外再说一句，这个文档是为 LwIP 0.6++ 版编写，笔者搜遍了整个 LwIP 官方网站，没有发现比这更新的，笔者只好认为操作系统模拟层在 0.6++ 之后没有任何改动，如果有谁发现了更新的，一定通知笔者，先谢谢了。好的，言归正传，下面就是译文：

LwIP 0.6++ sys_arch 接口

作者：Adam Dunkels

操作系统模拟层（sys_arch）存在的目的主要是为了方便 LwIP 的移植，它在底层操作系统和 LwIP 之间提供了一个接口。这样，我们在移植 LwIP 到一个新的目标系统时，只需修改这个接口即可。不过，不依赖底层操作系统的支持也可以实现这个接口。

sys_arch 需要为 LwIP 提供信号量(semaphores)和邮箱(mailboxes)两种**进程间通讯方式**(IPC)。如果想获得 LwIP 的完整功能，**sys_arch 还必须支持多线程**。当然，对于仅需要基本功能的用户来说，可以不去实现多线程。LwIP 以前的版本还要求 sys_arch 实现定时器调度，不过，从 LwIP 0.5 开始，这一需求在更高层实现。除了上文所述的 sys_arch 源文件需要实现的功能外，LwIP 还要求用户提供几个头文件，这几个头文件包含 LwIP 使用的宏定义。下文将详细讲述 sys_arch 及头文件的实现。

信号量即可以是计数信号量，也可以是二值信号量——LwIP 都可以正常工作。邮箱用于消息传递，用户**即可以将其实现为一个队列，允许多条消息投递到这个邮箱，也可以每次只允许投递一个消息**。这两种方式 LwIP 都可以正常运作。不过，**前者更加有效**。需要用户特别注意的是一**投递到邮箱中的消息只能是一个指针**。

在 sys_arch.h 文件中，我们指定数据类型“sys_sem_t”表示信号量，“sys_mbox_t”表示邮箱。至于 sys_sem_t 和 sys_mbox_t 如何表示这两种不同类型，LwIP 没有任何限制。

以下函数必须在 sys_arch 中实现：

- void sys_init(void)

初始化 sys_arch 层。

- sys_sem_t sys_sem_new(u8_t count)

建立并返回一个新的信号量。参数 count 指定信号量的初始状态。

- void sys_sem_free(sys_sem_t sem)

释放信号量。

- void sys_sem_signal(sys_sem_t sem)

发送一个信号。

- u32_t sys_arch_sem_wait(sys_sem_t sem, u32_t timeout)

等待指定的信号并阻塞线程。timeout 参数为 0，线程会一直被阻塞至收到指定的信号；非 0，则线程仅被阻塞至指定的 timeout 时间（单位为毫秒）。在 timeout 参数值非 0 的情况下，返回值为等待指定的信号所消耗的毫秒数。如果在指定的时间内并没有收到信号，返回值为 SYS_ARCH_TIMEOUT。如果线程不必再等待这个信号（也就是说，已经收到信号），返回值也可以为 0。注意，**LwIP 实现了一个名称与之相似的函数来调用这个函数，sys_sem_wait()**，注意区别。

- sys_mbox_t sys_mbox_new(void)

建立一个空的邮箱。

- void sys_mbox_free(sys_mbox_t mbox)

释放一个邮箱。**如果释放时邮箱中还有消息，它表明 LwIP 中存在一个编程错误**，应该通知开发者（原文如此，这句话很费解。个人理解的意思是：当执行 sys_mbox_free() 这个函数时，按道理邮箱中不应该再存在任何消息，如果用户使用 LwIP 时发现邮箱中还存在消息，说明 LwIP 的开发者存在一个编程错误，不能把邮箱中的消息全部取出并处理掉。遇到这种情况，用户应该告诉 LwIP 的作者，纠正这个 bug，译注）。

- void sys_mbox_post(sys_mbox_t mbox, void *msg)

投递消息“msg”到指定的邮箱“mbox”。

- u32_t sys_arch_mbox_fetch(sys_mbox_t mbox, void **msg, u32_t timeout)

阻塞线程直至邮箱收到至少一条消息。最长阻塞时间由 timeout 参数指定（与 sys_arch_sem_wait() 函数类似）。msg 是一个结果参数，用来保存邮箱中的消息指针（即 *msg = ptr），它的值由这个函数设置。“msg”参数有可能为空，这表明当前这条消息应该被丢弃。返回值与 sys_arch_sem_wait() 函数相同：等待的毫秒数或者 SYS_ARCH_TIMEOUT——如果时间溢出的话。LwIP 实现的函数中，有一个名称与之相似的——sys_mbox_fetch()，注意区分。

- struct sys_timeouts *sys_arch_timeouts(void)

返回一个指向当前线程使用的 sys_timeouts 结构的指针。LwIP 中，每一个线程都有一个 timeouts 链表，这个链表由 sys_timeout 结构组成，sys_timeouts 结构则保存了指向这个链表的指针。这个函数由 LwIP 的超时调度程序调用，并且不能返回一个空（NULL）值。单线程 sys_arch 实现中，这个函数只需简单返回一个指针即可。这个指针指向保存在 sys_arch 模块中的 sys_timeouts 全局变量。

如果底层操作系统支持多线程并且 LwIP 中需要这样的功能，那么，下面的函数必须实现：

- sys_thread_t sys_thread_new(void(*thread)(void *arg), void *arg, int prio)

启动一个由函数指针 thread 指定的新线程，arg 将作为参数传递给 thread() 函数，prio 指定这个新线程的优先级。返回值为这个新线程的 ID，ID 和优先级由底层操作系统决定。

- sys_prot_t sys_arch_protect(void)

这是一个可选函数，它负责完成临界区域保护并返回先前的保护状态。该函数只有在小的临界区域需要保护时才会被调用。基于 ISR 驱动的嵌入式系统可以通过禁止中断来实现这个函数。基于任务的系统可以通过互斥量或禁止任务来实现这个函数。该函数应该支持来自于同一个任务或中断的递归调用。换句话说，当该区域已经被保护，sys_arch_protect() 函数依然能被调用。这时，函数的返回值会通知调用者该区域已经被保护。

如果你的移植正在支持一个操作系统，sys_arch_protect() 函数仅仅是一个需要。

- void sys_arch_unprotect(sys_prot_t pval)

该函数同样是一个可选函数。它的功能就是恢复受保护区域的先前保护状态，先前是受到保护还是没有受到保护由参数 pval 指定。它与 sys_arch_protect() 函数配套使用，详细信息参看 sys_arch_protect() 函数。

该函数的说明是按照译者个人理解的意思翻译，原文讲述不是很清楚，如有错误，欢迎批评指正，译注。

OS 支持的模拟层需要添加的头文件说明

- cc.h 与硬件平台及编译器相关的环境变量及数据类型声明文件（一些或许应该移到 sys_arch.h 文件）。

LwIP 使用的数据类型定义——u8_t, s8_t, u16_t, s16_t, u32_t, s32_t, mem_ptr_t。

与编译器相关的 LwIP 结构体封装宏：

```
PACK_STRUCT_FIELD(x)
PACK_STRUCT_STRUCT
PACK_STRUCT_BEGIN
PACK_STRUCT_END
```

与平台相关的调试输出：

```
LWIP_PLATFORM_DIAG(x)    - 非故障，输出一条提示信息。
LWIP_PLATFORM_ASSERT(x)  - 故障，输出一条故障信息并放弃执行。
```

“轻便的 (lightweight)” 同步机制：

```
SYS_ARCH_DECL_PROTECT(x) - 声明一个保护状态变量。
SYS_ARCH_PROTECT(x)      - 进入保护模式。
SYS_ARCH_UNPROTECT(x)    - 脱离保护模式。
```

如果编译器不提供 `memset()` 函数，这个文件必须包含它的定义，或者包含 (`include`) 一个定义它的文件。

这个文件要么包含一个本地系统 (`system-local`) 提供的头文件 `<errno.h>`——这个文件定义了标准的 `*nix` 错误编码，要么增加一条宏定义语句：`#define LWIP_PROVIDE_ERRNO`，这将使得 `lwip/arch.h` 头文件来定义这些编码。这些编码被用于 LwIP 的各个部分。

- `perf.h` 定义了性能测量使用的宏，由 LwIP 调用，可以将其定义为一个空的宏。

```
PERF_START      - 开始测量。
PERF_STOP(x)    - 结束测量并记录结果。
```

- `sys_arch.h` `sys_arch.c` 的头文件。

定义 Arch (即整个移植所依赖的操作系统平台，译注) 需要的数据类型：`sys_sem_t`，`sys_mbox_t`，`sys_thread_t`，以及可选类型：`sys_prot_t`。

`sys_mbox_t` 和 `sys_sem_t` 变量的 NULL 值定义：

```
SYS_MBOX_NULL    NULL
SYS_SEM_NULL     NULL
```

4.2 编写操作系统模拟层

4.1 节已经明白的讲述了如何实现 `sys_arch` 接口，我们按照这个说明完成即可。

4.2.1 准备工作——建立文件、定义数据类型及其它

在 ADS 工程 LwIP 组中添加一个新组 `arch` 并在这个组下面建立源文件 `sys_arch.c`，实际存放路径亦如此组织，如下图所示：

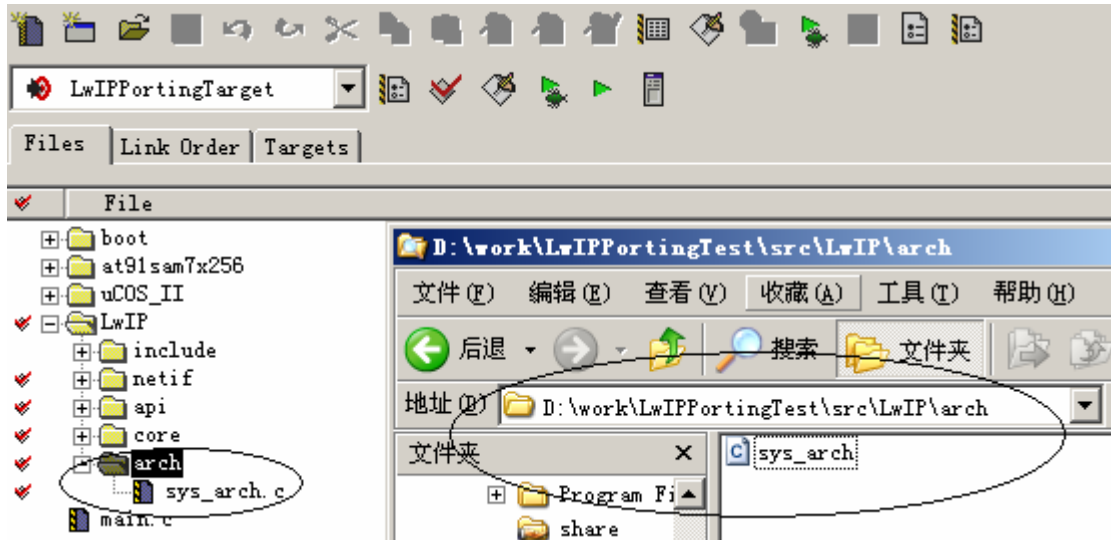


图 4.2.1

然后在 LwIP/include 组同样建立一个新组 arch, 在 arch 组建立新文件 sys_arch.h 及 cc.h, 如下图示:

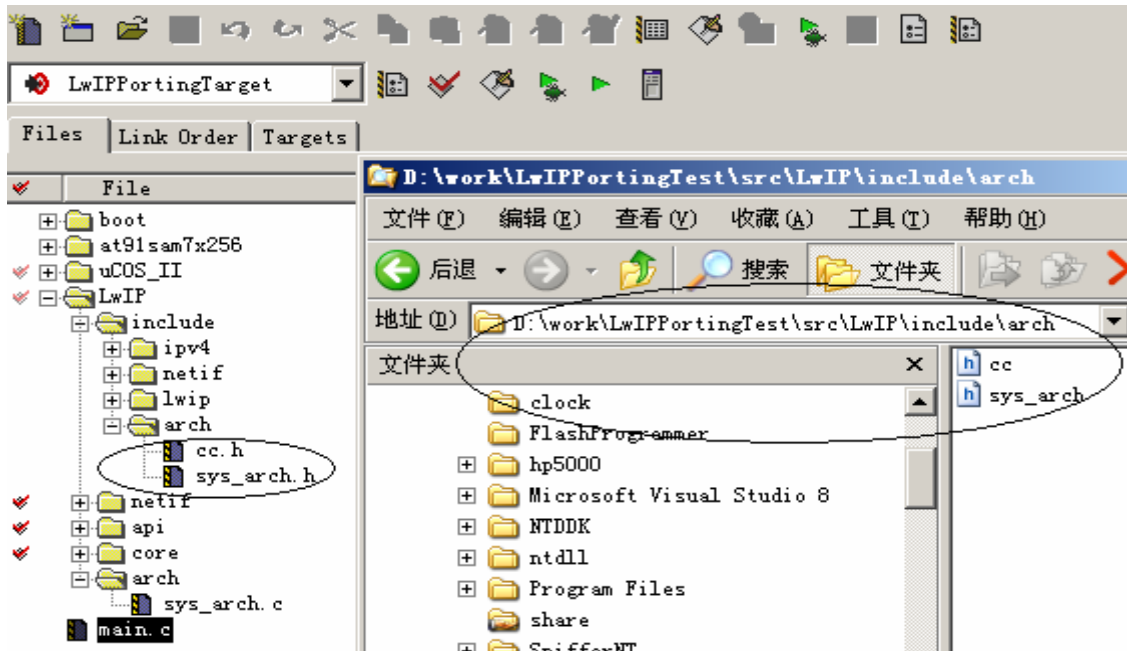


图 4.2.2

文件建立完成, 我们先实现数据类型定义, 这个实现完全按照移植说明一文进行。首先在 cc.h 文件中定义常用的数据类型。这些常用数据类型不仅模拟层接口函数使用, 底层协议栈实现亦要使用。在 cc.h 文件中添加如下语句 (参见 LwIPPortingTest_4 文件夹下的 cc.h 文件):

```
typedef unsigned char  u8_t;
typedef char           s8_t;
typedef unsigned short u16_t;
typedef short          s16_t;
typedef unsigned int   u32_t;
typedef int            s32_t;
typedef u32_t          mem_ptr_t;
```

在上面的数据类型定义中, 除了 mem_ptr_t 之外其它类型均很直观, 不需解释。至于 mem_ptr_t

为什么指定为 `u32_t`，而不是像它的名称所表现的一样将其指定为指针呢？其实原因很简单，笔者在定义它时首先找到了使用它的相关语句，从这些语句中才确定这样声明。读者可找到 `mem.h` 文件看看里面有关 `mem_ptr_t` 的使用语句就能明白怎么回事。好了，不再多说，让我们的准备工作接着进行。在 `sys_arch.h` 文件中添加如下语句：

```
typedef HANDLER sys_sem_t;
```

其中 `HANDLER` 是笔者本人自定义的一个宏，它是为了方便 `uC/OS-II` 的使用定义的。读者可以在所附源码文件 `\src\uCOS-II\API\os_api.h` 中找到相关定义：

```
typedef OS_EVENT* HANDLER;
```

它实际上就是指向 `uCOS` 中 `OS_EVENT` 结构的指针。

声明相关数据类型的语句添加完成后，我们再把这两个文件 `sys_arch.h` 和 `cc.h` 添加到 `sys_arch.c` 文件中，以使该文件里的相关函数能够使用这些新定义的数据类型：

```
#include "/LwIP/include/arch/cc.h"
```

```
#include "/LwIP/include/arch/sys_arch.h"
```

这里一定要注意顺序，先包含 `cc.h` 文件，再包含 `sys_arch.h` 文件，因为 `sys_arch.h` 文件中有些语句需要用到 `cc.h` 文件中的类型声明。好了，准备工作已经完成，现在开始编写接口函数。

4.2.2 信号量操作函数

相关函数实现读者也可直接参看 `sys_arch.c` 文件。

```
- sys_new_sem()
```

```
/*-----
/* 函数名称 : sys_sem_new
/* 功能描述 : 建立并返回一个新的信号量
/* 入口参数 : <count>[in] 指定信号量的初始状态
/* 出口参数 : 返回新的信号量
/*-----
sys_sem_t sys_sem_new(u8_t count)
{
    return OSAPISemNew(count);
}
```

这个函数的实现其实很简单，因为 `uC/OS-II` 提供了信号量，我们只需直接调用建立信号量的相关函数就行了。上面的源码中 `OSAPISemNew` 是笔者本人为了统一对 `OS` 底层函数的调用重新定义的一个接口函数，这个接口函数负责调用 `OS` 底层函数完成相应功能。在后面的模拟层接口函数实现中，笔者使用了很多这样的 API。这些接口函数都以 `OSAPI` 作为函数名前缀，其实现细节请参看 `\src\uCOS-II\API\os_api.c` 文件，本文不再赘述。

```
- sys_sem_signal()
```

```
/*-----
/* 函数名称 : sys_sem_signal
/* 功能描述 : 发送信号
/* 入口参数 : <sem>[in] sem 指定要发送的信号
/* 出口参数 : 无
/*-----
```



```
void sys_sem_signal(sys_sem_t sem)
{
    OSAPISemSend(sem);
}
```

这个函数就不再多说了，与 sys_sem_new() 函数的实现机制相同。

```
- sys_sem_free()
/*-----
/* 函数名称 : sys_sem_free
/* 功能描述 : 释放信号量
/* 入口参数 : <sem>[in] 指定要释放的信号量
/* 出口参数 : 无
/*-----
void sys_sem_free(sys_sem_t sem)
{
    OSAPISemFreeExt(sem);
}
```

与前两个函数相似，不再赘述。

```
- sys_arch_sem_wait()
/*-----
/* 函数名称 : sys_arch_sem_wait
/* 功能描述 : 等待由参数 sem 指定的信号并阻塞线程
/* 入口参数 : <sem>[in] sem 指定要发送的信号
/*           : <timeout>[in] 指定等待的最长时间（单位为毫秒）。为 0，线程会一直
/*           : 被阻塞直至收到指定的信号；非 0，指定线程最长等待时
/*           : 间
/* 出口参数 : - 0: 在指定时间内等到指定信号
/*           : - SYS_ARCH_TIMEOUT: 在指定时间内没有等到指定信号
/*-----
u32_t sys_arch_sem_wait(sys_sem_t sem, u32_t timeout)
{
    if(OSAPISemWait(sem, timeout) == OS_NO_ERR)
        return 0;
    else
        return SYS_ARCH_TIMEOUT;
}
```

实现信号量等待的 OS 底层函数是 OSSemPend()。打开源码文件\src\uCOS_II\Os_sem.c，找到这个函数，我们会发现它与我们要实现的函数还是有些差别的：其一、OSSemPend() 函数没有返回值，它使用了一个结果参数 err 来代替返回值；其二、OSSemPend() 函数的执行结果与 sys_arch_sem_wait() 函数的执行结果（返回值）不相同，OSSemPend() 函数有 5 种执行结果，而 sys_arch_sem_wait() 函数只有两种结果；其三、timeout 参数的时间单位不同，一个是时钟节拍数（OSSemPend()），另一个则是毫秒数（sys_arch_sem_wait()）。这些差别的存在导致我们在调用 OSSemPend() 函数时必须根据需求作出相应的调整。

读者可以看到，在上面的代码实现中，笔者使用了自定义的 OS 接口函数来代替对 OSSemPend() 函数的直接调用。这个接口函数与 sys_arch_sem_wait() 函数相比不存在一、三这

两种差别。唯一的差别就是返回值，OSAPISemWait() 函数仍然保留了 OSSemPend() 函数的全部 5 种执行结果，而 LwIP 只关心两种结果：等到指定信号或者超时。笔者为了省事，sys_arch_sem_wait() 函数只判断是否等到指定信号（OSAPISemWait(sem, timeout) == OS_NO_ERR），其它情况一律视为超时（实际上除了超时外，另外三种属于程序的 BUG，一个运行稳定的系统不应该存在），返回 SYS_ARCH_TIMEOUT。

另外，按照笔者个人的理解（移植说明一文讲述不是很清楚），在指定时间内等到指定信号时，函数即可以返回实际等待的毫秒数也可以返回 0，为了编程方便，笔者选择返回 0。

既然移植说明一文告诉我们等待超时要返回 SYS_ARCH_TIMEOUT，那么在 LwIP 中一定会存在这个宏定义。使用 SYS_ARCH_TIMEOUT 作为关键字搜索整个 ADS 工程（CTRL+SHIFT+M）发现 sys.h 定义了这个宏，因此，sys_arch.c 文件还需要包含这个文件：

```
#include "LwIP/include/lwip/sys.h"
```

把上面的语句添加到 sys_arch.c 中，如下图示：

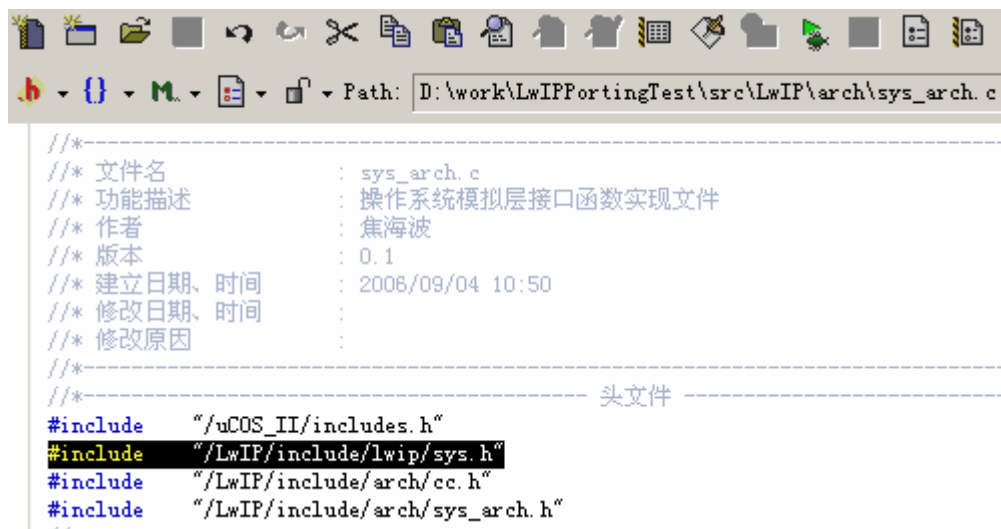


图 4.2.3

好了，编写到这里，让我们先把这个文件编译一下，看看有什么问题。读者可以使用 ADS 直接打开所附文档\LwIPPortingTest_4\LwIPPortingTest_4_1（它是直接反映本文当前移植进度的阶段性文档），找到 sys_arch.c 文件进行编译。编译完成，呵，错误真多，28 个，见下图：

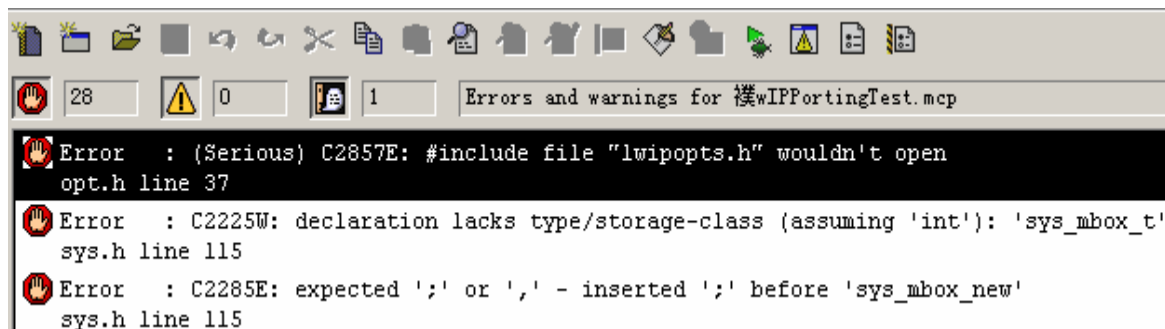


图 4.2.4

不怕，老话说的好：兵来将挡，水来土掩。跟着错误说明，让我们一个个地消灭。第一个错误是说无法打开 lwipopts.h 文件，出错的地方在 opt.h 文件。我们看看 opt.h 文件中关于 lwipopts.h 文件的描述，如下图示：

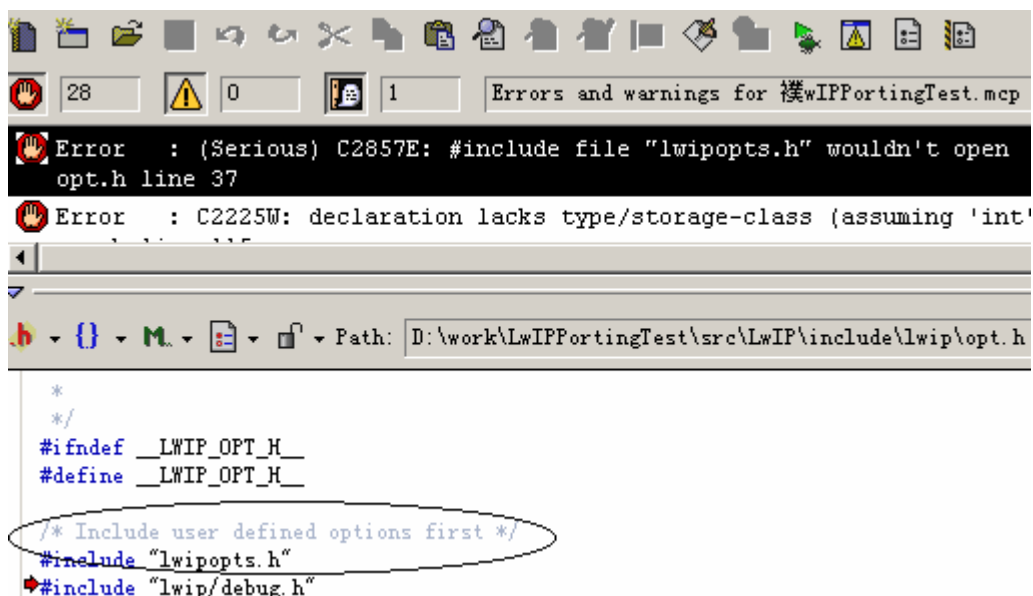


图 4.2.5

上图圈起来的注释语句就是对 lwipopts.h 文件的说明。很简单，它是用户自定义的配置文件，我们不需要这个，因此没有建立这个文件，编译器找不到它，只好报错了。解决方法很简单，我们把这条语句注释掉即可，如下图示：

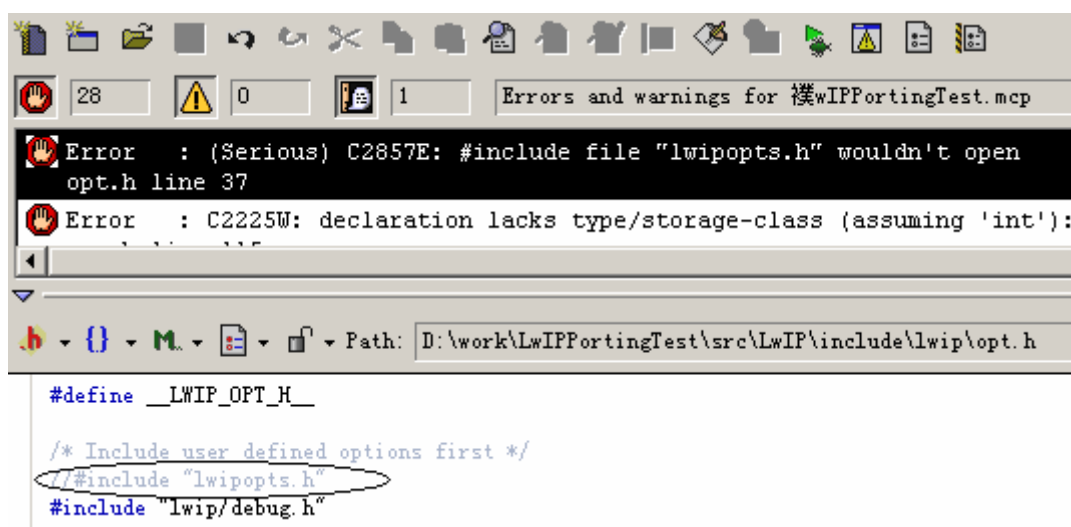


图 4.2.6

再编译看看，还有多少个错误，还有 27 个，少了一个，呵呵，进步啊。好，让我们继续进步……下面这个错误是说“sys_mbox_t”缺少相关类型声明。移植说明一文已经提到过 sys_mbox_t，其用来表示一个邮箱，暂时还用不到，所以在这里先将其随便声明为一种数据类型，先编译通过，以后再说。根据移植说明，我们在 sys_arch.h 头文件里定义它，打开 sys_arch.h，添加如下语句：

```
typedef HANDLER sys_mbox_t;
```

见图示：

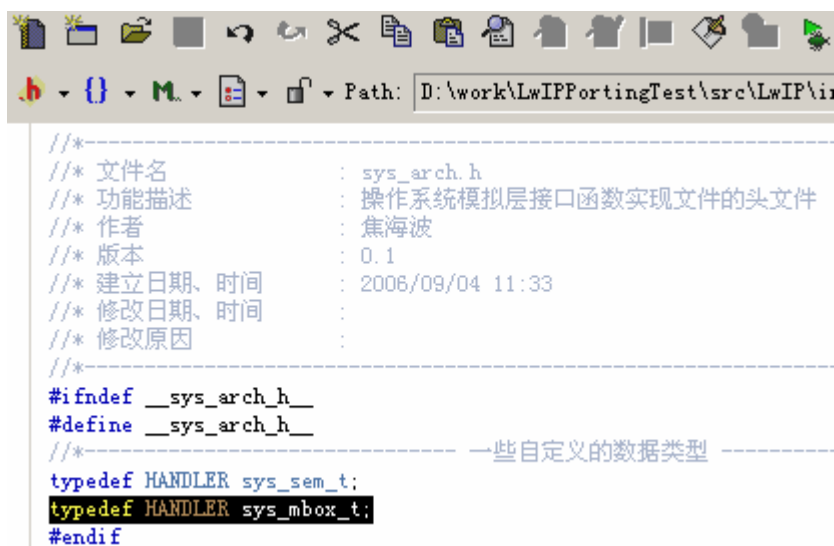


图 4.2.7

编译，出乎意料，还剩三个错误。太好了，已经看到胜利的彼岸了。剩下的这几个错误，核心问题与上一个问题一样，缺少声明，这次是“sys_thread_t”。它是新线程的 ID，在 uCOS 中，ID 号实际就是优先级，也就是 0-63 之间的数字，所以在 sys_arch.h 文件中添加如下语句：

```
typedef u8_t sys_thread_t;
```

如下图所示：

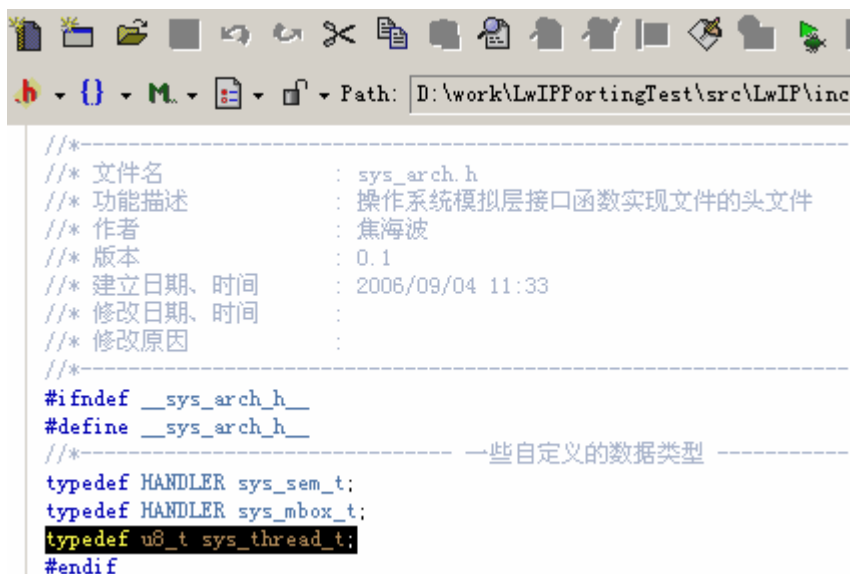


图 4.2.8

再编译，没有任何错误，通过。好了，信号量相关的操作函数已经完成，下面的工作就是实现邮箱操作函数。

4.2.3 邮箱操作函数

在动手实现之前，让我们再回顾一下移植说明一文关于邮箱的描述：“邮箱用于消息传递，用户即可以将其实现为一个队列，允许多条消息投递到这个邮箱，也可以每次只允许投递一个消息，这两种方式 LwIP 都可以正常运作。不过，前者更加有效。需要用户**特别注意的是一一投递到邮箱中的消息只能是一个指针**。”从这个描述中，我们得到两条有价值的信息：其一、邮箱一次能够接收多条消息比仅接收一条消息更加有效；其二、邮箱中的消息是一个指针。

先说说第一条。很显然，**使用消息队列带来的性能提升是影响我们选择的关键因素**，我们并不希望使用一个打了折扣的 LwIP。其实，在 uCOS 中实现消息队列很简单，它提供了非常丰富的消息队列管理函数，我们只需在此基础上实现即可。这也是影响我们如此选择的一个重要因素。对于第二条，其带给我们的信息很直白，不需赘述，唯一需要交待的是一一uCOS 中投递到邮箱中的消息也是一个指针，这对我们来说实在是一个大好消息。

好了，言归正传，下面谈谈笔者本人的设计思路，为读者抛砖引玉。

我的想法是系统可以同时建立多个邮箱，这些邮箱通过一个单向链表链接在一起。每个邮箱一次可以接收多条消息，接收消息的最大数量由消息数组的大小决定。如下图所示：

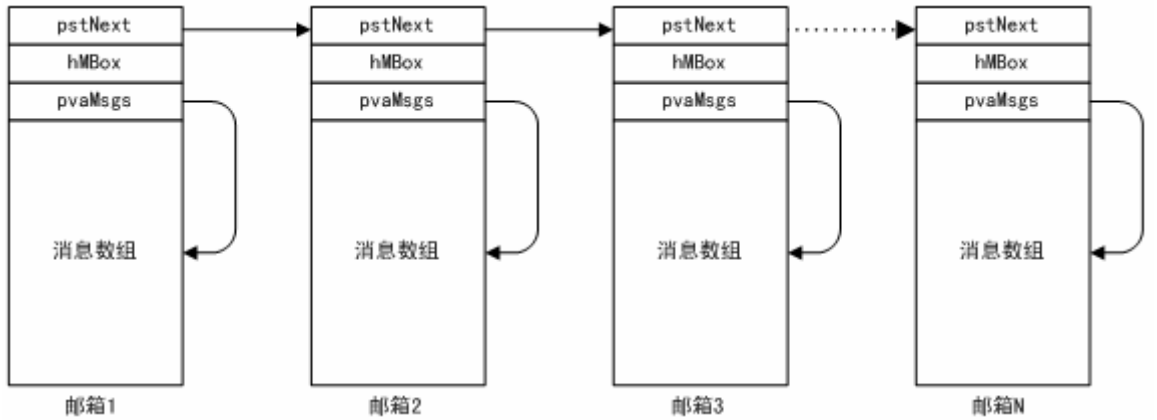


图 4.2.9

我们在系统初始化阶段完成建立邮箱和链表的工作，并把链表的首地址保存到全局变量 pstCurFreeMbox 指针中。

申请建立一个新邮箱时，我们先看看 pstCurFreeMbox 是否为空值。为空，表明链表已经没有任何空闲节点，无法满足申请。不为空，则立即将 pstCurFreeMbox 指向的当前节点从链表中取出交给申请者使用，pstCurFreeMbox 指向下一个空闲节点（见图 4.2.10）。在系统第一次申请的时候，pstCurFreeMbox 指向节点 1，申请完毕，节点 1 与链表断开链接，整个链表减少了一个节点。这时，pstCurFreeMbox 指针被赋值为节点 1 的 pstNext 值，其向后移动了一个节点，指向了节点 2。以此类推，假设系统只能申请建立邮箱，不能撤销申请将邮箱归还给链表，这将使得链表逐渐变短。当 pstCurFreeMbox 指针被赋值为节点 N 的 pstNext 值时，这意味着 pstCurFreeMbox 指针已经越过了链表的最后一个节点，链表的长度缩减为 0，链表不再可用。这时，如果系统再有新的申请产生，该申请将被抛弃。如果我们建立的链表长度过短，当网络系统繁忙导致占用的邮箱不能及时归还时，很有可能出现链表不可用的情况。这将直接导致网络系统的响应时间变长甚至不再响应。这对于一个设计可靠的系统来说，是一个很严重的问题。要想避免这个问题，就必须对网络系统的负荷能力作出正确的评估，从而为我们选择一个合理的链表长度提供判断依据。笔者对这个问题的解决方法是：在申请邮箱的函数中增加记录链表不可用次数的代码，然后模拟各种情况对目标系统进行尽可能多的并发访问，不断调整链表长度，直至不可用次数正好为 0。最后，把当前链表长度再增加一定数量以应付意外情况的发生

（一般再增加 25%-50% 的长度即可），就得到了我们想要的值。

邮箱使用完毕，归还给链表的过程参见图 4.2.10 归还节点部分。归还节点实际上就是重建链表，链表会从无到有，从短到长，逐渐恢复原来的样子。而 `pstCurFreeMBox` 指针的移动方向正好与申请时相反，它是从链表的尾端向前移动。不过，这里需要特别注意的是，链表的重建将会打乱链表最初的节点顺序，正如图 4.2.10 所示的一样。为什么会出现这种情况呢？原因很简单，虽然申请节点时我们是从链表的首部开始顺序申请，但是申请者占用各个节点的时间是不相同的，有的长有的短，与申请顺序无关。最先申请的节点有可能最后被归还，而最后申请的节点则有可能最先被归还。因为 `pstCurFreeMBox` 指针始终指向最后被归还的节点，也就是链表的首地址，因此，最后申请的节点成为了链表首部的节点，而原先首部的节点则成了重建后链表的尾部节点。图 4.2.10 很直观的描述了这一过程。善于思考的读者可能还会发现这种处理机制的一个特点，假设系统只申请了一个邮箱，然后归还，然后再申请，那么再申请的邮箱还是最先申请的那个邮箱。也就是说，如果系统最多只需要三个邮箱，而我们的链表共有五个邮箱，那么链表中的后两个邮箱将一直不被使用，而前三个邮箱则使用频繁。链表中的前三个节点顺序将不断的重新排列（123、321、231……），而节点 4 和节点 5 则会一直保持这个顺序。不过不用担心，这种情况除了造成一定的内存浪费之外（当然，如果我们的链表长度合适，这个问题也可避免），不会对系统的稳定产生任何问题。另外，链表建立在 RAM 而不是 FLASH 中，我们更不用考虑频繁擦写固定区域对存储器造成的伤害。

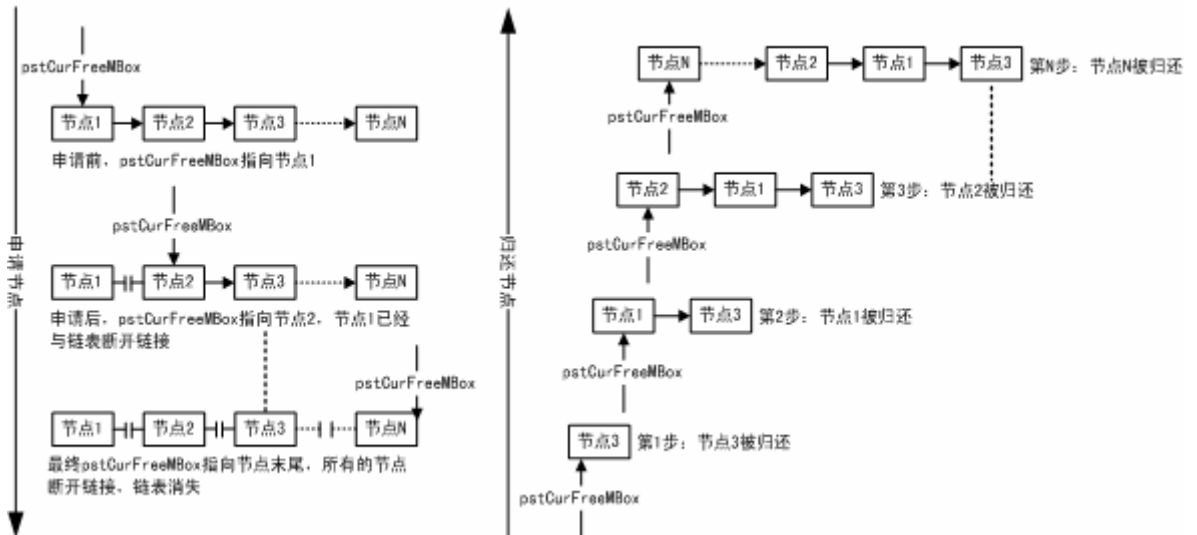


图 4.2.10

不知读者是否还记得在讲解信号量操作函数时，我们曾经提到过用于表示邮箱的自定义数据类型 `sys_mbox_t`。当时我们只是简单的为其作了类型定义，并没有考虑是否能够满足实际应用。春去春回、斗转星移，而现在我们对于如何实现邮箱已经了然于胸。前文已经说过，链表中的每一个节点就是一个邮箱，用 C 语言来描述——邮箱实际上就是一个结构体。`sys_mbox_t` 是用来表示邮箱的，而邮箱就是结构体，显然 `sys_mbox_t` 应该声明为结构体。为了操作简便，我们亦可以将其声明为指向这个结构体的指针。现在我们要做的就是实现这个结构体，然后再将 `sys_mbox_t` 声明为指向这个结构体的指针，完成邮箱操作的基本单元构建工作。实现这个结构体的工作很简单，聪明的读者一定会发现笔者已经在前文实现了这个结构体，只是没有明说而已。没错，图 4.2.9 已经很直观的给出了结构体成员列表，我们用代码实现即可。打开 `sys_arch.h` 文件，在这个文件里定义这个结构体并声明 `sys_mbox_t`，相关代码如下：

```

#define MBOX_SIZE    16                /* 指定邮箱能够接收的消息数量 */
#define MBOX_NB      8                /* 指定邮箱个数，也就是链表长度 */

/* LwIP 邮箱结构 */
typedef struct stLwIPMBox{
    struct stLwIPMBox    *pstNext;
    HANDLER              hMBox;
    void                 *pvaMsgs[MBOX_SIZE];
} ST_LWIP_MBOX, *PST_LWIP_MBOX;

typedef PST_LWIP_MBOX sys_mbox_t;    /* LwIP 邮箱 */

```

有了 sys_mbox_t，我们就可以建立一个 sys_mbox_t 类型的数组，为链表中的各个节点分配最基本的存储空间。打开 sys_arch.c 文件，建立一个具有全局属性的 ST_LWIP_MBOX 数组，数组大小由 MBOX_NB 宏指定。然后再建立一个全局变量 pstCurFreeMBox，用于保存链表的首地址，建立后的结果如下图所示：

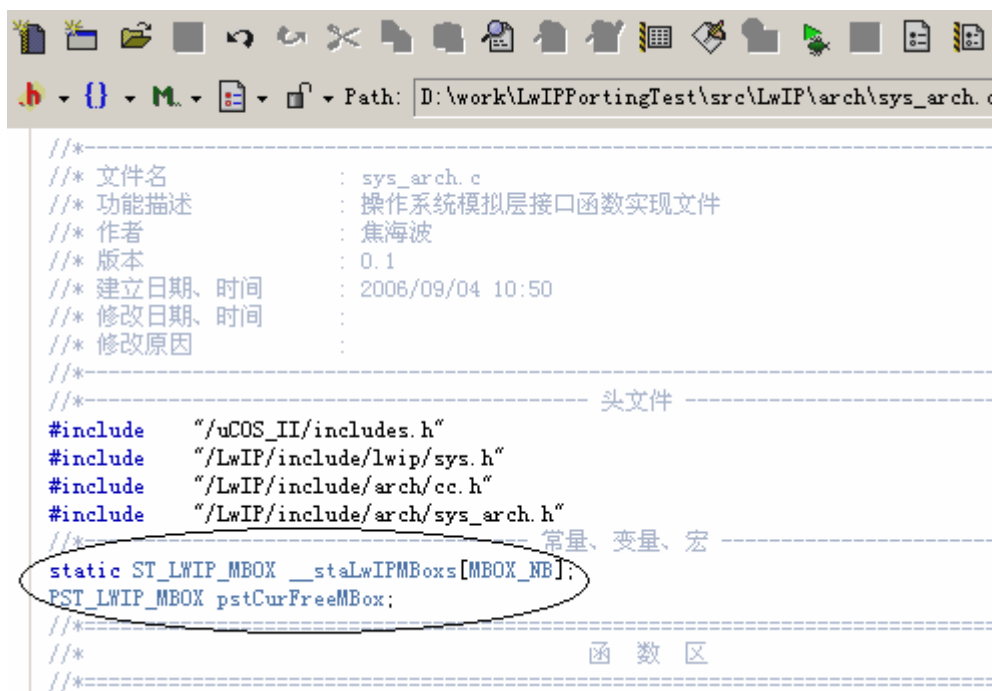


图 4.2.11

接下来的工作就是编写初始化代码——建立链表和邮箱。这些代码在 sys_init() 函数中实现。而移植说明一文已经很清楚的交待了 sys_init() 函数的功能——初始化 sys_arch 层，所以在这个函数里完成建立链表和邮箱的工作是正确的。代码如下：

```

- sys_init()

/*-----
/* 函数名称 : sys_init
/* 功能描述 : 根据 sys_arch.txt 文件建立，功能为初始化 sys_arch 层
/* 入口参数 : 无
/* 出口参数 : 无
/*-----

```



```

void sys_init(void)
{
    u8_t i;

    /* 先把数组清 0
    memset(__staLwIPMBx, 0, sizeof(__staLwIPMBx));

    /* 建立链表和邮箱
    for(i=0; i<(MBOX_NB-1); i++)
    {
        /* 把数组中的各成员链接在一起
        __staLwIPMBx[i].pstNext = &__staLwIPMBx[i+1];
        /*
        建立邮箱，系统必须保证邮箱能够顺利建立，如果出现错误，那是程序 BUG，应
        该在调试阶段排除
        */
        __staLwIPMBx[i].hMBox =
            OSQCreate(__staLwIPMBx[i].pvaMsgs, MBOX_SIZE);
    }

    /* 别忘了数组中最后一个元素，它还没有建立邮箱呢
    __staLwIPMBx[MBOX_NB-1].hMBox =
        OSQCreate(__staLwIPMBx[MBOX_NB-1].pvaMsgs, MBOX_SIZE);

    /* 保存链表首地址
    pstCurFreeMBox = __staLwIPMBx;
}

```

上面给出的 `sys_init()` 函数实现的只是与邮箱操作相关的代码，随着移植的深入，该函数还会陆续增加其它代码，特请读者注意。因为建立链表和邮箱的代码很简单，并且注释很详细，笔者对这些代码就不作详细解释了。

初始工作完成，剩下的工作就是编写移植说明一文要求实现的邮箱操作函数。

先看看 `sys_mbox_new()` 函数。关于这个函数的所有信息在移植说明一文中只有一句话：“建立一个空的邮箱”。非常简单的一句话，在移植说明一文的作者看来它能带给我们的信息已经足够。可对于我们来说，该文作者显然没有交待一个至关重要的问题：如果建立不成功怎么办？这个问题实际上在我们实现信号量操作函数一节就存在。当时只是因为 `sys_sem_new()` 函数采用直接调用 OS 底层函数的方法实现，建立是否成功由底层函数来处理，所以我们没有注意这个问题。现在不同了，我们必须知道 LwIP 是如何处理这个问题的，因为我们要自己实现它，而不再是由别人代劳。要想解决这个问题，我们只有看源码了。打开我们的 ADS 工程文件，CTRL+SHIFT+M，敲入 `sys_mbox_new`，回车，好的，结果出来了，见图 4.2.12。图中圈起来的语句就是我们要找的线索。仔细看看这句对 `sys_mbox_new()` 函数的调用代码，原来它已经考虑了邮箱不能成功建立的情形。如果 `sys_mbox_new()` 函数返回 `SYS_MBOX_NULL`，则意味着对 `sys_mbox_new()` 函数的调用并没有建立一个新邮箱。对于 `sys_mbox_new()` 函数来说，不能成功建立邮箱的情形只有一种，那就是链表长度变为 0，链表不再可用，这时我们的 `sys_mbox_new()` 函数直接返回 `SYS_MBOX_NULL` 即可。

好了，知道了解决这个问题的方法，我们是不是可以动手编写了？不行，我们还有一个小

问题没有确认，那就是 SYS_MBOX_NULL 宏该如何定义？仔细阅读过移植说明一文的读者可能会有印象，在说明一文的结尾提到了 SYS_MBOX_NULL，和它在一块的还有 SYS_SEM_NULL。关于如何定义，说明一文对此只是说这两个宏代表了邮箱（sys_mbox_t）和信号量（sys_sem_t）类型变量的 NULL 值。那么它的 NULL 值到底是什么呢？让我们先看看信号量。信号量的建立是直接调用 OS 底层函数实现的，对此，前文已经交待。那么，OS 底层函数在信号量建立不成功时的返回值又是什么呢？还是要看源码。打开 Os_sem.c 文件，找到 OS_SemCreate() 函数的返回值说明部分——原来建立不成功时的返回值是 (void *)0。那么 SYS_SEM_NULL 的值这下就能确定了——是 (void *)0，而 SYS_MBOX_NULL 的值呢，干脆也指定为 (void *)0 得了，这也是为了保持代码的规范和一致性。

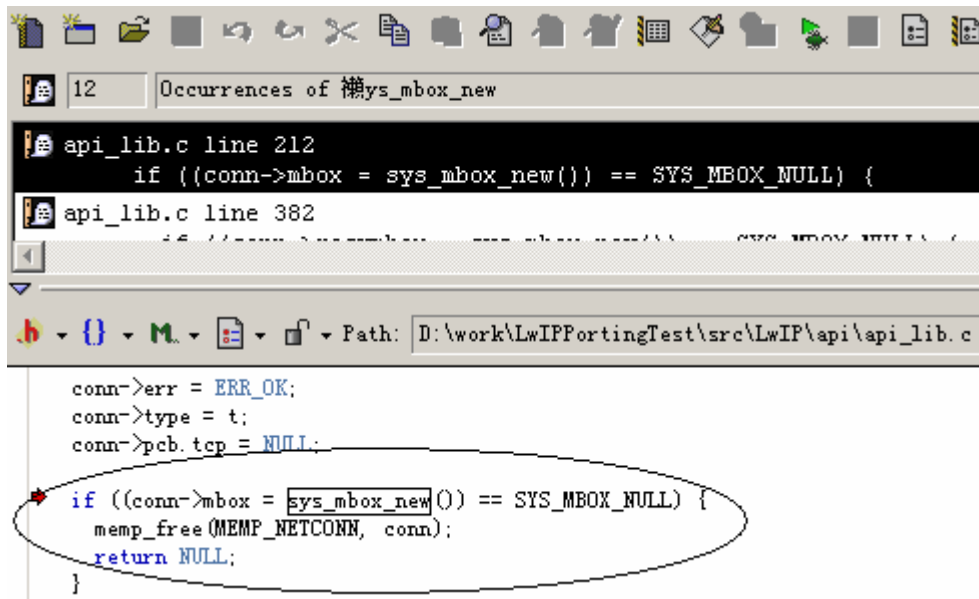


图 4.2.12

经过上面的准备，问题解决的差不多了，现在可以动手实现 sys_mbox_new() 函数了。先打开 sys_arch.h，添加如下语句：

```
#define SYS_MBOX_NULL    (void *)0
#define SYS_SEM_NULL    (void *)0
```

再打开 sys_arch.c，把下面给出的 sys_mbox_new() 函数的实现源码添加进去。

```

- sys_mbox_new()

/*-----
/* 函数名称 : sys_mbox_new
/* 功能描述 : 建立一个空的邮箱
/* 入口参数 : 无
/* 出口参数 : - != SYS_MBOX_NULL : 邮箱申请成功，返回一个指向被申请邮箱的指针
/*           : - = SYS_MBOX_NULL : 邮箱没有申请成功
/*-----

sys_mbox_t sys_mbox_new(void)
{
    PST_LWIP_MBOX __pstMBox = SYS_MBOX_NULL;
  
```

```

#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR      cpu_sr = 0;
#endif

    OS_ENTER_CRITICAL()
    {
        if(pstCurFreeMBox != NULL)
        {
            __pstMBox = pstCurFreeMBox;
            pstCurFreeMBox = __pstMBox->pstNext;
        }
    }
    OS_EXIT_CRITICAL()

    return __pstMBox;
}

```

上面给出的代码中，出现了两个新的函数——OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()。熟悉 uCOS 的读者肯定对此很熟悉。这两个函数负责开系统中断，以保证其间的代码在执行时不被打断。在前面的讲述中读者已经知道，pstCurFreeMBox 指针是一个全局变量。而我们的 LwIP 将支持多线程。因此，为了保证在同一时刻只有一个线程读写该变量，相关读写代码必须加上临界保护（也就是关中断）。关于这两个函数的详细说明不在本文的描述范围之内，这里不再赘述，对其不是很了解的读者可参考相关资料。

接着实现释放邮箱的函数，这个函数很简单，这里只给出源码，不再解释。

```

- sys_mbox_free()

/*-----
/* 函数名称 : sys_mbox_free
/* 功能描述 : 释放邮箱，将邮箱归还给链表
/* 入口参数 : <mbox>[in] 要归还的邮箱
/* 出口参数 : 无
/*-----

void sys_mbox_free(sys_mbox_t mbox)
{
    PST_LWIP_MBOX    __pstMBox = SYS_MBOX_NULL;

    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR      cpu_sr = 0;
    #endif

    /* 为了防止意外情况发生，再主动清空一次邮箱
    OSQFlush(mbox->hMBox);

    OS_ENTER_CRITICAL()
    {

```

```

        mbox->pstNext = pstCurFreeMBox;
        pstCurFreeMBox = mbox;
    }
    OS_EXIT_CRITICAL()
}

```

下一个要实现的函数是 `sys_mbox_post()`，它完成投递消息到指定邮箱的工作。

```

- sys_mbox_post()

/*-----
/* 函数名称 : sys_mbox_post
/* 功能描述 : 将消息投递到指定的邮箱
/* 入口参数 : <mbox>[in] 指定要投递的邮箱
/*           : <msg>[in] 指定要投递的消息
/* 出口参数 : 无
/*-----

void sys_mbox_post(sys_mbox_t mbox, void *msg)
{
    OSAPIQPost(mbox->hMBox, msg);
}

```

这个函数使用了笔者本人自定义的 OS 底层接口函数，这个函数在调用 OS 直接接口 `OSQPost()` 函数时增加了一些容错代码。当 `OSQPost()` 函数返回 `OS_Q_FULL` 时，意味着邮箱已满，无法再投递新的消息。这时，`OSAPIQPost()` 会每隔 100 毫秒调用一次 `OSQPost()` 函数直至投递成功。对于 `OSQPost()` 函数的其它返回值，除 `OS_NO_ERR` 是我们想要的外，其余的实际上都属于编程错误的范畴，对于一个经过严格测试的系统不需要考虑，所以 `OSAPIQPost()` 函数没有对这些返回值做任何处理。下面给出这个函数的源码，以使读者能够理解笔者的处理思路：

```

/*-----
/* 函数名称 : OSAPIQPost
/* 功能描述 : 投递一个消息到指定的消息队列
/* 入口参数 : <hQueue>[in] 指定要投递到的消息队列
/*           : <pvMsg>[in] 指定要投递的消息
/* 出口参数 : 与 OSQPost() 函数返回值除了没有 OS_Q_FULL 之外，其它相同
/*-----

UBYTE OSAPIQPost(HANDLER hQueue, void *pvMsg)
{
    UBYTE __ubErr;

    while((__ubErr = OSQPost(hQueue, pvMsg)) == OS_Q_FULL)
        OSTimeDlyHMSM(0, 0, 0, 100);

    return __ubErr;
}

```

从指定邮箱接收消息的函数源码如下：

```

- sys_arch_mbox_fetch()

/**-----
/** 函数名称 : sys_arch_mbox_fetch
/** 功能描述 : 在指定的邮箱接收消息, 该函数会阻塞线程
/** 入口参数 : <mbox>[in] 指定接收消息的邮箱
/**           : <msg>[out] 结果参数, 保存接收到的消息指针
/**           : <timeout>[in] 指定等待接收的最长时间, 单位为毫秒
/** 出口参数 : - 0: 在指定时间内收到消息
/**           : - SYS_ARCH_TIMEOUT: 在指定时间内没有收到消息
/**-----
u32_t sys_arch_mbox_fetch(sys_mbox_t mbox, void **msg, u32_t timeout)
{
    if(OSAPIQReceive(mbox->hMBox, msg, (u16_t)timeout) == OS_NO_ERR)
        return 0;
    else
        return SYS_ARCH_TIMEOUT;
}

```

这个函数的实现机制与 sys_arch_sem_wait() 函数相同, 这里不再赘述。

4.2.4 实现 sys_thread_new() 函数

对于 uC/OS 来说, 建立一个新线程实际上就是建立一个新任务。因此, sys_thread_new() 函数的实现就变的比较简单了。我们只需调用 uC/OS 提供的建立任务的函数就行了。关键问题解决了, 还有一个小问题没有解决: 线程优先级该如何安排? 移植说明一文没有交待。实际上说明一文的作者也没法交待, 他可不是神仙, 能够预见到我们正在使用的 OS 是啥样的。不过, 这也难不倒我们, 还是老办法, 看源码。我们看看 LwIP 调用这个函数时如何指定的线程优先级。打开 ADS 工程文件, 仍然是 CTRL+SHIFT+M, 查找 sys_thread_new。搜索结果见下图:

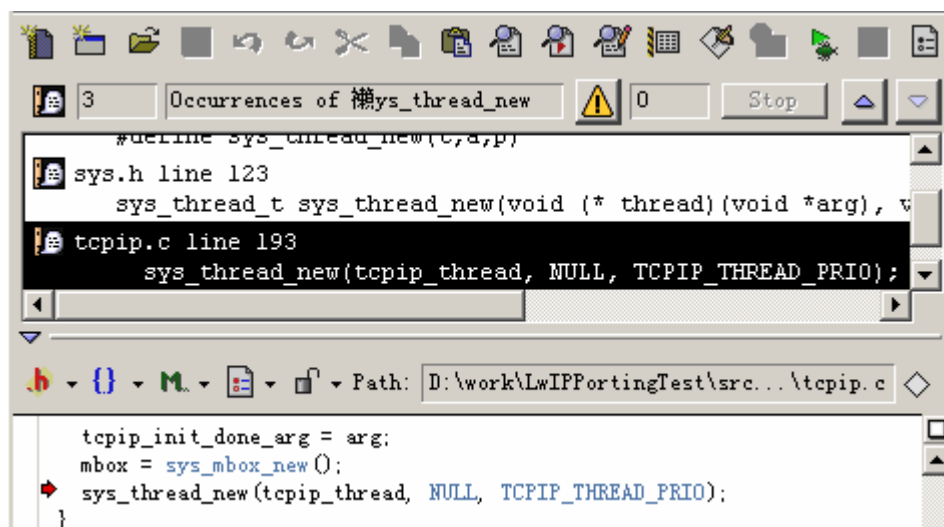


图 4.2.13

图中 TCPIP_THREAD_PRIO 就是分配给该线程的优先级。我们找到它的定义（在 opt.h 文件中）：

```
#define TCPIP_THREAD_PRIO 1
```

它的优先级指定为 1。再看看与这条语句相邻的宏，这个文件还指定了 SLIP 及 PPP 线程的优先级，也是 1。熟悉 uCOS 的读者肯定知道，uCOS 的任务优先级是不能相同的。显然，上面的定义在 uCOS 中是错误的。细心的读者可能还记得，笔者在本文的开头部分已经交待过，我们的移植并不包括 SLIP 及 PPP 部分。因此，SLIP 及 PPP 线程根本不会被系统建立，也就不存在优先级冲突的问题。这样，TCPIP 线程的优先级号可以不用修改。不过，这又产生了一个新的问题。因为按照 uCOS 的优先级规则，各任务的优先级号在 0-63 之间，优先级号越低，任务的优先级越高。TCPIP 线程的优先级定义为 1，那么只有优先级号为 0 的任务才会比 TCPIP 线程的优先级高。这显然不合理。我们要为后续开发留出足够的扩展空间。因此，我们在实现 sys_thread_new() 函数时必须重新调整优先级号。笔者的思路是：指定一个起始优先级号 T_LWIP_THREAD_START_PRIO，然后与 sys_thread_new() 函数中的 prio 参数相加得到新线程的优先级。问题搞清楚了，开始编码吧。打开 sys_arch.h 文件，添加如下语句：

```
#define T_LWIP_THREAD_START_PRIO 7
#define T_LWIP_THREAD_MAX_NB 1
#define T_LWIP_THREAD_STKSIZE 512
```

其中 T_LWIP_THREAD_MAX_NB 宏指定系统最多允许建立几个线程，T_LWIP_THREAD_STKSIZE 指定线程使用的堆栈大小。线程起始优先级号与堆栈大小，读者可根据自己的需求进行调整。再打开 sys_arch.c 文件，先建立任务堆栈，如下图所示：

```
/*----- 头文件 -----
#include "/uCOS_II/includes.h"
#include "/LwIP/include/lwip/sys.h"
#include "/LwIP/include/arch/cc.h"
#include "/LwIP/include/arch/sys_arch.h"
/*----- 常量、变量、宏 -----
static ST_LWIP_MBOX __staLwIPMBoxes[MBOX_NB];
PST_LWIP_MBOX pstCurFreeMBox;

/* LwIP线程使用的堆栈
OS_STK T_LWIP_THREAD_STK[T_LWIP_THREAD_MAX_NB][T_LWIP_THREAD_STKSIZE];
/*----- 函数区 -----
/*----- 函数区 -----
```

图 4.2.14

把 sys_thread_new() 函数添加进去：

```
- sys_thread_new()

/*-----
/* 函数名称 : sys_thread_new
/* 功能描述 : 建立一个新线程
/* 入口参数 : <thread>[in] 新线程的入口地址
/*           : <arg>[in] 传递给新线程的参数
/*           : <prio>[in] 由 LwIP 指定的新线程优先级，这个优先级从 1 开始
/* 出口参数 : 返回线程优先级，注意这与 prio 不同。这个值实际等于
/*           : T_LWIP_THREAD_START_PRIO + prio，如果建立不成功则返回 0
/*-----
sys_thread_t sys_thread_new(void(*thread)(void *arg), void *arg, int prio)
{
```

```

u8_t __ubPrio = 0;

/* 如果优先级定义没有超出系统允许的范围
if(prio > 0 && prio <= T_LWIP_THREAD_MAX_NB)
{
    __ubPrio = T_LWIP_THREAD_START_PRIO + (prio - 1);

    if(OS_NO_ERR ==
        OSTaskCreate(thread,
                      arg,
                      &T_LWIP_THREAD_STK[prio-1][T_LWIP_THREAD_STKSIZE-1],
                      __ubPrio))
        return __ubPrio;
}

return __ubPrio;
}

```

代码编写完了，还需要澄清一件事：T_LWIP_THREAD_MAX_NB 为什么指定为 1？换句话说，为什么我们只允许建立一个线程，单线程的 LwIP 其功能不是不完整么？要回答这个问题，先让我们回忆一下，在讲解优先级定义时，笔者提到的 opt.h 文件，在这个文件里一共包括三个线程优先级号定义，它们分别是 TCPIP 线程、SLIP 线程、PPP 线程。其中 SLIP 线程和 PPP 线程我们目前不需要，所以 LwIP 内部建立的线程只剩下 TCPIP 线程。而移植说明一文所说的——实现多线程——就能够获得的——LwIP 的完整功能，指的就是 TCPIP、SLIP、PPP 协议支持。当我们只需要 TCPIP 时，一个单线程的 LwIP 已经足够。因此，T_LWIP_THREAD_MAX_NB 的值当然就是 1 了。说到这里，聪明的读者一定会发现 sys_thread_new() 函数实际上可以再简化，毕竟单线程实现要比多线程实现简单多了。不过，笔者还是按照多线程需要实现了这个函数，这样做的目的很简单，你我都不是神仙，谁也不会预见到将来是怎么回事。

另外，在这里还需要特别交待的是，使用 LwIP 接口函数建立的应用层线程（或者任务）与 LwIP 内部线程（TCPIP、SLIP、PPP）完全不同。应用层线程就是一个普通的 uCOS 任务，优先级低于 LwIP 线程。它使用 OSTaskCreate()（或 OSTaskCreateExt()）函数建立。如果使用 sys_thread_new() 函数建立，它就会占用非常宝贵的线程资源，包括优先级号以及下文将要阐述的 sys_timeouts 数组资源。

4.2.5 实现 sys_arch_timeouts() 函数

移植说明一文对这个函数的描述能够提供给我们的信息有四条：

- 1、每一个线程都有一个 timeouts 链表；
- 2、要返回的 sys_timeouts 结构保存了 timeouts 链表的首地址；
- 3、该函数在任何情况下都不能返回一个空值（NULL）；
- 4、单线程实现只需要一个 sys_timeouts 结构。

从这些信息中我们可以看出，LwIP 存在几个线程，系统就需要为其准备几个 sys_timeouts 结构。同时，每一个线程所对应的 timeouts 链表在建立之后的首地址必须固定（这是很显然的，否则如何访问这个链表？）。因此，用于保存链表首地址的 sys_timeouts 结构必须与线程一一对应，而且其地址不能改变，以便能够随时得到链表的首地址。显然，要达到这个目的就必须

建立一个具有静态存储时期的变量。为此笔者在 `sys_arch.c` 文件中建立了一个 `sys_timeouts` 结构数组。这个数组具备静态存储时期，在系统运行期间数组成员的地址会一直固定不变。线程优先级号作为进入数组的索引之用，这样每一个线程就能对应一个固定的 `sys_timeouts` 结构。基本思路理清了，现在就可以动手实现了。打开 `sys_arch.c` 文件，把如下语句添加进去，建立 `sys_timeouts` 数组：

```
static struct sys_timeouts __staSysTimeouts[T_LWIP_THREAD_MAX_NB + 1];
```

添加位置如下图所示：



```

/*----- 头文件 -----*/
#include "uCOS_II/includes.h"
#include "LwIP/include/lwip/sys.h"
#include "LwIP/include/arch/cc.h"
#include "LwIP/include/arch/sys_arch.h"
/*----- 常量、变量、宏 -----*/
static ST_LWIP_MBOX __staLwIPMBoxes[MBOX_NB];
PST_LWIP_MBOX pstCurFreeMBox;

/* LwIP线程使用的堆栈
OS_STK T_LWIP_THREAD_STK[T_LWIP_THREAD_MAX_NB][T_LWIP_THREAD_STKSIZE];

/* sys_timeouts数组，用于保存timeouts链表的首地址
static struct sys_timeouts __staSysTimeouts[T_LWIP_THREAD_MAX_NB + 1];
/*----- 函数区 -----*/

```

图 4.2.15

细心的读者可能会发现，笔者定义的 `sys_timeouts` 数组其成员怎么会比 LwIP 线程数量多一个呢？不是说一一对应的么。多这一个有什么用呢？要回答这个问题，我们必须再回到本节的开头。在本节的开头，笔者列出了移植说明一文带给我们的有关这个函数的四条信息，其中第三条是这样说的：该函数在任何情况下都不能返回一个空值（NULL）。这就是说，如果出现任何意外，当我们的函数不能根据线程优先级号进入数组，无法获取一个匹配的 `sys_timeouts` 结构的时候，我们仍然能够返回一个可用的 `sys_timeouts` 结构。这个多出来的数组成员就是为这种情况准备的。下面给出这个函数的实现源码，如下：

```

- sys_arch_timeouts()

/*-----
/* 函数名称 : sys_arch_timeouts
/* 功能描述 : 获取当前线程使用的 sys_timeouts 结构的指针
/* 入口参数 : 无
/* 出口参数 : 返回一个指向当前线程使用的 sys_timeouts 结构的指针
/*-----
struct sys_timeouts *sys_arch_timeouts(void)
{
    u8_t __ubIdx;

    /* 减去起始量获得偏移量，也就是 LwIP 内部的优先级定义
    __ubIdx = OSTCBCur->OSTCBPrio - T_LWIP_THREAD_START_PRIO;

```

```

    /* 当前线程在指定的 LwIP 线程优先级号范围之内
    if(__ubldx >= 0 && __ubldx < T_LWIP_THREAD_MAX_NB)
        return &__staSysTimeouts[__ubldx];

    /* 如果出现意外情况，当前线程并不在指定的 LwIP 线程优先级资源之内，则返回
    /* __staSysTimeouts 数组的最后一个成员
    return &__staSysTimeouts[T_LWIP_THREAD_MAX_NB];
}

```

这个函数在实现时使用了一定的技巧。LwIP 内部线程优先级定义必须在 1 到 T_LWIP_THREAD_MAX_NB（含）之间。线程建立时，内部优先级号是先减 1 后再与 T_LWIP_THREAD_START_PRIO 相加的（参见实现 sys_thread_new() 函数一节）。这样，在 sys_arch_timeouts() 函数中，线程的当前优先级号减去 T_LWIP_THREAD_START_PRIO，就可以直接作为 sys_timeouts 数组索引了。

好了，完成了 sys_arch_timeouts() 函数，我们是不是该转入下一节了？还不行，我们还忘了一件很重要的事：sys_timeouts 数组还没有初始化？每一个数组成员保存的链表首地址应该初始化为 NULL，因为在系统初始化阶段，LwIP 线程的 timeouts 链表还没有建立。在哪里进行初始化呢？还是 sys_init() 函数。找到 sys_init() 函数，在其中添加如下语句：

```

for(i=0; i<sizeof(__staSysTimeouts); i++)
    __staSysTimeouts[i].next = NULL;

```

修改后的 sys_init() 函数如下：

```

void sys_init(void)
{
    u8_t i;

    /* 先把数组清 0
    memset(__staLwIPMBBoxes, 0, sizeof(__staLwIPMBBoxes));

    /* 建立链表和邮箱
    for(i=0; i<(MBOX_NB-1); i++)
    {
        /* 把数组中的各成员链接在一起
        __staLwIPMBBoxes[i].pstNext = &__staLwIPMBBoxes[i+1];

        /* 建立邮箱，系统必须保证邮箱能够顺利建立，如果出现错误，那是程序 BUG，
        /* 应该在调试阶段排除
        __staLwIPMBBoxes[i].hMBox =
            OSQCreate(__staLwIPMBBoxes[i].pvaMsgs, MBOX_SIZE);
    }

    /* 别忘了数组中最后一个元素，它还没有建立邮箱呢
    __staLwIPMBBoxes[MBOX_NB-1].hMBox =
        OSQCreate(__staLwIPMBBoxes[MBOX_NB-1].pvaMsgs, MBOX_SIZE);
}

```

```

    /* 保存链表首地址
    pstCurFreeMBox = __staLwIPMBoxes;

    /* 初始化 sys_timeouts 数组，将每个数组成员保存的链表地址设置为 NULL
    for (i=0; i<sizeof(__staSysTimeouts); i++)
        __staSysTimeouts[i].next = NULL;
    }

```

4.2.6 实现临界保护函数

在实现 LwIP 的临界保护函数之前，让我们先看看 uCOS 的临界保护机制。熟悉 uCOS 的读者一定很清楚，它实现临界保护的方式很简单，就是关中断。进入临界段时关中断，退出时或者直接开中断或者恢复原先的中断开关状态。这与移植说明一文讲述的临界保护函数的功能基本相同。唯一的一点小差别就是——退出临界段时，uCOS 有两种方式（即两个不同的退出函数）可供选择，而 LwIP 只提供——恢复原先的中断开关状态。这样，我们只需将 uCOS 亦改为与 LwIP 相同的退出方式即可共享同一组临界保护函数。

打开 uCOS_II 文件夹下的 os_cpu.h 文件，找到 OS_CRITICAL_METHOD 宏，将其值改为 3，如下图所示：

```

*****
*/

#define OS_CRITICAL_METHOD    3

#if OS_CRITICAL_METHOD == 1
#define OS_ENTER_CRITICAL()  ARMDisableInt(); /* Disable interrupts */
#define OS_EXIT_CRITICAL()   ARMEEnableInt(); /* Restore interrupts */
#endif

#if OS_CRITICAL_METHOD == 3
#define OS_ENTER_CRITICAL()  (cpu_sr = OS_CPU_SR_Save()); /* Disable interrupts */
#define OS_EXIT_CRITICAL()   (OS_CPU_SR_Restore(cpu_sr)); /* Restore interrupts */
#endif

/*
*****
*
*                               ARM Miscellaneous
*
*****
*/

```

图 4.2.16

这时，我们只需直接定义移植说明一文提出的那几个用于临界保护的宏即可实现 LwIP 的临界保护机制，不需要再实现 sys_arch_protect() 和 sys_arch_unprotect() 函数了。读者如果对此不是很明了，只需以 SYS_ARCH_PROTECT 作为关键字搜索整个 LwIP 源码即可明白其中原因。这几个宏的声明语句如下：

```

#define SYS_ARCH_DECL_PROTECT(u1IntStatus)    u32_t    u1IntStatus = 0;
#define SYS_ARCH_PROTECT(u1IntStatus)        (u1IntStatus = OS_CPU_SR_Save())
#define SYS_ARCH_UNPROTECT(u1IntStatus)      (OS_CPU_SR_Restore(u1IntStatus))

```

读者将其直接添加到 cc.h 文件中即可。上面给出的语句中，我们并没有使用移植说明一文提到的 sys_prot_t 类型来声明 u1IntStatus 变量，而是直接使用了 u32_t 来声明。原因很简单，搜遍 LwIP 的所有源码，凡是需要进行保护的临界代码段都是直接使用上面给出的这三个宏，根本没有任何一个地方使用 sys_prot_t 类型（sys.h 文件中也只是在没有定义 SYS_ARCH_PROTECT

宏时使用)。而且，即使声明 `sys_prot_t`，也会将其声明为 `u32_t` 类型（至于为什么将其声明为 `u32_t`，这与 MCU 有关，不明白的读者可参考 MCU 中有关 CPSR 寄存器的相关信息）。因此，笔者就没有任何必要再定义 `sys_prot_t` 类型了，直接使用 `u32_t` 即可。

4.2.7 扫尾——结束操作系统模拟层的编写

经过了前 6 节的努力，我们已经完成了所有模拟层接口函数的编写工作。但是还有一些与接口函数无关的工作需要完成，这一节就完成这些工作。完成这些工作后，操作系统模拟层就可以交付使用了。

首先要先完成 `cc.h` 文件，这个文件还有几个宏没有定义。第一个是与编译器密切相关的结构封装宏。我们知道，ARM 系统默认以 4 字节对齐，如果我们设计的结构不能够 4 字节对齐，那么 ARM 编译器会重新调整结构以达到 4 字节对齐，因为这样的安排会提高访问效率。举个例子来说，我们设计如下一个结构：

```
typedef struct {
    char a;
    short b;
    int c;
} sttest;
```

然后使用代码检查一下这个结构的大小及各个成员的存储地址，如下：

```
int main(void)
{
    char stsize;
    sttest stTest;
    void *addra, *addrb, *addrc;

    stsize = sizeof(sttest);

    addr = &stTest.a;
    addr = &stTest.b;
    addr = &stTest.c;
}
```

执行结果如下图所示：

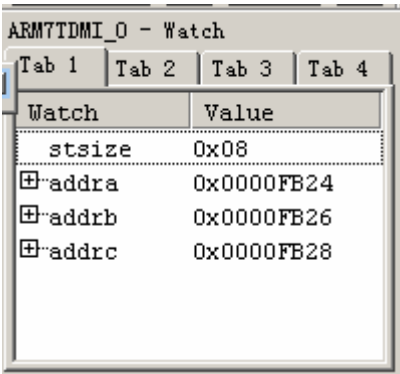


图 4.2.17

很明显，编译器对结构作了调整，否则结构体的大小应该是 7。多出的这一个字节使得 `b` 字段的存储地址正好处于 16 位的边界，而 `c` 字段则正好处于 32 位的边界。前面已经说了，编译器

这样作的目的是为了提_高访问效率。但是，有时候我们需要按照实际的成员边界访问结构体，比如保存网络数据包的结构体。要达到这样的目的就必须关闭编译器的这一优化选项，那么如何关闭呢？答案就在编译器的数据手册里。笔者使用的 C/C++编译器由集成开发环境 ADS 提供（armcc/armcpp/tcc/tcpp）。我们在安装 ADS 时安装程序会询问是否安装相关文档，这里面就包括编译器的数据手册。打开 ADS 安装目录，在 PDF 文件夹下找到 ADS_CompilerGuide_D.pdf，它就是 **ARM 编译器的数据手册**。在这个文件的第 70 页，读者必须仔细阅读__packed 关键字的相关内容，答案就在这里。数据手册里有的内容，笔者在这里就不再赘述，我们只是举个简单的例子讲解__packed 关键字的用法及用途。还是上面那个例子，只不过在定义结构体时增加了__packed 关键字：

```
__packed typedef struct {
    char a;
    short b;
    int c;
} sttest;
```

这时再看执行结果，是我们想要的结果了吧，见下图：

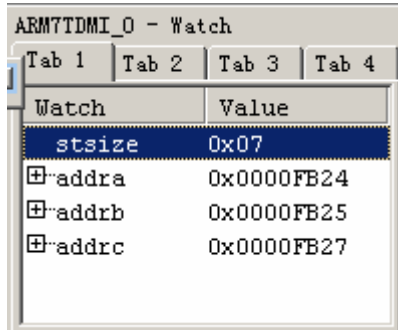


图 4.2.18

cc.h 文件中要定义的这几个结构体封装宏就是要实现这样的目的。那么如何实现呢？还是老办法——搜，看看这几个宏是如何使用的不就明确了么。从搜索结果中随便打开其中的一个文件（下图）：

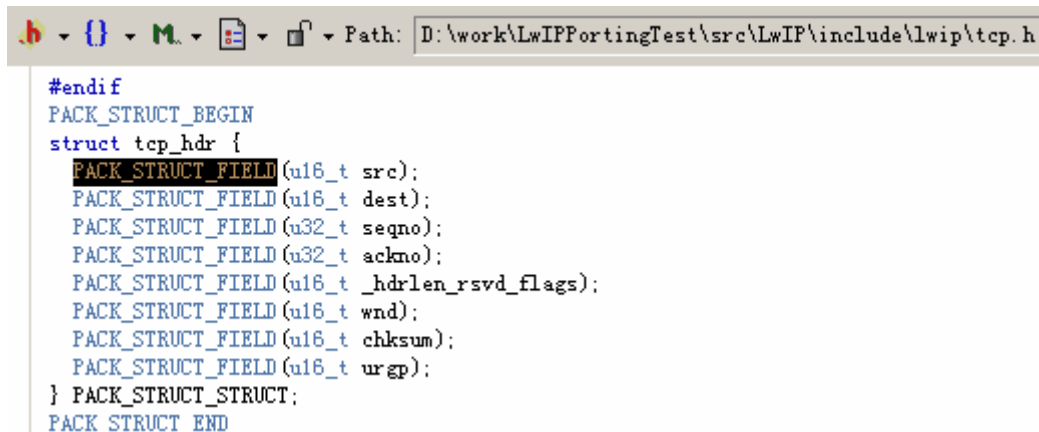


图 4.2.19

明白了吧，参照上面给出的例子，这几个宏如何定义读者一定很清楚了。如果读者还不明白，建议读者找本 C 语言的书系统学习一遍后再回来阅读本文。下面给出这几个宏的定义语句：

```
#define PACK_STRUCT_FIELD(x)    __packed x
#define PACK_STRUCT_STRUCT
#define PACK_STRUCT_BEGIN      __packed
```

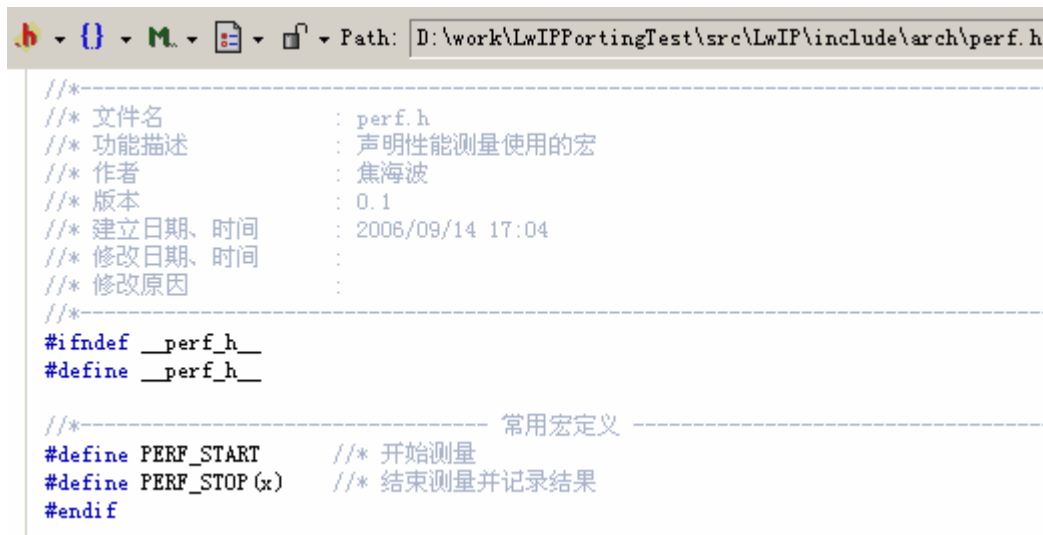
```
#define PACK_STRUCT_END
```

下一个要定义的是与平台相关的调试输出宏，这项功能对于我们来说可有可无，笔者为图省事，就不实现它了。

按照移植说明，我们要么增加一条宏定义，要么包含一个头文件<errno.h>，这个文件定义了标准的*nix 错误编码。笔者选择增加一条宏定义语句：

```
#define LWIP_PROVIDE_ERRNO
```

至此，cc.h 文件彻底完成。下一个文件是 perf.h，这个文件用于声明性能测量使用的宏。笔者暂时没有这方面的需要，因此在这个文件里只是对其进行了简单声明，并没有指定具体的值（下图）：



```
/*-----  
/* 文件名          : perf.h  
/* 功能描述        : 声明性能测量使用的宏  
/* 作者            : 焦海波  
/* 版本            : 0.1  
/* 建立日期、时间  : 2006/09/14 17:04  
/* 修改日期、时间  :  
/* 修改原因        :  
/*-----  
#ifndef __perf_h_  
#define __perf_h_  
  
/*----- 常用宏定义 -----  
#define PERF_START      /* 开始测量  
#define PERF_STOP(x)    /* 结束测量并记录结果  
#endif
```

图 4.2.20

好了，整个操作系统模拟层的编写工作至此完成。下面该考虑如何使用 LwIP 及底层网络驱动的事情了。

注：文件夹\LwIPPortingTest_4\LwIPPortingTest_4_2 中保存着我们到目前为止的工作成果，读者可根据需要使用。

5 LwIP 接口——初始设置及网络驱动

从这一章将开始我们的 LwIP 应用之旅。按照一般顺序，先讲解 LwIP 的初始设置。LwIP 源码文件携带的文档 rawapi.txt 描述了 LwIP 初始化的一般过程。不过，这个文档并不是一个很好的参考。我们必须在使用这篇文档的同时仔细研究相关源码才能理清其初始化顺序。当然，如果读者能够在网上找到一些相关的应用源码，那么这项工作就变得较为简单。sourceforge.net 就是一个很好的网站，上面提供了大量源码可供下载，读者没事经常到上面看看，相信会有收获的。不过对于本文的读者来说，现在就没有必要再找 LwIP 的相关应用源码了，笔者会在接下来的篇幅中给出。

5.1 准备工作——建立 LwIP 入口函数文件

首先，在 ADS 工程的 LwIP 组下面建立一个新文件，文件名为 LwIPEntry.c，如下图所示：

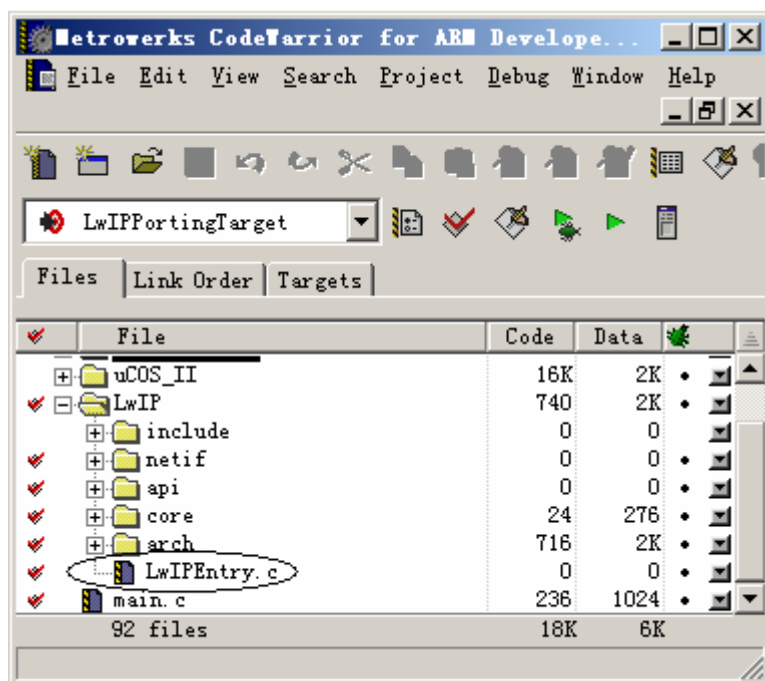


图 5.1.1

在这个文件里添加如下代码，建立入口函数的基本结构。笔者将在随后的几节逐步完善这个结构：

```
#include    "/uCOS_II/includes.h"

/*-----
/* 函数名称 : T_LwIPEntry
/* 功能描述 : LwIP 入口函数
/* 入口参数 : <pvArg>[in/out] 保留参数
/* 出口参数 : 无
/*-----

void T_LwIPEntry(void * pvArg)
{
    /* 初始化 LwIP
    __ilvInitLwIP();

    /* 设置 LwIP，包括添加配置网络接口、建立接收任务等工作
    __ilvSetLwIP();

    /*
    在这里建立 LwIP 的应用，比如可以建立一个 TCP 服务器等待接收客户端的连接请求并作出响应
    .....
    */
}
```

5.2 __ilvInitLwIP()

这个函数负责完成 LwIP 最基本的初始化工作。这些工作涉及 LwIP 使用的内存区、PBUF、PCB

(UDP/TCP) 以及 OS 模拟层等各个方面。其实现主要参照了 rawapi.txt 一文，其源码如下：

```

/*-----
/* 函数名称 : __ilvInitLwIP
/* 功能描述 : 完成 LwIP 最基本的初始化工作
/* 入口参数 : 无
/* 出口参数 : 无
/*-----
__inline void __ilvInitLwIP(void)
{
    sys_init();

    mem_init();

    memp_init();

    pbuf_init();

    tcpip_init(NULL, NULL);
}

```

这个函数与 rawapi 一文描述的系统初始化过程有些差别：其一、有几个在 rawapi 一文中提到的函数并没有被调用；其二、多了一个文中并没有提到的函数——tcpip_init()。先说说第一个差别，在这些未被调用的函数中，除 ip_init()、udp_init()、tcp_init() 之外的其它函数均属于 LwIP 初始配置的范畴，它们将在 __ilvSetLwIP() 函数中被调用，因此它们并没有出现在 __ilvInitLwIP() 函数中。至于第二个问题，tcpip_init() 函数是 LwIP 提供的一个 API，负责完成 TCPIP 层的初始化及处理线程的建立工作。它建立的线程在启动时会调用上文提到的 ip_init()、udp_init()、tcp_init() 这三个函数以完成相关的初始设置工作，因此笔者使用了 tcpip_init() 函数来代替对这三个函数的直接调用。

5.3 __ilvSetLwIP()

这个函数完成 LwIP 的初始配置工作，实现网络驱动与 LwIP 的接口。它是整个网络系统的关键一环，是网络系统得以顺利运转的源动力。它会告诉 LwIP 网络数据的发送出口，建立网络数据的接收入口并实现接收入口与 LwIP 处理入口的对接。当底层网络收发设备进入正常工作状态，网络数据能够正常流入流出 LwIP 后，这个函数进行的工作才会宣告结束，其要完成的具体工作详见图 5.3.2。函数源码如下：

```

/*-----
/* 函数名称 : __ilvSetLwIP
/* 功能描述 : 设置 LwIP，包括添加配置网络接口、建立接收任务等工作
/* 入口参数 : 无
/* 出口参数 : 无
/*-----
__inline void __ilvSetLwIP(void)
{
    extern err_t ethernetif_init(struct netif *stNetif);
}

```

```

struct ip_addr __stIpAddr, __stNetMask, __stGateway;

/** 注意, 这里使用了 static 关键字, 这样作的目的是——当这个函数结束执行后, 我们
/** 定义的网络接口仍然能够使用, 否则 LwIP 无法工作。当然, 我们可以不使用 static,
/** 而是在函数外部定义该网络接口, 这样作具有同样的效果。
static struct netif __stEMACNetif;

/** 初始化缺省网络接口(即习惯意义上的网卡)及网络接口链表(即 netif 结构体链表),
/** 根据函数说明, 它必须率先被调用
netif_init();

/** 建立并配置 EMAC 接口
IP4_ADDR(&__stIpAddr, IP_ADDR_0, IP_ADDR_1, IP_ADDR_2, IP_ADDR_3);

IP4_ADDR(&__stNetMask, NET_MASK0, NET_MASK1, NET_MASK2, NET_MASK3);
IP4_ADDR(&__stGateway, GATEWAY_ADDR_0,
          GATEWAY_ADDR_1, GATEWAY_ADDR_2, GATEWAY_ADDR_3);

netif_add(&__stEMACNetif, &__stIpAddr, &__stNetMask,
          &__stGateway, NULL, ethernetif_init, tcpip_input);

netif_set_default(&__stEMACNetif);
netif_set_up(&__stEMACNetif);
}

```

读者不难看出, 这个函数的基本结构及实现流程与 rawapi 一文的描述大体相同。存在的微小差别并没有影响该函数完成 rawapi 一文要求完成的工作。总结这些差别共有三处: 一处是在 `netif_add()` 函数之前多了一个 `netif_init()` 函数; 另一处是传递给 `netif_add()` 函数的 `input` 指针并没有指向 `ip_input()`; 第三处是在函数的结尾少了 `dhcp_start()` 函数。先说第一处, 多的这个函数并不是真的多了, 而是在调用 `netif_add()` 等函数之前必须要先调用的。读者可以找到这个函数的原型声明看看有关它的描述就会明白笔者之所以这样说的原因了 (见图 5.3.1)。

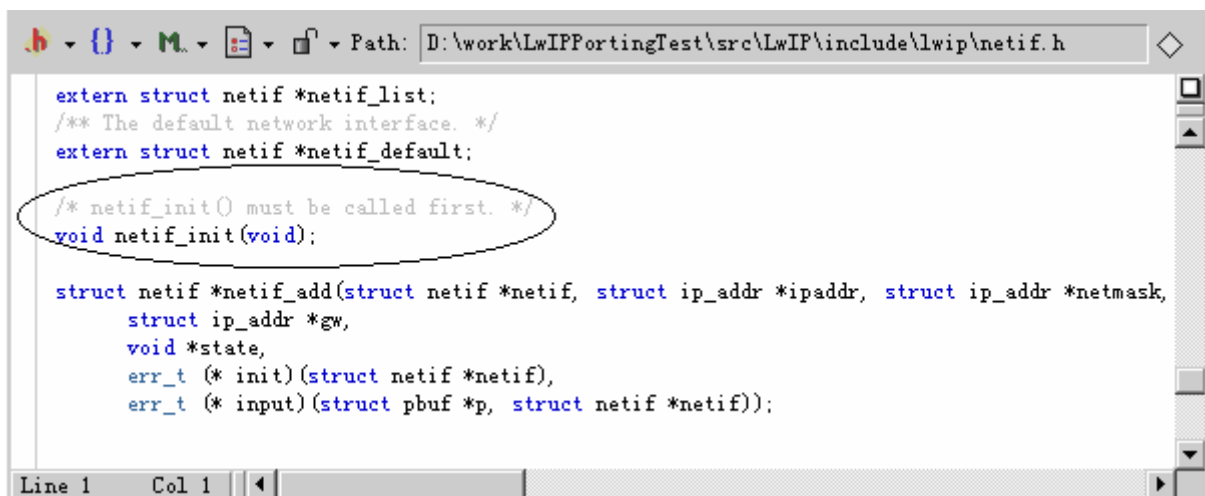


图 5.3.1

实际上，LwIP 提供多网络接口支持。用户在一个系统中可以添加多个不同 IP 地址及 MAC 地址的网络接口，这些接口可以独立处理网络通讯而不互相干扰。LwIP 把它们以链表的方式组织起来。`netif_add()` 函数负责把网络接口添加到链表中，而这个链表的初始化工作就是由 `netif_init()` 函数完成的。讲到这里，读者应该明白为什么要先调用 `netif_init()` 函数的真正原因了吧，链表不先进行初始化，`netif_add()` 函数是不能调用的。按道理这么一个必不可少的函数竟然没有出现在 `rawapi` 一文中实在是让人难以理解。不过纵观 LwIP 一直以来就很糟糕的文档（文档很少而且时间也很早了，不能与软件版本同步），就不难理解 `rawapi` 一文的作者为什么会出现如此的疏漏了。

关于第二处差别，其原因就稍微复杂一些。`netif_add()` 函数在调用时需要向其传递两个函数指针作为参数。一个是 `init` 指针，其指向的函数完成底层网络设备的初始化工作，`rawapi` 一文对此已经有所交待，这里不再赘述。另一个是 `input` 指针，`rawapi` 一文明确说明其应该指向 `ip_input()` 函数，而笔者却将其指向了 `tcpip_input()`。这到底是为什么呢？要明白这个问题，我们得先看看 `tcpip_input()` 函数的源码，如下：

```
err_t
tcpip_input(struct pbuf *p, struct netif *inp)
{
    struct tcpip_msg *msg;

    msg = memp_malloc(MEMP_TCPIP_MSG);
    if (msg == NULL) {
        pbuf_free(p);
        return ERR_MEM;
    }

    msg->type = TCPIP_MSG_INPUT;
    msg->msg.inp.p = p;
    msg->msg.inp.netif = inp;
    sys_mbox_post(mbox, msg);
    return ERR_OK;
}
```

这个函数中最关键的地方就是上面带有下划线的那条语句，其很直观的表明了这个函数的真正用途：投递消息。那么这个要投递的消息到底来自哪里，又到哪里去呢？答案就在 `mbox` 里。前文已经讲过，`mbox` 是一个邮箱，其用于不同任务之间的通讯。我们只要找到哪些任务在使用这个邮箱，问题就不难解决了。实际上，这个邮箱正由 TCPIP 处理线程使用，它与 `tcpip_input()` 函数就在同一个文件里，函数名为 `tcpip_thread()`。讲到这里，细心的读者一定会有印象，我们在 4.2.4 节曾经提到过它。它负责处理与 TCPIP 协议族相关的所有网络通讯，其源码如下：

```
static void
tcpip_thread(void *arg)
{
    struct tcpip_msg *msg;

    .....

    while (1) { /* MAIN Loop */
        sys_mbox_fetch(mbox, (void *)&msg); (1)
        switch (msg->type) {
```

```

    case TCPIP_MSG_API:
        .....

    case TCPIP_MSG_INPUT:                                     (2)
        .....

        ip_input(msg->msg.inp.p, msg->msg.inp.netif);         (3)
    case TCPIP_MSG_CALLBACK:
        .....

    default:
        break;
    }
    .....
}
}

```

上面给出了这个函数中最能说明问题的几个地方，笔者分别用 (1) (2) (3) 进行了标注。标注为 (1) 的这条语句回答了 tcpip_input() 函数投递的消息到哪里去的问题，而第 (2)、第 (3) 句则说明了为什么没有把 netif_add() 函数中的 input 指针指向 ip_input() 函数的原因。在 tcpip_input() 函数中，LwIP 把 tcpip_msg 结构的 type 字段赋值为 TCPIP_MSG_INPUT。因此，只要是 tcpip_input() 函数投递的消息，tcpip_thread 总是会将其传递给 ip_input() 函数处理，这实际上就相当于直接调用 ip_input() 函数了。在我们的实现中，从底层网络设备接收的最原始的数据包去掉以太网包头后就是 IP 包，而最先接管 IP 包的函数就是 ip_input()。如果我们采用函数调用的方式把底层接收函数得到的 IP 包作为参数传递给 ip_input()，那么，我们的接收与处理将肯定在一个线程里面。这样只要处理流程稍微被延迟，接收就会被阻塞，直接造成频繁丢包、响应不及时等严重问题。因此，接收与协议处理必须分开。LwIP 的作者显然已经考虑到了这一点，他为我们提供了 tcpip_input() 函数来处理这个问题，虽然他并没有在 rawapi 一文中说明。讲到这里，读者应该知道 tcpip_input() 函数投递的消息从哪里来的答案了吧，没错，它们来自于由底层网络驱动组成的接收线程。我们在编写网络驱动时，其接收部分以任务的形式创建。数据包到达后，去掉以太网包头得到 IP 包，然后直接调用 tcpip_input() 函数将其投递到 mbox 邮箱。投递结束，接收任务继续下一个数据包的接收，而被投递得 IP 包将由 TCPIP 线程继续处理。这样，即使某个 IP 包的处理时间过长也不会造成频繁丢包现象的发生。关于底层网络驱动在哪里实现、如何实现的问题，笔者将在讲解 ethernetif_init() 函数时详细解答，这里就不再赘述了。

最后一处差别，关于为什么少了 dhcp_start() 函数的原因就很简单了，因为在笔者的实现中不需要 DHCP，所以就没有必要调用这个函数了。

笔者在本节的开头已经对 __ilvSetLwIP() 函数要完成的工作作了一个大致的描述，图 5.3.2 按照该函数的实现流程把这些工作分解为四部分。这四部分工作分别由四个紧密相连的函数来完成，其先后顺序不能更改。正如前文所述，netif_init() 函数必须最先被调用，它不仅完成网络接口链表的初始化工作，还要完成缺省网络接口的初始化工作，而这项工作正是在调用 netif_set_default() 函数之前的准备工作。链表被初始化后，netif_add() 函数率先被调用，完成整个 LwIP 初始化进程中最复杂、最关键的几项工作。这些工作相当于我们为 LwIP 提供了一个引擎并带动其高速运转起来。也就是说，如果我们没有进行这些工作，那么 LwIP 就像一个失去了引擎的汽车，只能趴在那里烂成一堆废铁。就是这么关键的一个函数，它的源码却一点也不复杂，甚至可以认为相当简单，为什么会这样呢？答案就在 init 参数指向的 ethernetif_init() 函数里，这个函数完成了绝大部分工作，netif_add() 函数仅对其简单调用即可，源码当然就不复杂了。确切的说，netif_add() 函数要完成的四部分工作（见图 5.3.2 中 netif_add() 部分），前三部分由 ethernetif_init() 函数完成，第四部分加入链表的工作，才真正由 netif_add() 函数完成。下一节我们将详解 ethernetif_init() 函数，这里不再赘述。在 netif_add() 函数完成第一个网络接口的配置及加入链表的工作之后，我们可以

继续调用该函数添加其它接口直至添加完毕。之后，**注册**缺省网络接口。这项工作在系统仅有一个接口的情况下也不能省略，否则将增加读者应用层程序的复杂程度。在所有工作按部就班的完成之后，就可以**使能**所有已添加的网络接口（`netif_set_up()`），告诉 LwIP 开始处理网络通讯。

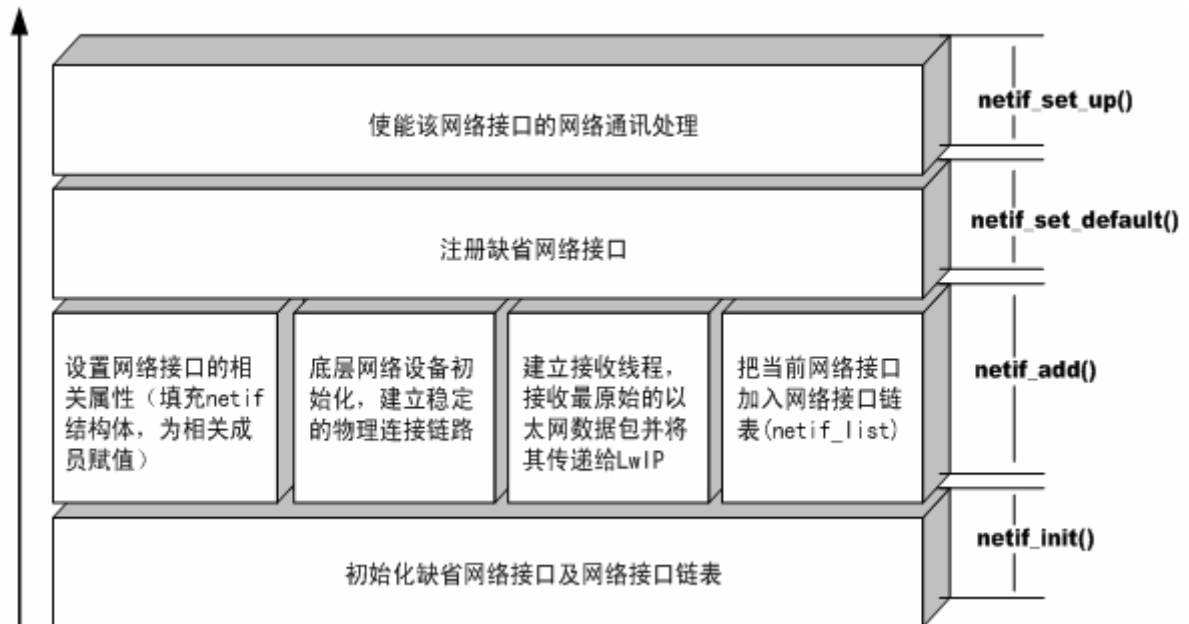


图 5.3.2

在结束本节之前，我们把这个文件编译一下，看看在增加一些代码之后，是不是还有一些工作要做，比如缺少某些数据类型的定义和声明等。编译的结果还真没有出乎我们的意料，足足有 23 个错误，主要问题就集中在缺少宏定义及结构体声明这两个方面。这些缺少的定义和声明共分为两部分，其中一部分是笔者本人自定义的，另一部分则由 LwIP 提供，我们只需把相关定义文件 include 进来即可。先说自定义宏部分，`__ilvSetLwIP()` 函数中自定义宏按类别分共有三个，它们分别是 `IP_ADDR_0—IP_ADDR_3`、`NET_MASK_0—NET_MASK_3`、`GATEWAY_ADDR_0—GATEWAY_ADDR_3`。这三种自定义宏代表不同的网络配置参数。其中，`IP_ADDR_x` 代表 IP 地址的四个部分，`NET_MASK_x` 代表子网掩码的四个部分，而 `GATEWAY_ADDR_x` 则代表网关地址的四个部分。读者可根据自己所处网络的实际情况对其自行定义。在笔者本人的实现中，这几个宏的定义如下（参见 `LwIPEntry.c`）：

```
/* IP 地址
#define IP_ADDR_0 ..... IP_ADDR_3  它们分别为 192、168、10、72

/* 网关地址
#define GATEWAY_ADDR_0 ..... GATEWAY_ADDR_3  它们分别为 192、168、10、1

/* 掩码地址
#define NET_MASK_0 ..... NET_MASK_3  它们分别为 255、255、255、0
```

自定义部分完成，接下来的工作就是 include 相关头文件。这项工作很简单，无非就是搜索包含宏或结构体声明的头文件，然后 include，笔者对此不再赘述，这里只给出最终结果，如下：

```
#include    "/LwIP/include/lwip/netif.h"
#include    "/LwIP/include/lwip/tcpip.h"
```

这时再编译这个文件就可以编译成功了。

5.4 ethernetif_init() ——初始化底层接口

我们即将进行的工作相对于前面完成的操作系统模拟层来说，LwIP 的作者已经为我们完成了绝大部分，我们只需在作者设计好的框架内完成与底层硬件相关的部分即可，没有多少可自由发挥的空间。因此，笔者下面的讲述将严格按照整个框架的实现顺序进行（见图 5.4.1），从 ethernetif_init() 开始，逐一讲解每个函数。

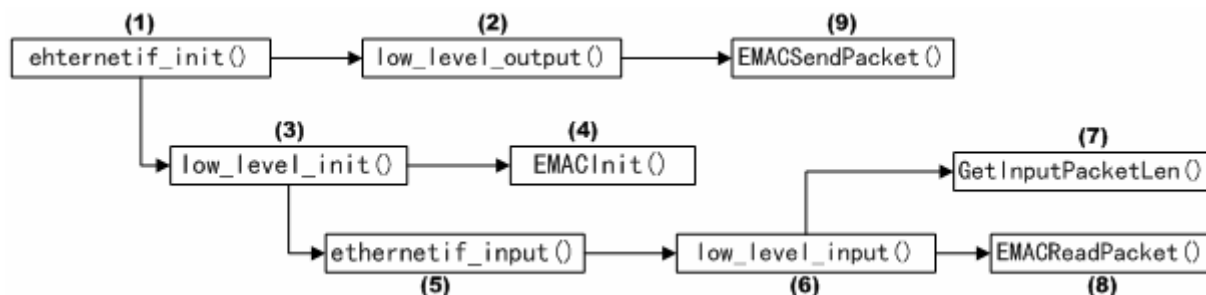


图 5.4.1

5.4.1 ethernetif_init() 函数分析

这个函数的源码如下 (ethernetif.c)：

```

err_t
ethernetif_init(struct netif *netif)
{
    static struct ethernetif ethernetif;                                (1)

    netif->state = &ethernetif;
    netif->name[0] = IFNAME0;
    netif->name[1] = IFNAME1;
    netif->output = ethernetif_output;                                  (2)
    netif->linkoutput = low_level_output;                               (3)

    ethernetif.ethaddr = (struct eth_addr *)&(netif->hwaddr[0]);      (4)

    low_level_init(netif);                                              (5)

    etharp_init();                                                       (6)

    sys_timeout(ARP_TMR_INTERVAL, arp_timer, NULL);                   (7)

    return ERR_OK;
}
  
```

- (1) **ethernetif** 是一个结构体，用于描述底层网络硬件设备（即通常所说的网卡）。这个结构体唯一不可或缺的成员就是网卡的 MAC 地址，它是 LwIP 用于响应 ARP 查询的核心数据。除了它，读者如果没有其它特殊需求，就可以不用为这个结构体增加其它成员

了。

- (2) 向 LwIP 注册发送函数。注意，这个函数并不是真正的发送函数，它是通过调用第 (3) 句注册的函数完成信息包发送的。
- (3) 向 LwIP 注册链路层发送函数，该函数完成实际的信息包发送。这个函数需要读者结合实际的硬件情况亲自编写实现，LwIP 仅提供了一个参考结构。
- (4) ethernetif->ethaddr 指针指向 netif 中保存的网卡 MAC 地址。
- (5) 网卡初始化，建立稳定的物理连接链路并建立接收线程（见图 5.4.1 中的第 (3)、(4) 步）。这个函数直接与底层硬件打交道，LwIP 仅能为其提供一个参考结构，需要我们结合实际硬件情况重新设计实现。
- (6) 对于这个函数细心的读者一定会有印象，rawapi 一文曾经提到过它，它应当出现在系统初始化序列中（详见 5.2 节）。但是，当时我们并没有把它放到 __ilvInitLwIP() 函数内，现在读者应该明白为什么会这样了吧，原来 LwIP 的作者早有安排。有关它的功能描述详见 rawapi 一文。
- (7) 开启 ARP 表的定时更新处理。这个函数由 LwIP 提供，不需要任何修改。

笔者给出的源码与 LwIP 提供的源码相比几乎没有差别，笔者只是去掉了其中的调试输出语句，这在本质上没有什么不同。笔者之所以着重说明这一点，原因很简单，因为这个函数是 LwIP 提供给我们进行底层接口设计的路线图，任何对该函数流程、结构的修改都有可能影响整个路线图，所以读者最好不要随便修改。图 5.4.1 很形象的描绘了这个路线图，我们接下来的工作将沿此进行。

5.4.2 low_level_output() —— 链路层发送函数

先给出该函数的实现源码，然后再分析，如下：

```
static err_t low_level_output(struct netif *pstNetif, struct pbuf *pstPbuf)
{
    struct pbuf *__pstSendPbuf = pstPbuf;                                (1)
    static HANDLER __hBlockOutput = NULL;
    err_t __errReturn = ERR_OK;

    if(__hBlockOutput == NULL)                                           (2)
        __hBlockOutput = OSAPIBlockNew(5);

    if(OS_NO_ERR == OSAPIBlockEnter(__hBlockOutput, 2000))              (3)
    {
        for(; __pstSendPbuf!=NULL; __pstSendPbuf=__pstSendPbuf->next)    (4)
        {
            if(!EMACSendPacket(__pstSendPbuf->payload,                    (5)
                                __pstSendPbuf->len,
                                (__pstSendPbuf->next == NULL)))
            {
                __errReturn = ~ERR_OK;                                     (6)
                break;
            }
        }
    }
}
```

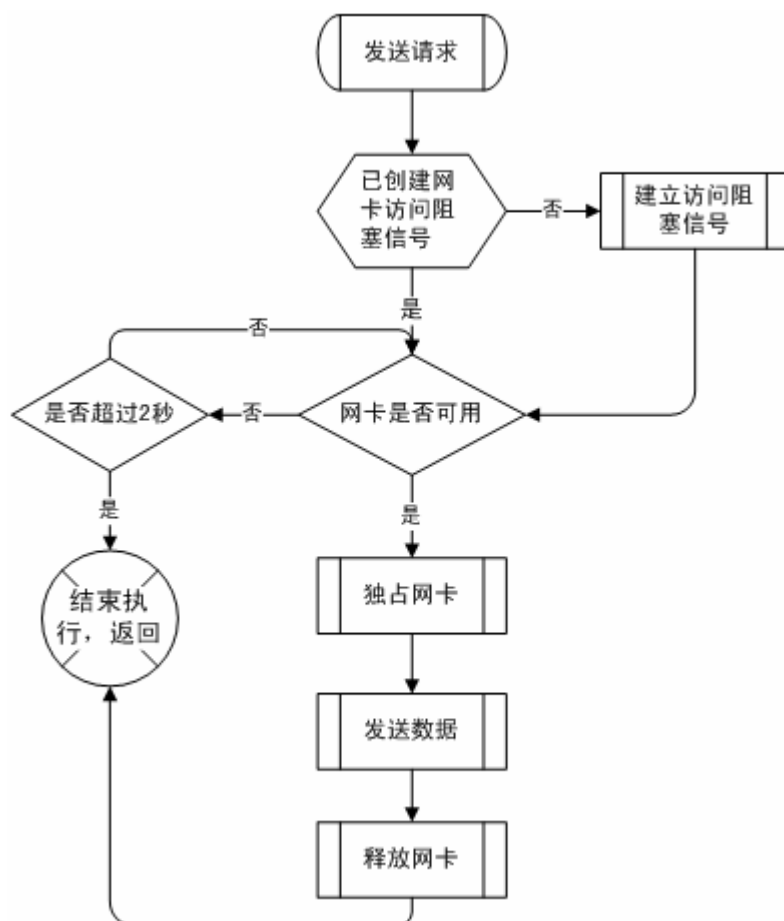


```
        OSAPIBlockExit(__hBlockOutput);                (7)
    }

    return __errReturn;
}
```

- (1) 关于 pbuf 结构的详细说明参看讲解 LwIP 实现细节的数据手册。其下载地址为：
<http://www.sics.se/~adam/lwip/doc/lwip.pdf>
不喜欢看英文的读者朋友可以到我的 G00GLE 论坛上下载笔者自己翻译的中文数据手册
(翻译得不好，权当参考吧)：
http://groups.google.com/group/marsstory/browse_thread/thread/6cf85ea241094106/cab64043c02361b3#cab64043c02361b3
- (2) 如果 __hBlockOutput 为 NULL 则表明这是第一次调用这个函数，对网卡的访问阻塞信号还没有建立，建立之后函数才能继续执行。其中 OSAPIBlockNew() 函数是笔者本人编写的 uCOS 的接口 API，前文已经对此有所交待，它负责建立并返回一个新的互斥型信号量。不熟悉的读者可参考 uCOS 中互斥型信号量的相关资料。
- (3) 申请网卡的使用权。如果当前其它任务还在使用，则最长等待两秒。这期间如果网卡可用，则立即独占并阻塞后续申请，如果不可用，则不再等待，立即退出函数的执行，返回调用者。
- (4)
- (5)
- (6) (4) 到 (6) 这三句完成数据包的发送。因为 pbuf 是一个链表，该链表的每一个节点都仅包含要发送数据包的一部分，因此必须使用一个 for 循环完成整个数据包的发送。for 循环中调用的 EMACSendPacket() 函数完成实际的发送工作。有关它的详细信息笔者将在 5.4.9 节讲解。
- (7) 释放网卡，结束占用

在结束本节之前，如果还有读者对这个函数的整体流程不是很清晰，请参见图 5.4.2，笔者对此不再赘述。

图 5.4.2 `low_level_output()`

5.4.3 `low_level_init()` ——网卡初始化函数

虽然这个函数从代码长度看并不复杂，但它要完成的实际工作量却很大，其源码如下：

```

static void
low_level_init(struct netif *pstNetif)
{
    UBYTE    __ubOldPrio;
    UBYTE    __ubErr;
    pstNetif->hwaddr_len = NETIF_MAX_HWADDR_LEN;           (1)

    pstNetif->hwaddr[0] = 0xBC;                               (2)
    pstNetif->hwaddr[1] = 0x20;
    pstNetif->hwaddr[2] = 0x06;
    pstNetif->hwaddr[3] = 0x09;
    pstNetif->hwaddr[4] = 0x30;
    pstNetif->hwaddr[5] = 0x11;

    pstNetif->mtu = 1500;                                     (3)

    pstNetif->flags = NETIF_FLAG_BROADCAST;                  (4)

```

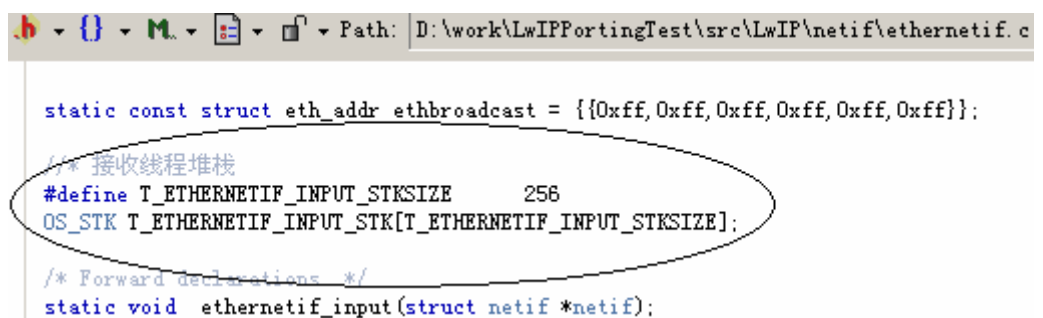
```

__ubOldPrio = OSTCBCur->OSTCBPrio;                                (5)
__ubErr = OSTaskChangePrio(OS_PRIO_SELF, OS_IDLE_PRIO - 1);      (6)
EMACInit();                                                       (7)
if(__ubErr == OS_NO_ERR)                                           (8)
    OSTaskChangePrio(OS_PRIO_SELF, __ubOldPrio);

OSTaskCreate(ethernetif_input,                                    (9)
             pstNetif,                                           (10)
             &T_ETHERNETIF_INPUT_STK[T_ETHERNETIF_INPUT_STKSIZE-1], (11)
             6);                                                  (12)
}

```

- (1) 设置网卡 MAC 地址的长度。这个长度由 LwIP 定义的宏 NETIF_MAX_HWADDR_LEN 指定，长度值为 6，单位为字节。
- (2) 设置网卡的 MAC 地址，这里将其指定为 BC-20-06-09-30-11。
- (3) 指定网卡的 MTU 值。
- (4) 允许网卡处理广播通讯。
- (5)
- (6)
- (7)
- (8) 初始化 EMAC。EMACInit() 函数包含查询状态位代码，并且这些查询代码并没有使用 OSTimeDly() 等函数主动释放 CPU 使用权。如果网线在当时并没有接触良好，则这个过程需要相当长的时间。为了避免阻塞其它低优先级任务的正常运行，我们使用了 uCOS 提供的任务管理函数，先将其所在任务的优先级降低，等初始化完成之后再恢复其优先级。EMACInit() 函数笔者将在下一节详解。
- (9) 建立接收线程，该线程的入口地址由 ethernetif_input 指针指定。
- (10) pstNetif 作为参数传递给接收线程。
- (11) 指定接收线程的栈顶地址。其相关定义如下图所示：



```

static const struct eth_addr ethbroadcast = {{0xff, 0xff, 0xff, 0xff, 0xff, 0xff}};

/* 接收线程堆栈 */
#define T_ETHERNETIF_INPUT_STKSIZE      256
OS_STK T_ETHERNETIF_INPUT_STK[T_ETHERNETIF_INPUT_STKSIZE];

/* Forward declarations */
static void ethernetif_input(struct netif *netif);

```

图 5.4.3

- (12) 指定任务优先级，最好将其指定为系统最高优先级。如果确实存在更高优先级的任务，则一定要避免更高优先级的任务长时间占用 CPU 的使用权，否则将严重影响网络性能。

5.4.4 EMACInit() ——网卡初始化工作的实际完成者

在前面的描述中,聪明的读者不难看出,ethernetif.c 文件实际上相当于硬件抽象层(HAL),它提供了硬件访问框架,但并不真正驱动硬件(函数调用与直接驱动虽然结果相同但从实现的角度看却相差甚远)。真正驱动硬件的底层函数我们必须单独实现。为此,笔者又新建了一个文件,专门存放驱动函数(参见 lib_emac.c)。这些驱动函数不仅包括 EMACInit(),还包括 EMACReadPacket() 及 EMACSendPacket(),它们分别负责网络数据包的接收和发送。我们先看看 EMACInit(),其源码如下:

```
void EMACInit(void)
{
    extern HANDLER hEthernetInput;

    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR      cpu_sr = 0;
    #endif

    __ResetPHY();                                     (1)

    OSTimeDlyHMSM(0, 0, 3, 0);                       (2)

    AT91C_BASE_PIOB->PIO_ASR = EMAC_MII_PINS;         (3)
    AT91C_BASE_PIOB->PIO_PDR = EMAC_MII_PINS;         (4)

    AT91C_BASE_PIOB->PIO_PER = AT91C_PB12_ETXER;      (5)
    AT91C_BASE_PIOB->PIO_ODR = AT91C_PB12_ETXER;      (6)

    AT91C_BASE_EMAC->EMAC_NCFGR |= AT91C_EMAC_CLK_HCLK_32; (7)

    __CheckPHYID();                                   (8)

    __SetupLinkSpeedAndDuplex();                      (9)

    hEthernetInput = OSAPISemNew(0);                  (10)

    __InitDescriptorsForRxBAndTx();                   (11)

    AT91C_BASE_EMAC->EMAC_USRIO = AT91C_EMAC_CLKEN;   (12)

    AT91C_BASE_EMAC->EMAC_RSR =
        AT91C_EMAC_OVR | AT91C_EMAC_REC | AT91C_EMAC_BNA; (13)

    AT91C_BASE_EMAC->EMAC_NCFGR |= AT91C_EMAC_CAF
                                   | AT91C_EMAC_NBC
                                   | AT91C_EMAC_DRFCS; (14)
```

```

AT91C_BASE_EMAC->EMAC_SA1L = 0xBC                                (15)
    | ((ULONG) (0x20 << 8))
    | ((ULONG) (0x06 << 16))
    | ((ULONG) (0x09 << 24));
AT91C_BASE_EMAC->EMAC_SA1H =
    0x30 | ((ULONG) (0x11 << 8));                                (16)

OS_ENTER_CRITICAL()                                              (17)
{
    AT91C_BASE_EMAC->EMAC_IER = AT91C_EMAC_RCOMP | AT91C_EMAC_TCOMP; (18)

    at91_irq_open(AT91C_ID_EMAC,                                (19)
        AT91C_AIC_PRIOR_HIGHEST,
        AT91C_AIC_SRCTYPE_INT_HIGH_LEVEL,
        irqEMACISR);

    AT91C_BASE_AIC->AIC_IECR = 0x1 << AT91C_ID_EMAC;            (20)
}
OS_EXIT_CRITICAL()                                              (21)

AT91C_BASE_EMAC->EMAC_NCR |= AT91C_EMAC_TE                      (22)
    | AT91C_EMAC_RE;
}

```

- (1) 复位PHY层芯片。笔者使用的PHY层芯片是台湾Realtek公司生产的RTL8201BL。它是一个单端口的物理层收发器，实现了全部的 10/100M以太网物理层功能。有关它的详细信息读者可参考它的数据手册（中文版下载地址是：http://groups.google.com/group/marsstory/browse_thread/thread/0781145527a06576/ba037e2a7c2a04e7#ba037e2a7c2a04e7，英文版自己搜索），不再赘述。在这里唯一一个需要笔者说明的问题是：为什么要主动复位这个芯片？答案与 8201 的第 24 脚有关。这个引脚在上电复位期间的锁存状态直接决定了 8201 的操作模式：如果逻辑电平为高则 8201 进入光纤模式；为低，则进入UTP模式（这是我们需要的模式）。那么，谁来决定这个引脚的锁存状态呢？是 7x256。因为这个引脚与 7x256 的PI0口线直接相连，这意味着PI0口线在上电时的逻辑电平会被 8201 作为输入进行锁存。因此，要想了解 8201 在上电后到底进入何种操作模式（光纤或者UTP？），就必须清楚 7x256 的PI0口线在上电时的电平状态。查看 7x256 的数据手册后我们得知，7x256 的所有PI0口线都内置了一个上拉电阻，其在上电时的逻辑电平为高。而 8201 虽然在内部已经为该引脚连接了一个弱下拉电阻，但这并不足以保证能够将电平重新拉低，除非 7x256 不再上拉该引脚。因此，我们必须在 7x256 能够正常工作后，使用相关程序代码，禁止对该PI0口线的上拉。这样，8201 内部的弱下拉电阻才会起作用。这时再主动复位 8201，该芯片就能够进入我们想要的UTP模式了。有关__ResetPHY()函数的源码读者可直接查阅LwIPPortingTest_5文件夹下的lib_emac.c文件，这里不再给出。
- (2) 8201 在复位后就可以建立稳定的物理连接链路了。如果这时网线接触不良，那么建立物理链路的时间可能会很长，因此笔者选择了一个比较适中的等待时间。
- (3) 把 MII 接口相关的引脚设置为外设 A (Peripheral A) 的引脚使用（不明白为什么要这样做的读者可参考 7x256 的数据手册，不愿读英文的读者也可查阅 7s64 的数据手册，

atmel 提供了官方发布的中文数据手册，其 PIO 口线部分与 7x256 相同)。其中 EMAC_MII_PINS 是笔者自定义的宏，它定义了与 MII 接口相关的 PIO 口线。详细定义说明请参考 lib_emac.h 文件。

(4) MII 接口引脚由 EMAC 控制，禁止 PIO 控制器控制。

(5)

(6) 8201 并没有提供 ETXER (PB12) 引脚，因此程序主动使能 PIO 控制器对该引脚的控制以避免 EMAC 控制该引脚。为了避免出现不可预知的错误，该引脚还被禁止输出。

AT91C_PB12_ETXER 在 \at91sam7x256\include\AT91SAM7X256.h 中定义。

(7) 网络配置寄存器 EMAC_NCFGR 的 CLK 位用于设置 MDC 时钟分频数，其分频后得到的 MDC 不能超过 2.5MHz (见 7x256 数据手册的 EMAC 部分)。笔者使用的系统 MCK 值为 48MHz，必须 32 分频才能满足这个要求。

AT91C_EMAC_CLK_HCLK_32 在 \at91sam7x256\include\AT91SAM7X256.h 中定义。

(8) 检查物理层芯片的 ID 号。如果 ID 号匹配则证明 8201 工作正常，如果不匹配该函数将一直阻塞当前线程直至匹配。之所以一直阻塞当前线程，主要是因为 8201 一旦不能正常工作，网络系统将完全瘫痪，这是一个致命故障，必须解决后才可继续网络系统的初始化工作。__CheckPHYID() 函数使用了 PHY 维护寄存器 (EMAC_MAN) 与 8201 进行通讯读取它的 ID 号。7X256 与 8201 进行通讯的操作相对比较简单，读者可根据笔者提供的相关源码结合 8201 的数据手册自行研究，笔者因篇幅所限不再赘述，如有不明白的问题欢迎 MSN 交流：marsstory99@hotmail.com，笔者白天正常工作时间会一直在线。

特别提示：网络状态寄存器 (EMAC_NSR) 的 IDLE 位为 0 表明 EMAC 还未完成对 PHY 芯片的操作，为 1 才表明结束操作。这一点 7x256 的数据手册正好相反。

(9) 从 8201 获取自动协商的结果，设置 EMAC 自身的链路速度和单双工方式。注意，该函数会阻塞所在任务的正常执行直至设置成功。

(10) 建立接收线程使用的信号量，该信号将由 EMAC 接收中断发送给接收线程。有关 EMAC 接收中断的详细信息笔者将在后面讲解。在这里需要特别交待的是 hEthernetInput 是一个全局变量，ethernetif.c 文件中的接收线程也要使用，读者在声明这个变量时要注意这一点。

(11) 初始化接收和发送缓冲区描述符队列，使其指向正确的内存区。在讲解 __InitDescriptorsForRxAndTx() 函数之前，先让我们从应用的角度了解一下 EMAC 的基本运作机制。

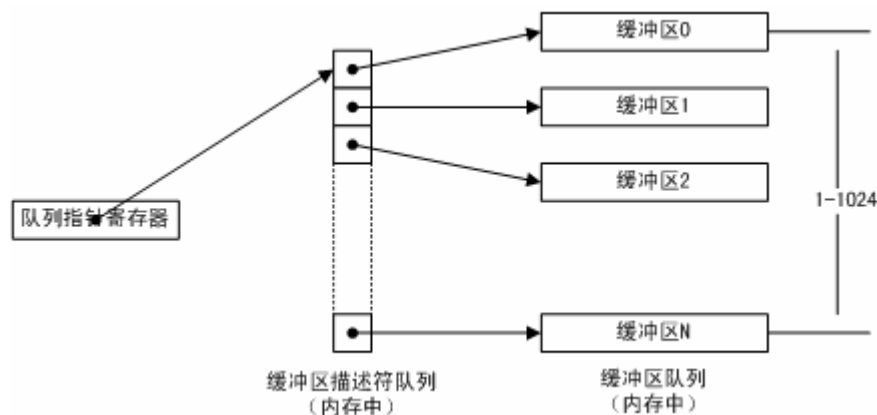


图 5.4.4

对于接收，当一帧数据到达时，EMAC 会立即读取接收缓冲区队列指针寄存器 (EMAC_RBQP) 得到当前缓冲区描述符的地址。而缓冲区描述符保存着与之相对的接收缓冲区的地址 (参见图 5.4.4)。于是，通过两次读取操作，EMAC 得到了真正的接收地

址。接着，EMAC 的接收 DMA 开始工作，把收到的数据写入接收缓冲区。EMAC 规定接收缓冲区的大小为 128 字节，当帧长超过 128 个字节时，EMAC 在写完当前缓冲区后会接着读取下一个缓冲区的地址，写入剩余数据。EMAC 会重复此工作直至写完一整帧。最后，EMAC 会立即更新接收缓冲区描述符的相关状态位，标记这块区域为程序所有。在数据被读取之前，EMAC 不再向其写入任何数据。EMAC_RBQP 寄存器的值，会随着数据帧的不断到达自动更新，其指向的描述符会沿着队列方向顺序下移，直至遇到结束位（wrap bit）置位的描述符或者第 1024 个描述符。这时 EMAC_RBQP 重新指向描述符队列的首部，开始新的循环。对于发送，其同样使用了指针寄存器（EMAC_TBQP）保存当前未使用的缓冲区，基本结构和运作流程与接收基本相似，这里不再赘述。这样，EMAC 的整体运作轮廓读者可以基本掌握。那么，从编程实现的角度看，决定 EMAC 正常工作的关键因素是什么呢？从笔者本人的实践看，有两点：其一、指针寄存器，其初始值是否设置正确直接影响 EMAC 能否工作；其二、缓冲区描述符，它提供了软件与硬件的通讯通道。它不仅为硬件（也就是 EMAC）提供了 DMA 的读取和写入地址（分别针对发送和接收），还为软件提供了帧数据的发送和接收结果。翻阅 7x256 的数据手册我们得知，无论是接收还是发送，其缓冲区描述符均很复杂，我们需要一定的编程技巧方可简化我们的工作，特别是在实现 EMAC 接收和发送函数时，这一点显得尤为重要。笔者采用了位字段的方法定义缓冲区描述符，这样作的好处是可以像访问结构体成员一样访问描述符中单独的位，编程简单，不容易出错。有关接收和发送缓冲区描述符的详细定义参见 lib_emac.h 文件，这里不再给出。

下面，笔者将结合 __InitDescriptorsForRxBAndTxB() 函数的源码，详解缓冲区及指针寄存器的初始化过程，源码如下：

```
static void __InitDescriptorsForRxBAndTxB(void)
{
    LONG          i;

    for(i=0; i<NB_RX_BUFS; i++)
        __staRxBDescriptors[i].ulRxBAddrAndFlag = (int)baRxBuFs[i]; (11.1)
    __staRxBDescriptors[NB_RX_BUFS-1].ulRxBAddrAndFlag
        |= RxDESC_FLAG_WRAP; (11.2)

    for(i=0; i<NB_TX_BUFS; i++)
    {
        __staTxBDescriptors[i].ulTxBAddr = (int)baTxBuFs[i]; (11.3)
        __staTxBDescriptors[i].uStatus.bstStatus.bitIsUsed = 1; (11.4)
    }
    __staTxBDescriptors[NB_TX_BUFS-1].uStatus.bstStatus.bitIsWrap
        = 1; (11.5)

    AT91C_BASE_EMAC->EMAC_RBQP = (int)__staRxBDescriptors; (11.6)
    AT91C_BASE_EMAC->EMAC_TBQP = (int)__staTxBDescriptors; (11.7)
}
```

(11.1) 队列中的所有描述符逐一指向各自的接收缓冲区，实现一对一绑定。NB_RX_BUFS 为接收缓冲区的个数，具体指定多少个没有多少规矩可依，用户可根据需要指定，不过最好不要小于 12 个，因为 12*128 正好是 1536。以太网的最大帧长是 1500，正好可以接收一个完整帧。笔者的实现中指定的缓冲区个数为 32 个。__staRxBDescriptor 是笔者定义的接收缓冲区描述符数组，其定义如下：

```
__align(8) static volatile AT91S_RxBDescriptor
__staRxBDescriptors[NB_RX_BUFS];
```

其中__align 是 ARM C/C++编译器提供的一个类型限定词，用于指定相关变量的字节对齐方式。在我们的实现中，缓冲区描述符需要占用两个整形数据的存储空间，也就是 8 个字节，所以，这里指定 8 字节对齐。因为__staRxBDescriptors 只在 lib_ether.c 中使用，不需要外部链接特性，因此，笔者使用了 static 限定词，显式的告知编译器该变量的内部链接特性。在前文的描述中我们知道，缓冲区描述符不仅会被软件改变，还会被硬件改变，因此必须指定 volatile 限定词以避免编译器优化。AT91S_RxBDescriptor 就是我们前文提到的接收缓冲区描述符的 C 实现。它被实现为一个稍微复杂一些的 C 结构。其中 ulRxBAddrAndFlag 字段就是接收缓冲区描述符的字 0（见 7x256 数据手册的 EMAC 部分），它保存了接收缓冲区的地址。每一个描述符绑定一个缓冲区。笔者为了省事，缓冲区以数组的形式实现（数组是一个连续的缓冲区，而图 5.4.4 所示的是一个分散的缓冲区），其定义如下：

```
__align(4) static volatile BYTE baRxBufs[NB_RX_BUFS][ETH_RX_BUF_SIZE];
```

根据数据手册我们得知，缓冲区描述符字 0 的最低两位为标志位，31:2 位保存缓冲区的开始地址，因此显式的指定 4 字节对齐，以保证缓冲区地址正好占用 31:2 位，最低两位保留。其它限定词与描述符定义相似，不再赘述。ETH_RX_BUF_SIZE 指定接收缓冲区的大小，根据数据手册，接收缓冲区的大小固定为 128 字节。

- (11.2) 置位队列中最后一个描述符的 Wrap 位，为描述符队列加上尾部标记。
- (11.3) 为发送缓冲区描述符指定发送缓冲区地址。NB_TX_BUFS 为发送缓冲区的个数，其个数多少与缓冲区大小相关，原则是缓冲区队列的总容量不能小于 1500。在笔者的实现中，NB_TX_BUFS 的值为 16，发送缓冲区的大小是 256，由 ETH_TX_BUF_SIZE 指定。__staTxDescriptors 的定义与__staRxBDescriptors 相同，这里不再给出。

baTxBufs 的定义如下：

```
static BYTE baTxBufs[NB_TX_BUFS][ETH_TX_BUF_SIZE];
```

相对接收缓冲区，其描述符的字 0 完全用于缓冲区地址，不需要保留个别位，因此其定义没有字节对齐的限定。

- (11.4) 标记这个缓冲区为程序所有。EMACSendPacket() 函数会使用这个标记作为是否向当前缓冲区写入数据的依据。如果置 1，表明当前缓冲区已被 EMAC 释放（即数据已经被发送出去），可以写入下一帧数据了。置 0，则正好相反。在 EMAC 初始化时，发送缓冲区肯定没有任何需要发送的数据，因此，所有的描述符该位置 1，以允许写入。
- (11.5) 与 (11.2) 相同。
- (11.6)
- (11.7) EMAC 接收和发送指针寄存器指向正确的地址。
- (12) 使能 MII 操作模式，使能收发器输入时钟。
- (13) 为避免意外情况的发生，先清一次接收状态寄存器的相关状态位。
- (14) 允许接收所有有效帧，不接收广播帧，抛弃 FCS 字段。
- (15)
- (16) 指定 EMAC 地址，还是 BC-20-06-09-30-11（见 5.4.3 节）。这里需要特别提醒读者的是，这一高一低两个特殊地址寄存器其写入顺序是不能随便更改的，必须先写入低位再写入高位。因为 7x256 的数据手册对此有明确交待：复位或者写低位寄存器，地址被取消，写高位寄存器，地址被启用。

(17)
(18)
(19)
(20)

(21) 这几行代码完成 EMAC 中断的初始设置工作。为了保证代码执行的连续性，笔者对其进行了临界保护。有关中断初始设置的技术细节不在本文的描述范围之内，不明白的读者可查阅相关的数据手册。在这几行代码中又出现了一个新的函数 `irqEMACISR()`。它是 EMAC 的中断处理函数，负责发送结束或接收结束之后的善后处理工作。其源码如下：

```
__irq void irqEMACISR(void) (19.1)
{
    extern HANDLER hEthernetInput;
    ULONG    __ulIntStatus, __ulReceiveStatus;

    OSIntEnter(); (19.2)
    {
        __ulIntStatus = AT91C_BASE_EMAC->EMAC_ISR; (19.3)
        __ulReceiveStatus = AT91C_BASE_EMAC->EMAC_RSR; (19.4)

        if((__ulIntStatus & AT91C_EMAC_RCOMP)
            || (__ulReceiveStatus & AT91C_EMAC_REC)) (19.5)
        {
            OSAPISemSend(hEthernetInput); (19.6)

            AT91C_BASE_EMAC->EMAC_RSR = AT91C_EMAC_REC; (19.7)
        }

        if(__ulIntStatus & AT91C_EMAC_TCOMP) (19.8)
        {
            __ilResetTxBDescriptors(); (19.9)
            AT91C_BASE_EMAC->EMAC_TSR = AT91C_EMAC_COMP; (19.10)
        }

        AT91C_BASE_AIC->AIC_EOICR = 0; (19.11)
    }
    OSIntExit(); (19.12)
}
```

(19.1) `__irq` 是 ARM C/C++编译器提供的针对函数的限定词，它用于显式的告知编译器这是一个中断服务子函数（ISR），在编译时必须确保所有 ISR 用过的寄存器在中断退出后能够被正确恢复，否则，被中断的程序将会出现不可预知的错误。

(19.2) 我们的系统建立在 uCOS 的基础上。根据 uCOS 的数据手册，uCOS 需要知道正在做中断服务，因此必须最先调用 `OSIntEnter()`，通知 uCOS 进入中断服务。关于 `OSIntEnter()` 函数的详细信息，参见 uCOS 的数据手册。

(19.3) 读取 EMAC 中断状态寄存器（EMAC_ISR），获取当前中断原因。注意，读取该寄存器会清除当前状态位，必须申请变量进行保存。

(19.4) 这是最容易忽略的地方，必须读取 RSR 寄存器，才可在写 AIC_EOICR 寄存器之后结束 EMAC 中断服务。

(19.5)

(19.6)

(19.7) 接收结束中断，立即向接收线程（即 `ethernetif_input()` 函数）发送信号（`hEthernetInput`），通知其把接收缓冲区中的数据取出。最后写 1 清除 `EMAC_RSR` 寄存器的 `REC` 位，以便为下一次中断提供正确的判断依据。

(19.8)

(19.9)

(19.10) 发送结束中断，重新标记发送缓冲区为程序所有，然后写 1 清除 `EMAC_TSR` 寄存器的 `COMP` 位。`__ilResetTxDescriptors()` 是一个内联函数（函数源码见 `lib_emac.c`），它负责回收 `EMAC` 占用的发送缓冲区。这个函数的关键是 `__lldxToReset` 变量，它在函数之外声明，具备 `static` 特性，所以它是一个私有静态变量。函数执行前，它保存着 `EMAC` 占用的缓冲区的开始位置。程序会从这个位置开始回收 `EMAC` 占用的缓冲区，随着 `__lldxToReset` 变量的顺序移动，最终收回全部缓冲区。这时 `__lldxToReset` 变量再一次与 `EMAC_TBQP` 寄存器指向的位置重叠，而这恰恰是下一次发送的开始位置。就这样，无论发送多少次数据，`__lldxToReset` 变量始终与 `EMAC_TBQP` 寄存器保持着位置同步，见下图：

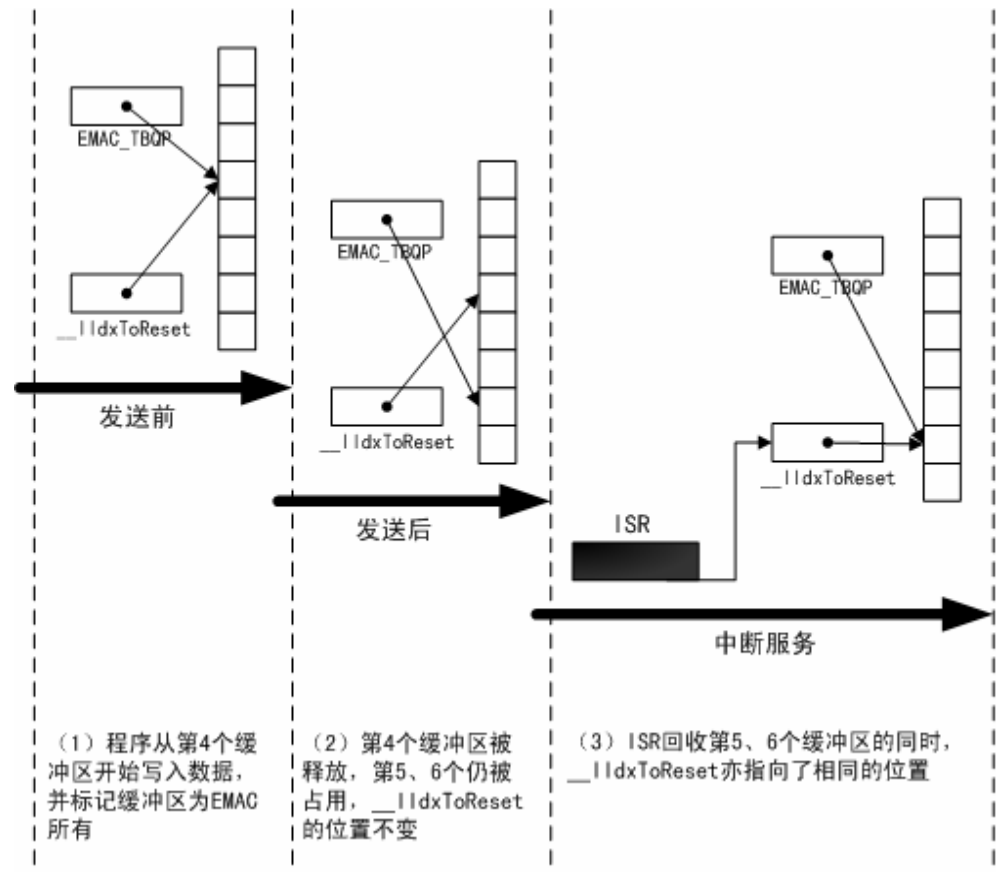


图 5.4.5

(19.11) 写 `AIC_EOICR` 寄存器，结束中断服务。

(19.12) 通知 `uCOS` 中断服务结束。关于 `OSIntExit()` 函数的详细信息，参见 `uCOS` 的数据手册（推荐邵贝贝翻译的《嵌入式实时操作系统 uC/OS-II》，北京航空航天大学出版社）。

无论是发送还是接收，`irqEMACISR()` 函数都是整个流程的必经之路。图 5.4.6 很直观的表现了这一点。对于发送，ISR 负责扫尾，虽是扫尾，却很关键，一旦出错，发送将被迫停止。对于接收，从软件实现的角度看，ISR 是整个流程的发起者，其重要性

可见一斑，笔者就不多说了。

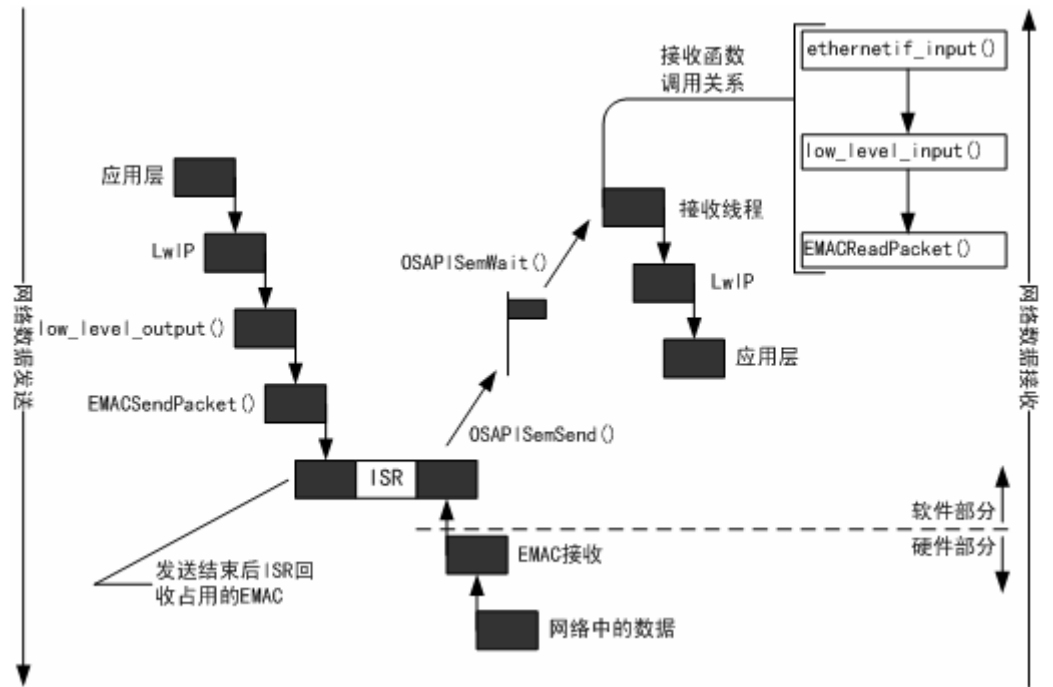


图 5.4.6

(22) 最后，使能接收、发送。

5.4.5 ethernetif_input() —— 实现接收线程

该函数以标准的 uCOS 任务的形式实现，完成网络数据的读取与传递工作。它不断的读取 EMAC 的缓冲区，一旦读取到数据就立即进行处理。它首先取得以太网帧头，然后根据以太网帧头携带的上层协议类型值判断要把数据传递给哪一个协议处理模块，传递完毕，流程重新开始。目前该函数仅支持两种协议类型——IP 和 ARP，其它类型不做处理，直接丢弃。该函数的源码如下：

```
static void ethernetif_input(void *pReserved)
{
    struct ethernetif    *__pstEthernetif;
    struct pbuf          *__pstPbuf;
    struct eth_hdr       *__pstEthhdr;
    struct netif         *__pstNetif;

    __pstNetif = (struct netif*)pReserved;
    while(TRUE)
    {
        __pstEthernetif = (struct ethernetif *)__pstNetif->state;
        do{
            __pstPbuf = low_level_input(__pstNetif);
            if(__pstPbuf == NULL)
                OSAPISemWait(hEthernetInput, 100);
        }while(__pstPbuf == NULL);
    }
```

```
__pstEthhdr = __pstPbuf->payload; (6)

switch(htonl(__pstEthhdr->type)) (7)
{
    case ETHTYPE_IP: (8)
        etharp_ip_input(__pstNetif, __pstPbuf); (9)
        pbuf_header(__pstPbuf, -sizeof(struct eth_hdr)); (10)
        __pstNetif->input(__pstPbuf, __pstNetif); (11)

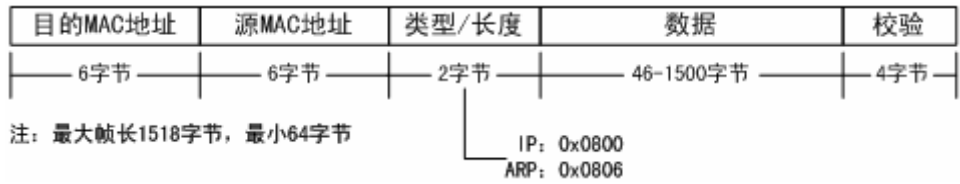
        break;

    case ETHTYPE_ARP: (12)
        etharp_arp_input(__pstNetif, (13)
                        __pstEthernetif->ethaddr,
                        __pstPbuf);

        break;

    default:
        pbuf_free(__pstPbuf); (14)
        __pstPbuf = NULL; (15)
        break;
}
}
```

- (1) 保存任务建立时传递的参数值，这个参数其实就是 5. 3 节提到的系统默认的网络接口 __stEMACNetif。
- (2) 这一句存在的目的是为了获得当前网卡的 MAC 地址，以便 ARP 协议处理模块使用。
- (3)
- (4)
- (5) 不断读取 EMAC 的接收缓冲区。low_level_input() 函数就是实现读取的函数，该函数将在 5. 4. 6 节讲解。这几行代码中值得关注的是 OSAPISemWait() 函数，它最长等待 100 毫秒，如果等不到指定的信号，其仍然会返回。程序会再一次主动查询 EMAC 的接收缓冲区，这样可以保证及时读取网络数据，而不是非要等到指定信号。
- (6) 获取以太网帧头。
- (7) 根据以太网帧头携带的上层协议类型值传递数据。有关以太网的详细信息，读者可查阅相关资料，这里仅给出以太网帧格式定义：



- (8)
- (9)
- (10)
- (11) 这几行代码完成数据向 IP 层传递的工作。首先更新 ARP 表，然后跳过以太网帧头，最后传递给 IP 协议处理模块，在这里实际上就是把收到的数据交给 `tcpip_input()` 函数。这个函数读者肯定不会陌生，这里不再赘述。
- (12)
- (13) 响应 ARP 查询请求，并更新自身的 ARP 表。
- (14)
- (15) 释放占用的 pbuf。

5.4.6 `low_level_input()` ——得到一整帧数据

这个函数对于接收来说非常关键，它是底层硬件与上层协议栈的连接枢纽。它负责向协议栈申请接收缓冲区——pbuf，然后调用 `EMACReadPacket()` 函数把 EMAC 收到的数据搬运到 pbuf 中，完成 EMAC 到协议栈的数据转移工作。在这个函数中，我们向协议栈申请的 pbuf 类型是 `PBUF_POOL`。因为从 pbufs pool 中分配一个 pbuf 的操作可以快速完成，非常适合底层驱动。pbufs pool 是由固定大小的 pbuf 组成，每个 pbuf 的长度由协议栈提供的 `PBUF_POOL_BUFSIZE` 宏指定（见下图）。

当一帧数据到达 EMAC 的接收缓冲区后，我们必须通过 `GetInputPacketLen()` 函数得到帧长，也就是 `__uwLen` 的值（见 `ethernetif.c` 文件中的 `low_level_input()` 函数源码）。然后调用协议栈提供的 pbuf 管理函数 `pbuf_alloc()` 从 pool 中分配足够数量的 pbufs 用于保存到达的帧。那么，怎么才算是足够数量呢？图 5.4.7 为此提供了一个算法，向读者展示了这个数量到底应该是多少。这个算法很简单，笔者就不多作解释了。`pbuf_alloc()` 函数就是使用与之类似的算法分配 pbuf 的。在图 5.4.7 中，最初的 pbufs pool 中的 pbuf 数量为 9。在我们调用 `pbuf_alloc()` 函数之后，pbuf 的数量变为 6。这说明我们需要 3 个 pbuf 保存这一帧数据。这 3 个 pbuf 被组织为一个单向链表，以方便程序处理。当 pbuf 申请成功，我们就可以立即调用 `EMACReadPacket()` 函数开始从 EMAC 接收缓冲区搬运数据到这个 pbuf 链了。如果 `PBUF_POOL_BUFSIZE` 的值为 256，那么一个 pbuf 正好可以保存两个 EMAC 接收缓冲区的的数据，这样的话 `EMACReadPacket()` 函数的数据搬运处理将变得简单，程序执行性能得到提升。因此，在笔者的实现中，`PBUF_POOL_BUFSIZE` 的值就是 256（参见 `opt.h`）。当然，不能只看问题的表面，进一步抽象说，只要是 pbuf 的长度正好是接收缓冲区的整数倍，其效果与两倍没什么不同，读者可自行酌定。

这个函数从源码本身看非常简单，流程也很清晰，笔者在这里就不再逐行讲解了。唯一需要读者注意的地方就是其调用的三个函数的关系，特别是 `GetInputPacketLen()` 与 `EMACReadPacket()` 函数，这两个函数之间看似没有直接联系，但实际上 `EMACReadPacket()` 函数完全依赖 `GetInputPacketLen()` 函数的执行结果。接下来的两节将详解这两个函数，读者在阅读时一定要注意这一点。

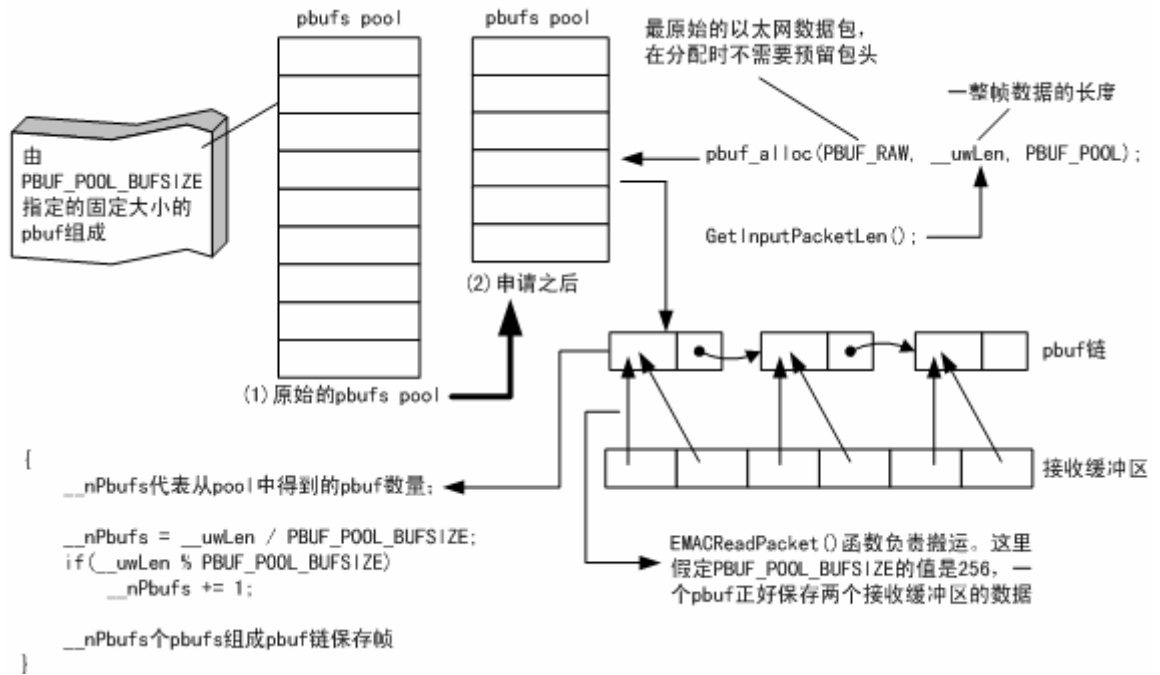


图 5.4.7

5.4.7 GetInputPacketLen() —— 获得帧长

这个函数让我们又重新回到了 lib_eamc.c 文件。它负责扫描整个 EMAC 接收缓冲区，查找完整的数据帧，获得帧长。下面，笔者将结合源码讲解这个函数的工作流程，源码如下：

```

UWORD GetInputPacketLen(void)
{
    UWORD    __uIdx, __uLen = 0;

    while(
        (__staRxBDescriptors[__uCurRxBIdx].uIRxBAddrAndFlag
        & RxDESC_FLAG_OWNERSHIP) &&
        !__staRxBDescriptors[__uCurRxBIdx].uStatus.bstStatus.bitStartOfFrm
    )
    {
        __staRxBDescriptors[__uCurRxBIdx].uIRxBAddrAndFlag &=
            (~RxDESC_FLAG_OWNERSHIP);
        __uCurRxBIdx++;
        if (__uCurRxBIdx >= NB_RX_BUFS )
        {
            __uCurRxBIdx = 0;
        }
    }

    __uIdx = __uCurRxBIdx;

    while((__staRxBDescriptors[__uIdx].uIRxBAddrAndFlag

```

```

        & RxDESC_FLAG_OWNESHIP))
    {
        __uwLen = __staRxBDescriptors[__uwIdx].uStatus.bstStatus.bitLen; (7)
        if(__uwLen > 0) (8)
            break;

        __uwIdx++; (9)
        if(__uwIdx >= NB_RX_BUFS) (10)
            __uwIdx = 0;

    }

    __pbFrom = (BYTE*) (11)
        (__staRxBDescriptors[__uwCurRxBIdx].uIRxBAddrAndFlag
            & EMAC_RxB_ADDR_MASK);

    return __uwLen; (12)
}

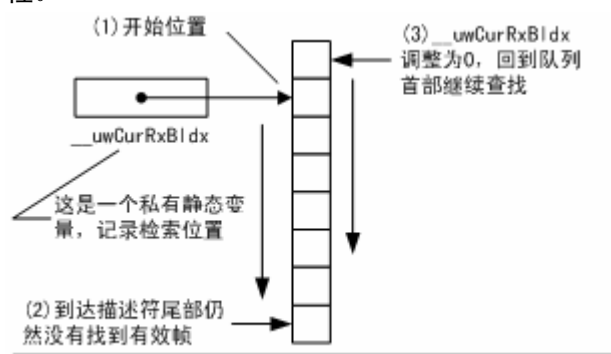
```

(1)

(2)

(3)

- (4) 这几行代码以__uwCurRxBIdx 变量作为索引，检索整个接收缓冲区描述符队列，以确定帧的开始位置，检索条件是判断缓冲区描述符的 Ownship 及 start-of-frame 位是否置位。这是一个环型检索，程序在到达队列尾部时如果仍然无法满足条件，则__uwCurRxBIdx 变量会被调整为 0，程序重新从队列的首部继续检索，图 5.4.7 很直观的表现了这一点。其中__uwCurRxBIdx 变量在函数外部定义，具备 static 特性，是一个静态私有变量。之所以在函数外部定义，原因很简单，__uwCurRxBIdx 变量在 EMACReadPacket() 函数中同样被作为进入缓冲区队列的索引使用，而且其必须在 GetInputPacketLen() 函数执行后使用，所以在函数外部定义使其具备文件作用域特性。



注：描述符队列的检索流程图示

图 5.4.8

在结束这几行代码的讲解之前还有两个问题需要向读者说明：第一个问题，为什么会出现缺少帧首（start-of-frame bit）的帧呢？第二个问题，如果当前缓冲区没有写入任何数据，也就是 Ownship 位为 0，那么，检索照样会结束，帧的开始位置并没有被确定，因此，我们的判断条件是错误的，可为什么还这样写？

先说第一个问题。缺少帧首的帧实际上就是数据碎片，它并不是一个完整的帧。所以我们的程序没有作任何处理就将其占用的缓冲区收回。根据 7x256 的数据手册，再加

上笔者的理解，出现数据碎片的原因可能是因为这样（数据手册对此讲得也不是很清楚，笔者只能根据数据手册的上下文猜测其中的原因，可能笔者的理解不是很正确，欢迎批评指正）：如果接收到一个超长帧，缓冲区无法全部放下，那么，数据碎片就出现了。如果某一个帧大于 128 个字节并且存在错误，那么根据数据手册，这个帧只能占用两个以上的缓冲区，而 EMAC 在发现帧存在错误后只恢复当前使用的缓冲区，所以先前的缓冲区就成为垃圾留在接收队列中了，这种情况也出现了数据碎片。不过，一个正常工作的以太网，不应该出现太长的帧或者大于 128 字节的有 CRC/FCS 错误的帧，因此，在接收队列中发现数据碎片的情况还是很罕见的。虽然这种情况很罕见，但我们的程序却不能忽略，因此在我们的实现中并没有假设一帧一帧的数据紧密排列的情况。另外需要读者特别注意的是，只要存在帧首，就预示着已经找到一个完整帧，在当前或后面的缓冲区中肯定能够找到帧尾得到帧长。

对于第二个问题，这样写到底对不对，关键是看会造成什么样的结果。这个函数存在两个循环，第二个循环根据第一个循环的结果进行检索。当第一个循环没有确定帧首位置时，这实际意味着在此之前的时间内，EMAC 没有收到任何完整的帧。在程序进入第二个循环时会出现两种情况：没有收到有效帧，帧长为 0；收到有效帧，帧长大于 0。对于第一种情况，函数会顺利返回调用者，接收线程继续等待数据的到来，这是我们想要的结果。对于第二种情况，会产生两种结果：结果一，从当前缓冲区（即 `__uwCurRxBIdx` 变量指向的缓冲区）开始得到了一个完整帧，那么 `EMACReadPacket()` 函数就可以读取这一帧然后传递给接收线程（`EMACReadPacket()` 函数从 `__uwCurRxBIdx` 变量指向的位置开始读取帧），这也是我们想要的结果；结果二，从当前缓冲区开始，至少一个以上的缓冲区存在数据碎片，`EMACReadPacket()` 函数会把这些碎片加上有效帧一同传递给接收线程，因为这些无效数据的存在，接收线程会丢弃这些数据，包括有效帧，这显然不是我们想要的结果。因此，从结果二看，我们这样写是不对的。但是，如果我们回过头来再看看前面忽略的一个问题，就能明白笔者为什么还要这样写的原因了。在进入第二个循环之前，确切的说完成第二个循环的第一次条件判断之前，存在着一个很短暂的纳秒级的时间空隙（就是几条指令而已）。这个短暂的时间空隙使得 EMAC 很难完成接收。因此，绝大多数情况下当前缓冲区不会写入任何数据，也就是 `Ownship` 位为 0，进入第二个循环的条件不成立，帧长为 0。只有一种情况 `Ownship` 位为 1，那就是在这个短暂的时间空隙内 EMAC 正好完成接收。很明显的一个事实是，EMAC 的接收时间远远大于这个时间空隙。因此，这个短暂的时间空隙内只能完成一次接收，要么接收的是数据碎片，帧长仍然为 0；要么就是有效帧，帧长大于 0，肯定不会出现数据碎片与有效帧共存的情况，所以笔者这样写是对的。写到这里，不知读者是否还记得在本文的第 39 页，笔者曾经明确说明接收线程的优先级最好设定为系统最高优先级，现在读者应该明白为什么会这样说了吧。没错，如果它不是最高优先级，两个循环之间的时间空隙将有可能变得很长，那么，数据碎片与有效帧共存的情况将会出现，丢包现象就会发生，因此，接收线程最好是最高优先级，否则，只能修改 `GetInputPacketLen()` 函数的算法了。

- (5) 设置第二次循环的索引初始值。之所以不再继续使用 `__uwCurRxBIdx` 变量，原因是这个变量保存着帧首位置，`EMACReadPacket()` 函数需要使用这个变量及其当前值，所以必须保留。
- (6)
- (7)
- (8)
- (9)
- (10) 正如前文所述，这几行代码的任务就是找到帧尾得到帧长。根据数据手册，只有最后一个缓冲区才保存帧尾，在这之前的全为 0，请读者注意这一点。

- (10) 调整 `__pbFrom` 的指向，使其指向帧的第一个接收缓冲区。`__pbFrom` 是一个私有静态变量，函数外部定义，具备文件作用域特性。它是为 `EMACReadPacket()` 函数准备的。
- (11) 返回帧长。

5.4.8 EMACReadPacket() —— 复制，从接收缓冲区到 pbuf

在 5.4.6 节，笔者已经对这个函数要完成的工作作了一些交待，特别是有关如何让程序处理更简单的问题，笔者作了重点说明。按照前面的思路，程序假设 pbuf 一定大于或等于接收缓冲区，小于的情况肯定不会出现，因此，读者一定要保证 `PBUF_POOL_BUFSIZE` 的值大于或等于 128，否则不要采用笔者的算法。在这个算法中，`PBUF_POOL_BUFSIZE` 的值大于或等于 128 还存在两种情况，一种是 `PBUF_POOL_BUFSIZE` 的值是 128 的整数倍，另一种是非整数倍。这两种情况的处理过程前者简单后者复杂，笔者在接下来的篇幅中将分别给出这两种情况的一般性算法实现，读者可选择实现其中的一种。

先谈谈第一种情况，这种情况可以用下图来表示：

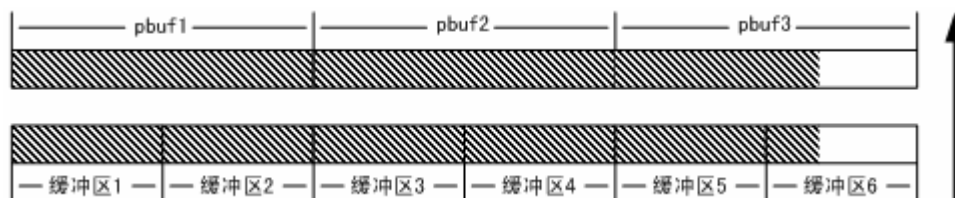


图 5.4.9

上图中，单个 pbuf 的长度是 EMAC 接收缓冲区的两倍（即 `PBUF_POOL_BUFSIZE` 的值为 256），图中阴影部分代表缓冲区中保存的有效数据，算法如下：

```
void EMACReadPacket(BYTE *pbTo, UWORD uwSegmentLen, BOOLEAN bIsLastPbuf)
{
```

`__nRxBufs` 表示需要搬运的接收缓冲区的数量：

`uwSegmentLen` 是 pbuf 链中单个 pbuf 需要接收的数据长度，这个变量的值由 `low_level_input()` 函数提供，它实际上小于或等于 pbuf 的实际长度，也就是 `uwSegmentLen <= PBUF_POOL_BUFSIZE`。确切的说在 pbuf 链中，只有最后一个 pbuf 才有可能小于 `PBUF_POOL_BUFSIZE`，其余的肯定等于 `PBUF_POOL_BUFSIZE`，如上图所示；

```
__nRxBufs = uwSegmentLen / 128;
```

按照上图所示的例子，`__nRxBufs` 的值应该是 2，或者 1。先把整数倍的数据搬运到 pbuf 中，代码如下：

```
for (l=0; l<__nRxBufs; l++)
{
```

小注：`__pbFrom` 及 `__uwCurRxBIdx` 的初始值由 `GetInputPacketLen()` 函数设置，详见 5.4.7 节；
`memcpy(pbTo + l * 128, __pbFrom, 128);`

```
__uwCurRxBIdx++;
```

```
.....;
```

```
__pbFrom 指向下一个接收缓冲区，即 __staRxBDescriptors[__uwCurRxBIdx];
```

```
        把接收缓冲区归还给 EMAC;
    }
    if(uwSegmentLen % 128)
    {
        不能整除, 表明这已经到了 pbuf 链的末尾, 上图中 pbuf 链的末尾是 pbuf3;

        把剩下的数据搬运到 pbuf。在上图中, 剩余数据实际上就是缓冲区 6 的阴影部分,
        缓冲区 5 的数据已经在 for 循环中完成搬运, 实际的 C 代码如下:
        memcpy(pbTo + __nRxBufs*128, __pbFrom, uwSegmentLen - __nRxBufs*128);

        .....;

        把接收缓冲区归还给 EMAC, 上图中则是把缓冲区 6 归还给 EMAC;
    }

    返回 low_level_input() 函数;
}
```

我想读者从上面的算法中应该能够感觉到, 数据搬运的处理过程还是比较简单的, 实现起来并不困难, 因此在笔者提供的源码中并没有采用这个算法, 而是采用了非整数倍的实现算法。非整数倍的情况如下图所示, 其处理过程相对比较复杂, 必须采取一定的编程技巧:

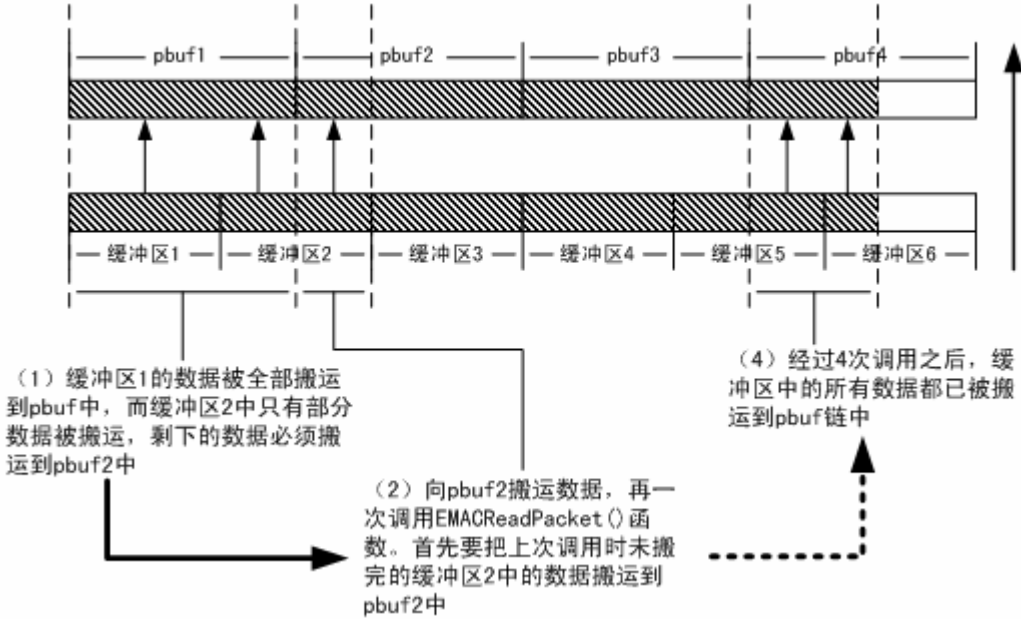


图 5.4.10

非整数倍的情况在实现时比较关键的地方就是接收缓冲区中剩余数据的处理, 如何准确获得缓冲区中剩余数据的位置及长度是整个算法的核心所在。就像图 5.4.10 所示的那样, 必须多次调用 EMACReadPacket() 函数才能完成一整帧数据的搬运工作, 每次调用只能搬运一个 pbuf 的数据。因此, 上图缓冲区 2 中的剩余数据的位置及长度必须进行保存, 并且在 low_level_input() 函数执行期间有效。要达到这个目的, 我们必须定义一个静态私有变量, 这样才能保证数据搬运的连续性。在笔者给出的源码中 (参见 lib_emac.c), __uwTotalLenInRxBToRead 变量就是这样的一个变量, 它具备 static 特性, 负责记录 EMAC 接收缓冲区中的已读数据长度, 根据这个值, 我们可以轻而易举的获得剩余数据的位置和长度。下面, 笔者给出非整数倍情况下的实现算法描述, C 源码参见 lib_emac.c, 读者可结合该算法描述阅读理解具体的实现源码:

```
void EMACReadPacket(BYTE *pbTo, UWORD uwSegmentLen, BOOLEAN bIsLastPbuf)
{
```

```
    while(是否已经搬运完毕)
```

```
    {
```

```
        if(pbuf 中还能够保存的数据长度 >= 接收缓冲区中未搬运的数据)
```

```
        {
```

在这个条件下，当前接收缓冲区肯定能够结束搬运，无论是这个缓冲区是否已经搬运过数据，换句话说，pbuf 至少还能够完全接收缓冲区中剩余的数据。这样直接导致__pbFrom 指向下一个接收缓冲区，即

```
__staRxBDescriptors[__uwCurRxBIdx];
```

最后将当前接收缓冲区归还给 EMAC;

```
    }
```

```
    else
```

```
    {
```

只有一种情况才能进入这个分支，那就是 pbuf 只能接收缓冲区中的一部分数据。有两种可能会出现这种情况，一种是像图 5.4.10 所示的那样，pbuf1 只能接收缓冲区 2 中的一部分数据，剩余数据在下次调用时搬运给 pbuf2；另外一种是在到达 pbuf 链的末尾，并且最后一个接收缓冲区中的有效数据长度小于 128（参见图 5.4.10 中的缓冲区 6），这时的 pbuf 还能够接收的数据长度等于接收缓冲区中的有效数据长度，而 if 判断条件中：

接收缓冲区中未搬运的数据 = 128 - __uwTotalLenInRxBToRead;

这时的__uwTotalLenInRxBToRead 肯定等于 0（至于为什么，读者可参看该函数的实现源码），因此**接收缓冲区中未搬运的数据**等于 128，if 条件不再满足。

读者明白了什么条件下才能进入这个分支，那么如何编写搬运代码就变成很简单的事情了，这里不再给出。

```
    }
```

```
}
```

在函数的最后，如果已经到达 pbuf 链的末尾，即 bIsLastPbuf 为 TRUE，那么还有一项工作可能需要做，那就是归还最后一个接收缓冲区。这里为什么说是可能呢，原因是只有帧长并不是 128 的整数倍时才需要归还，如果是 128 的整数倍，那么我们在上面的 if 分支中已经完成了这项工作，不明白的读者可参见该函数的源码，这里不再赘述。

```
}
```

5.4.9 EMACSendPacket() —— 发送一帧数据

这个函数与读取函数正好相反，它负责把数据从 pbuf 搬运到 EMAC 发送缓冲区中。与 EMACReadPacket() 函数相似，这个函数同样需要一个上层调用函数配合使用，它的上层调用函数就是我们在 5.4.2 节介绍过的 low_level_output()。两者之间的关系与读取函数组之间的关系相似，除了数据搬运方向之外。

low_level_output() 函数把要发送的数据分割保存在多个 pbuf 中。这些 pbufs 通过 pbuf->next 指针顺序链接在一起，形成一个 pbuf 链。low_level_output() 函数使用 for 循环

(参见 5.4.2 节) 把这些 pbuf 中的数据通过 pbFrom 参数传递给 EMACSendPacket() 函数, 每次传递一个 pbuf, 多次循环之后, 一整帧数据就被完整的搬运到 EMAC 发送缓冲区中。当 EMACSendPacket() 函数收到一个 pbuf 的数据之后, 它会首先判断 pbuf 中的数据长度 (即参数 ulLength, 见 lib_emac.c 文件中的 EMACSendPacket() 函数) 与当前使用的发送缓冲区长度是否相符, 如果长度超出, 则将其分割保存进多个发送缓冲区中。保存完毕, 函数会判断当前处理的 pbuf 是否已经到了链表的末尾, 也就是 pbuf->next 为空 (通过 bllsEndOfFrame 参数传递)。如果到了末尾并且数据复制完毕, 则置位当前缓冲区描述符的 Last Buffer 位, 没有, 则只需将当前缓冲区中保存的实际数据长度填充到缓冲区描述符中即可。如果我们在多次调用 EMACSendPacket() 函数之后到达了缓冲区队列的末尾, 也就是函数中使用的私有静态变量 __uwTxBlIndex 其索引计数等于发送缓冲区个数减 1 (即, __uwTxBlIndex == (NB_TX_BUFS-1)), 则索引计数立即复位为 0, 从缓冲区队列的开始位置继续保存剩余数据。同时, 置位当前缓冲区描述符的 Wrap 位, 以使发送缓冲区队列指针寄存器能够在数据发送时回到队列开始位置以顺利找到剩余数据。注意, 置位缓冲区描述符的 Wrap 位并不会使 EMAC 认为已经到达帧尾, 从而停止发送, 只有 Last Buffer 位置位才行。上述过程处理完毕, 最后, 如果标记了 Last Buffer 则立即启动传输 (置位网络控制寄存器 NCR 的位 9)。在这里需要进一步说明的是: 假设发送缓冲区队列存在 10 个缓冲区, 在发送第一帧数据时 (实际上就是调用了一次 __low_level_output() 函数) 我只使用了 7 个缓冲区, 这时 __uwTxBlIndex 索引计数就是 8, 其指向了下一个可用缓冲区, 同样在数据发送完毕后, 发送缓冲区队列指针寄存器 (EMAC_TBQP) 也指向了该位置。我需要再发送一帧数据时, 同样是再次调用 __low_level_output() 函数。现在要发送的这帧数据需要占用 10 个缓冲区队列。EMACSendPacket() 函数在复制数据到缓冲区时, __uwTxBlIndex 索引增加到了 10, 也就是到缓冲区末尾的时候, 前 7 个缓冲区并没有被使用, 所以函数将 __uwTxBlIndex 索引重新复位为 0, 剩余数据被保存到了前面。同样, 按照 EMAC 的数据手册, 置位 Wrap 位的目的就是使 EMAC_TBQP 寄存器重新指向发送队列开始位置。从上面的描述我们还看出, 发送队列一定要大于或等于帧的最大长度, 否则就会出现尾部数据覆盖头部数据的问题。

函数源码参见 lib_emac.c 文件, 文中就不再给出了。

5.4.10 编译——ethernetif.c 及 lib_emac.c

经过前面 9 节的努力, 所有底层驱动函数已经编写完成, 剩下的工作就是编译一遍 lib_emac.c 及 ethernetif.c 文件, 看看我们是不是还漏了一些工作, 比如 include 一些头文件等, 最终完成底层驱动的编写。

首先编译 ethernetif.c, 在编译之前, 我们先 include 一些头文件, 如下所示:

```
#include "/uCOS_II/includes.h"
#include "lwip/opt.h"
#include "lwip/def.h"
#include "lwip/mem.h"
#include "lwip/pbuf.h"
#include "lwip/sys.h"
#include "/LwIP/include/lwip/stats.h"
#include "/LwIP/include/netif/etharp.h"
```

现在编译, 如果你没有写错代码, 肯定能编译成功。如果读者编译时出错, 请参考笔者随本文档一同发布的 LwIPPortingTest_5 文件夹下的 ethernetif.c 文件,

第 2 个文件 lib_emac.c 同样是缺少一些头文件, 如下所示:

```
#include "/uCOS_II/includes.h"
#include "/at91sam7x256/include/AT91SAM7X256.h"
```

```
#include    "/at91sam7x256/include/lib_AT91SAM7X256.h"
#include    "/LwIP/include/lwip/opt.h"
#include    "lib_emac.h"
```

include 以后，编译即可成功。

6 ping——结束 LwIP 的移植

6.1 编译、链接整个工程

自从开始我们的移植旅程以来，我们还没有编译、链接整个工程，原因是当时的工程还很不完整。现在好了，经过我们的努力，工程已经完整，可以把它编译、链接成可执行文件了。先让我们编译、链接一遍，看看是不是能一遍成功（这可是最理想的结果），编译后的结果如下图所示：

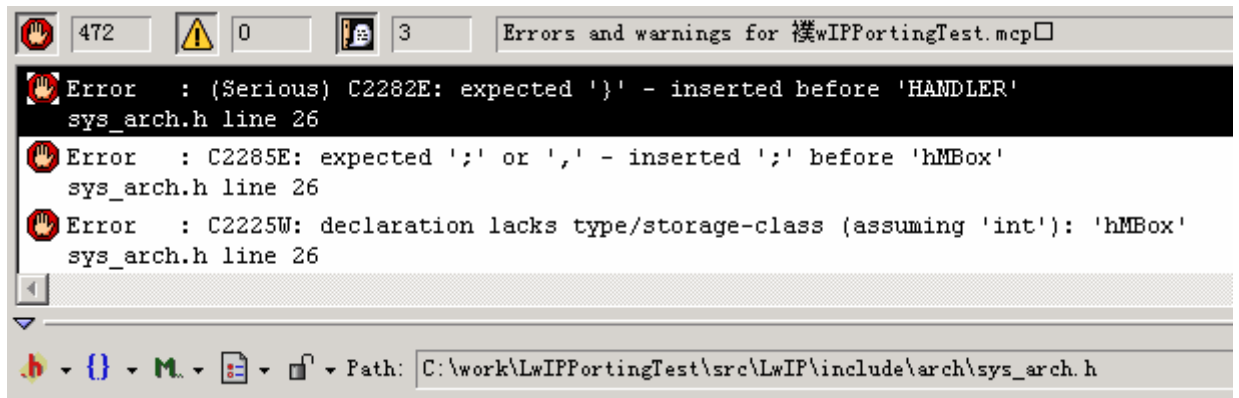


图 6.1.1

乖乖，472 个错误，很麻烦啊。不过想想主席的那首《七律·长征》：“红军不怕远征难，万水千山只等闲”，对于那次艰苦卓绝的长征来说，我们这些错误又算什么。还是老办法，一个个解决掉。

先看看第一个错误，产生这个错误的原因实际上是因为没有找到 HANDLER 这一自定义数据类型的声明。我们只需在 sys_arch.h 头文件中引入它的声明文件即可，如下图所示：

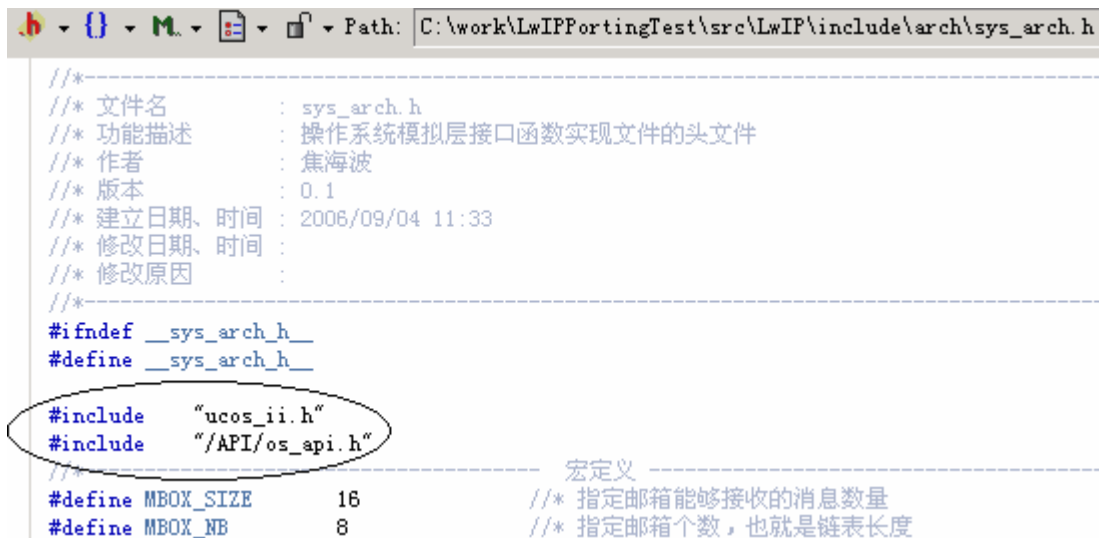


图 6.1.2

现在再编译、链接一遍，结果真让人不可思议，竟然还有 15 个，胜利的曙光就在前面。先看看第一个错误，说是一些宏出现重定义，这是怎么回事呢？原来这些宏与本地库提供的宏产生了冲突。这几个宏在<errno.h>中已经定义，而 sockets.c 文件使用了这个头文件，因此产生了重定义错误，我们只需在 sockets.c 文件中屏蔽掉相关的 include 语句即可，如下所示：

```
#include <string.h>
//#include <errno.h>

#include "lwip/opt.h"
#include "lwip/api.h"
#include "lwip/arch.h"
#include "lwip/sys.h"

#include "lwip/sockets.h"
```

图 6.1.3

继续编译、链接，错误还剩下 12 个。当前错误仍然还是缺少宏定义，这次缺少定义的是 BYTE_ORDER 宏，这个宏仅从字面意思看就能明白它被用于系统字节顺序的定义。笔者使用的系统是小端系统（即低八位在前，高八位在后），因此将其指定为小端系统即可。打开 cc.h 文件（这个文件被系统广泛引用，因此在这里定义），添加如下语句：

```
#define BYTE_ORDER LITTLE_ENDIAN
```

再编译、链接，错误还剩下 11 个。这 11 个是相同类型的错误，都是指针类型不匹配错误。造成这个错误的原因是__packed 限定词，如果把具备__packed 属性的指针变量赋值为普通的指针变量就会丢失__packed 属性（如下图所示），编译器在默认情况下就会报错。要解决这个问题有两个方法：一个是避免这种情况的出现，要么双方都具备__packed 属性，要么都不具备；另一个是让编译器忽略这种错误，不再报错。对于第一个方法，不仅工作量大，而且还牵扯 C 库函数的问题。比如我们在编译、链接时出现的如下一个错误：

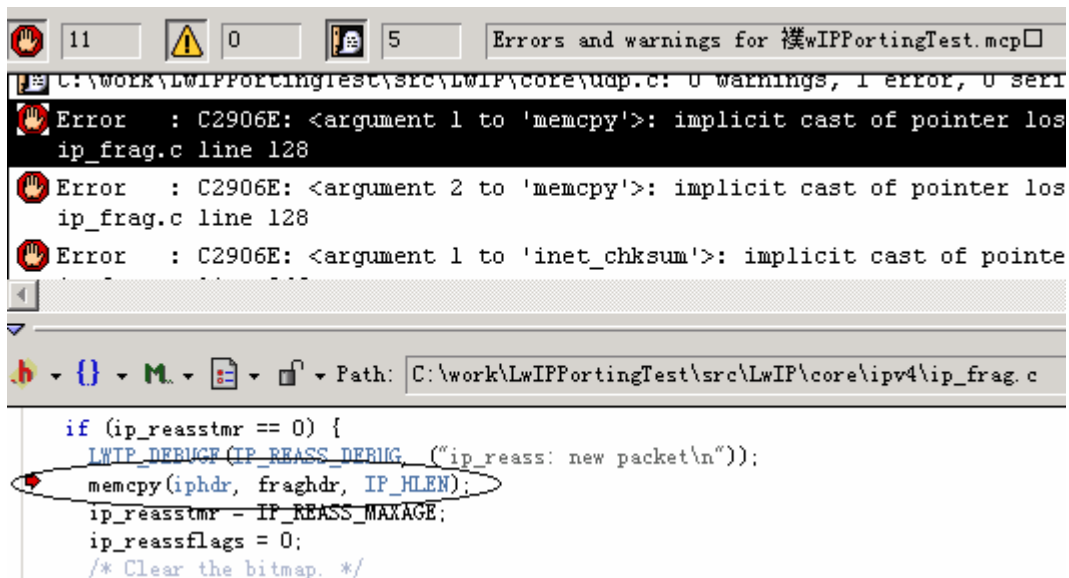


图 6.1.4

memcpy() 函数的第一个参数不具备__packed 属性，要让编译器不报错，我们唯一的选择就是去掉 iphdr 的__packed 属性，但是根据前文的讲述这样做显然是不对的，所以我们无法采用第一个方法。那么如何让编译器忽略这种错误呢？很简单，答案在工程设置里面，读者只要去掉相关选项即可（见下图）：

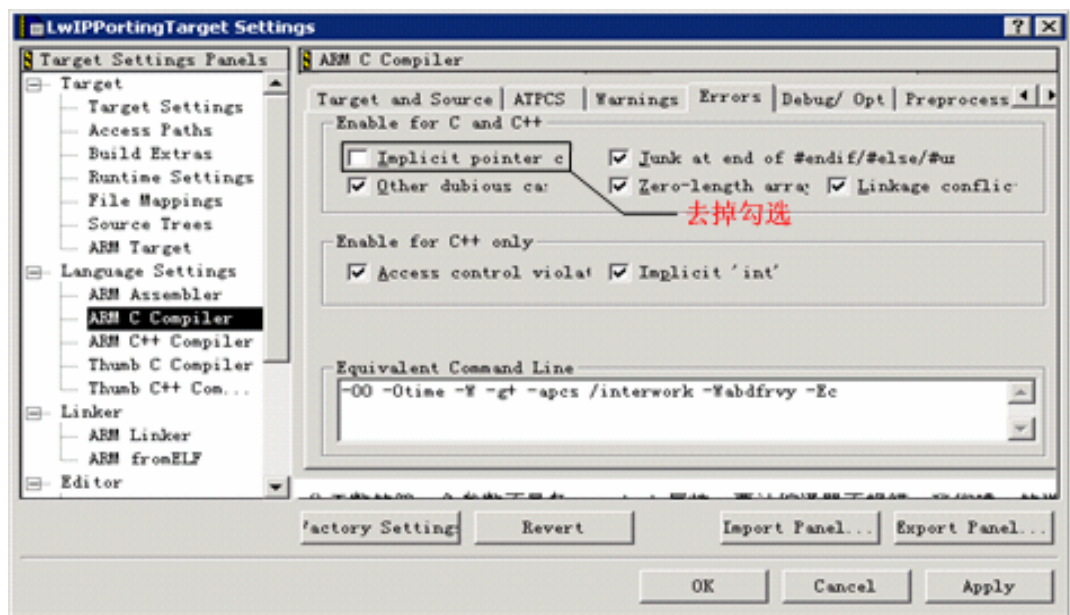


图 6.1.5

设置完成，再编译、链接一次，还剩一个错误，还是宏定义的问题。这个宏在调试输出时使用，我们不需要，所以只需在 debug.h（LwIP\include\lwip\debug.h）文件中添加如下一句即可：

```
#define LWIP_NOASSERT
```

再编译、链接，成功（完整工程在 LwIPPortingTest_6\LwIPPortingTest_6_1 文件夹下）。

6.2 ping 测试

在进行测试之前，我们必须先把LwIP添加到系统中。实际操作很简单，打开main.c，在 __TaskStart() 函数中建立LwIP的测试任务，相关源码在LwIPPortingTest_6\LwIPPortingTest_6_2 文件夹下。这项工作完成之后，我们还需要设置程序的执行地址及读写内存区地址，具体设置如下图所示：

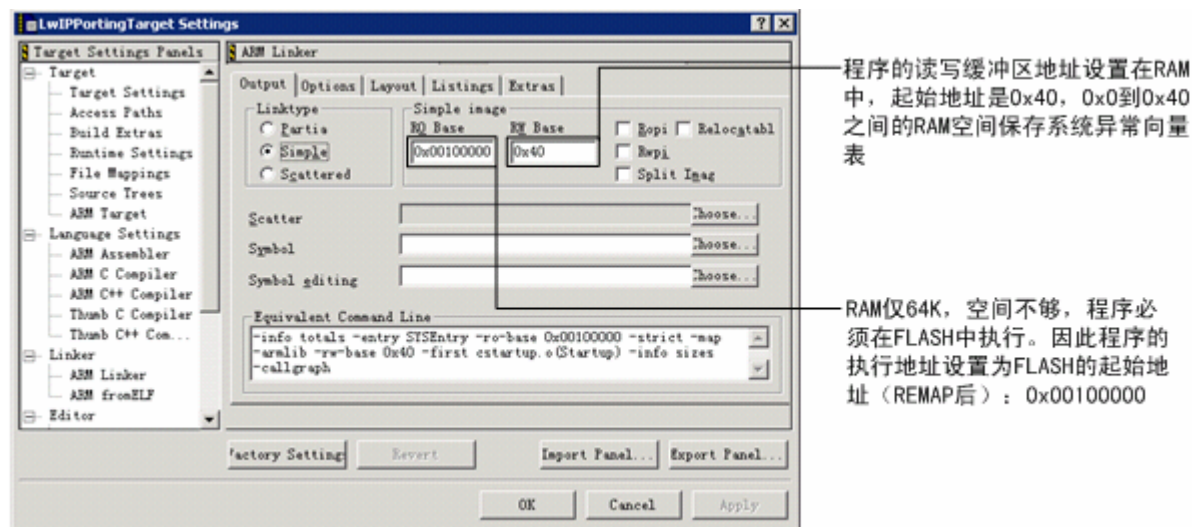


图 6.2.1

设置成功后，编译、链接整个工程，生成可执行文件，开始测试。在测试之前我们必须先把程序写

入 7x256 的 FLASH 中，否则无法仿真调试（FLASH 必须通过特殊指令组合才能写入程序，这与 RAM 完全不同）。把程序写入 FLASH 中的方法很多，读者可以使用专门的烧写工具，比如最新版本的 H-FLASHER（该软件支持 wiggler 仿真器）；也可以自己编写一个，使用 semihosting 方法下载程序。笔者使用 wiggler 仿真器（笔者的朋友设计制作，有需要的可以找我，MSN: marsstory99@hotmail.com），按理说可以使用 H-FLASHER 下载程序，但是笔者使用最新的 0.4 版本的 H-FLASHER 下载程序时总是出错，最终只得放弃（不知是不是笔者的方法不对，希望使用过的朋友多多指教）。笔者最终采用了第二个方法，动手编写了一个下载工具。该工具随本文档一同发布，名称为 FlashProgrammer。它是一个完整的 ADS 工程，读者可以直接使用。有关它的详细信息不在本文描述范围之内，这里不再赘述。笔者在这里只简单描述一下该工具的使用方法。

笔者在前文已经说过，我们自己编写的下载工具采用了 semihosting 方法。因此，在程序下载之前，必须使能仿真环境的 semihosting 模式。笔者为 wiggler 仿真器配套使用的驱动软件是 H-JTAG（官方网站: <http://twentyone.bokee.com/>），它允许用户打开或关闭 semihosting 模式，如下图所示：

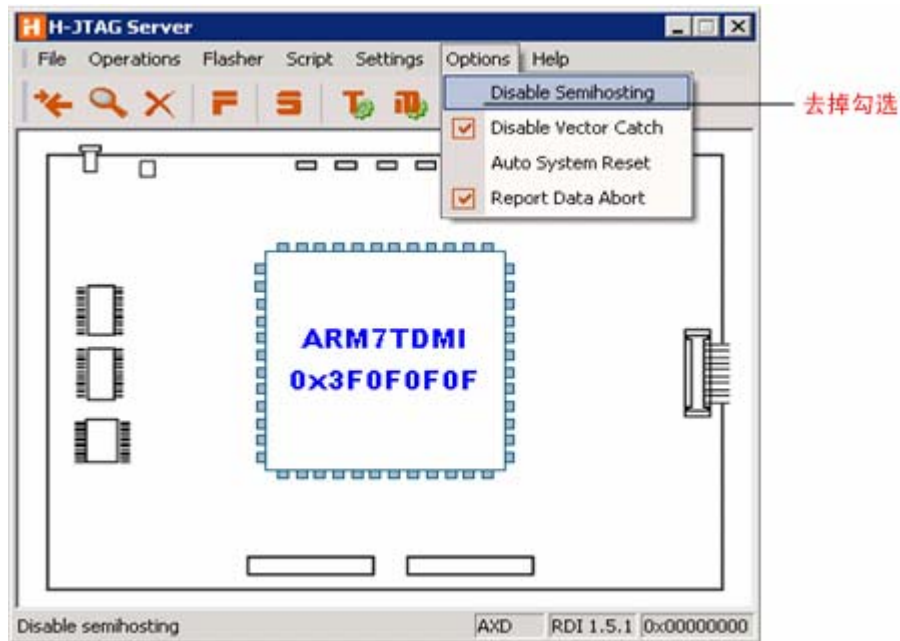


图 6.2.2

设置好 H-JTAG 后，我们还需要对 AXD 的 semihosting 模式进行设置，见下图：

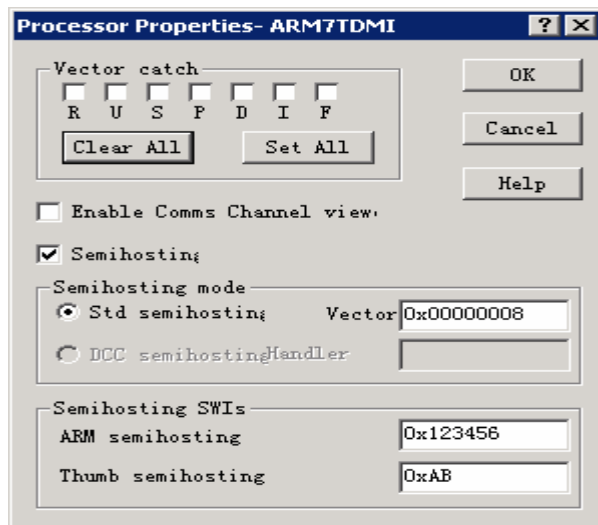


图 6.2.3

完成相关设置后，我们就可以使用ADS打开FlashProgrammer工程了。在这个工程中，需要读者修改的地方只有一处，见下图：

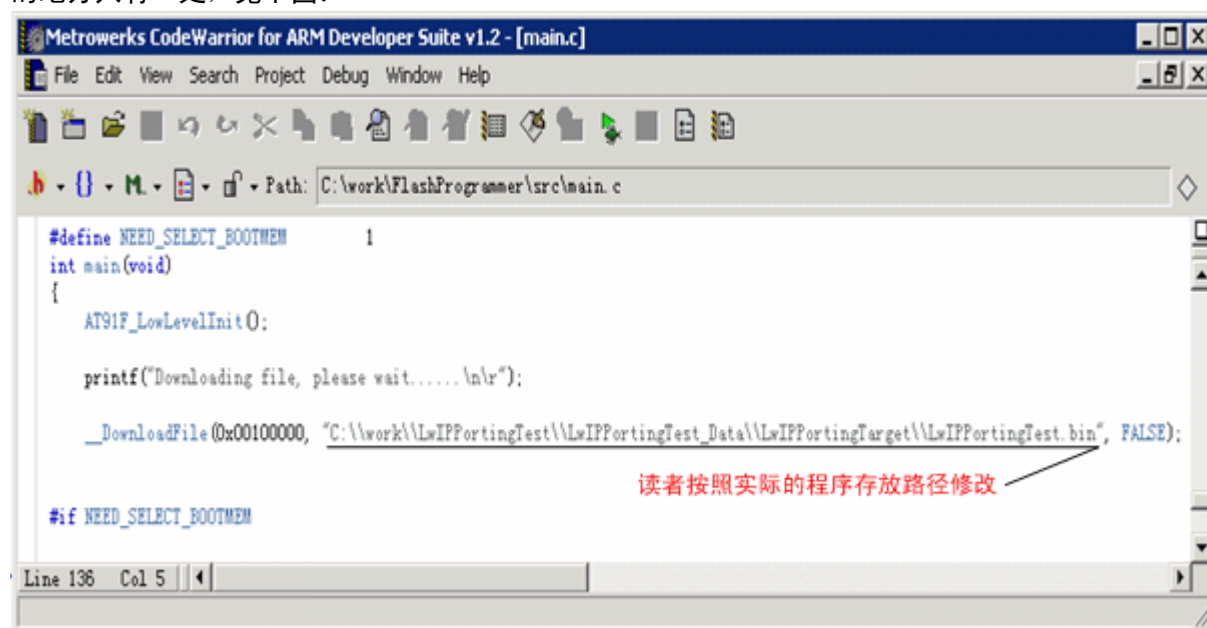


图 6.2.4

修改完成，按F5 进入AXD，再按两次F5，程序就开始下载了。下载时间可能会稍微长一些，据笔者本人测算，时间大约在 5、6 分钟左右。如果读者使用H-FLASHER下载，时间会大大缩短，原因是两者读取程序文件的方式不同，semihosting方式要慢很多。

程序下载完毕后，试验板接上网线，ping 一下，看看我们的板子能否正常响应。在笔者的板子上，这一步很顺利，见下图：

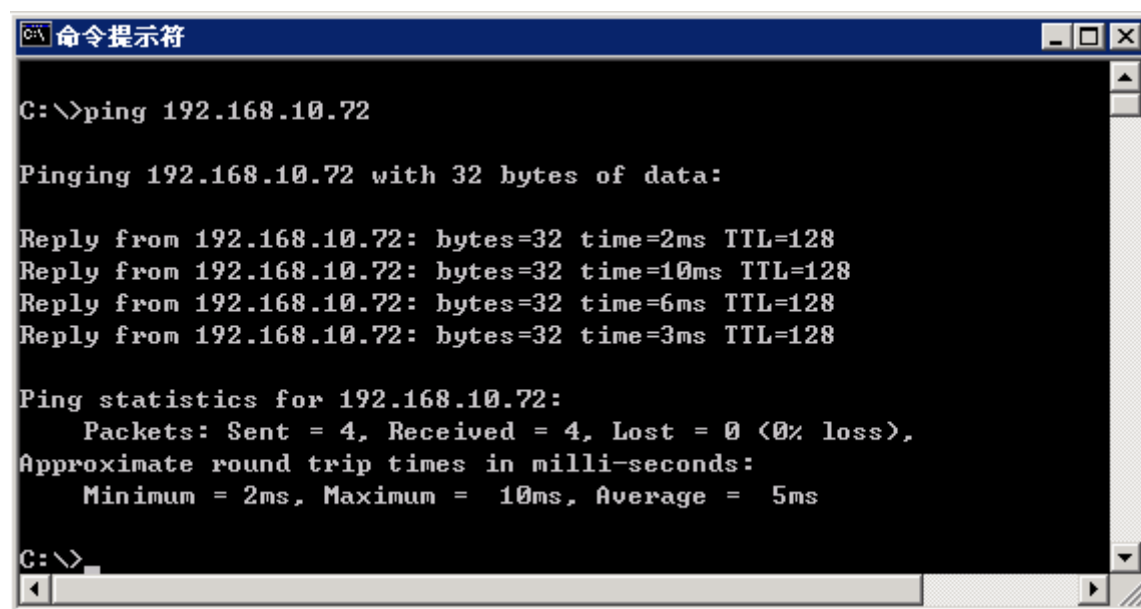


图 6.2.5

如果读者也能像笔者一样这么顺利，那么我们的移植工作就只剩最后一步了——测试上层协议。笔者为此建立了一个简单的WEB 服务器(源码在LwIPPortingTest_6\LwIPPortingTest_6_3文件夹下，仍然在T_LwIPEntry()测试任务中)，浏览器访问这个WEB 服务器的结果如下图所示：



图 6.2.6

至此，我们的移植工作大功告成。在结束本文之前，还有一件事需要特别说明：opt.h 中有一个指定字节对齐方式的宏——MEM_ALIGNMENT，这个宏需要根据自己的实际情况作出修改，否则会造成严重后果，这一点读者一定要注意。

后记

笔者移植使用的硬件平台还是比较稳定的，前期发布的带DNS客户端的源码既是在这个平台上完成，有需要的朋友可以与笔者联系（笔者的MSN前文已经公布）。（DNS客户端源码下载地址：http://groups.google.com/group/marsstory/browse_thread/thread/76ee5a496501eb61/#）。