

1 背景.....	5
2 简介.....	5
3 移植.....	6
3.1 以太网应用场景下的移植.....	6
3.1.1 将 onps 栈代码添加到工程.....	6
3.1.2 sys_config.h——配置协议栈 .....	8
3.1.3 datatype.h——基础数据类型定义 .....	9
3.1.4 编写 OS 适配层相关代码.....	9
3.1.5 编写网卡驱动并注册网卡 .....	18
3.1.6 dhcp 动态地址申请测试 .....	27
3.1.7 ping 测试 .....	29
3.1.8 dns 测试.....	35
3.1.9 sntp 网络校时测试.....	38
3.1.10 tcp 客户端测试.....	40
3.1.11 tcp 服务器测试.....	48
3.1.12 udp 通讯测试 .....	53
3.2 ppp 拨号场景下的移植.....	57
3.2.1 准备工作 .....	61
3.2.2 配置协议栈 .....	63
3.2.3 os 适配层——编写 tty 驱动.....	64
3.2.4 调整相关系统设置 .....	66
3.2.5 ping 等网络工具测试 .....	70
3.2.6 tcp 及 udp 客户端测试 .....	71

4 socket 函数使用说明 .....	74
socket.....	74
close .....	75
connect.....	76
connect_nb .....	77
is_tcp_connected.....	78
send.....	80
send_nb .....	81
is_tcp_send_ok.....	82
sendto .....	83
recv .....	84
socket_set_rcv_timeout .....	85
recvfrom .....	86
bind .....	87
listen.....	87
accept .....	88
tcpsrv_rcv_poll.....	90
socket_get_last_error/onps_get_last_error.....	90
socket_get_last_error_code .....	92
5 常用工具函数.....	92
htonll .....	93
htonl .....	93
htons .....	94

inet\_addr ..... 94

inet\_addr\_small ..... 94

inet\_ntoa ..... 95

inet\_ntoa\_ext..... 95

inet\_ntoa\_safe..... 96

inet\_ntoa\_safe\_ext ..... 96

ip\_addressing..... 97

strtok\_safe ..... 98

snprintf\_hex..... 98

printf\_hex..... 99

onps\_error..... 100

6 获取源码..... 100

7 后续工作计划..... 101

修改历史		
版本	日期	说明
1.0.0	2022-10-17	onps 栈初始版本移植及使用说明
1.1.0	2023-07-26	<div>1. 增加了 TCP SACK 选项支持； 2. 增加了 ipv6 支持； 3. 增加了 telnet 服务支持； 4. 增加了虚拟网络终端 NVT 支持； 5. 增加了新的底层 API 及工具函数； 6. 修改了部分函数原型定义； 7. 修正了一些描述错误；</div>

# 1 背景

大约是 06 年，因项目之需我开始接触应用于单片机系统的国外开源 tcp/ip 协议栈——LwIP，并借此顺势创作了我的第一本印成铅字的书——《嵌入式网络系统设计——基于 Atmel ARM7 系列》。这本书的反响还不错，好多人给我发 msn（可惜这么好的一个即时通讯工具就这么被微软放弃了，好多联系人就此失联，☹）或邮件咨询相关问题。在我原来的写作计划中，这本书的出版只是一个开始，接下来还要写第二本——系统介绍 LwIP 包含的 ppp 协议栈的移植、应用及设计实现等相关内容。但，事与愿违，这本书跳票了，且这一跳就是十二年……

细细想来，当初跳票的主因有二：其一，因家庭、工作等致可支配时间太少；其二，缺乏足够的 ppp 协议相关知识及技术储备致信心不足，畏首畏尾，裹足不前。但，这件事始终是我的一个遗憾。十二年的时间，不长亦不短，但足够让心底的遗憾变成一粒小小的种子并茁壮成长为一棵梦想的参天大树。

如今，世界来到了疫情肆虐的二零年代。我的可支配时间多了起来，技术能力亦远非当年可比。梦想之树到了开花结果的时候了。遥想当初，入行还没几年，技术能力有限，我只能站在大神的肩膀上研究如何移植、使用 LwIP，ppp 栈碰都没敢碰。现在，如果还只是延续十几年前的工作，那这件事做起来就无甚意义。基于对自身技术实力的准确认识，我决定自己从零开始搭建一个完整的网络协议栈。终，历 6 个月余，onps 协议栈（onps, open net protocol stack）完成初版开发，并内部测试通过。十余年的遗憾今日得偿。另，从业 20 余年，内心终有一个做核心基础软件的梦。今，这二之梦想亦借此得偿。

新莺初啼，总免不了会有诸多不尽如人意的地方。开源，则可与志趣相投者共享、共用、共研，历诸位严苛手段使之快速迭代，快速成熟，比肩 LwIP 可期☺。

# 2 简介

onps 是一个开源且完全自主开发的国产网络协议栈，适用于资源受限的单片机系统，提供完整的 ethernet/ppp/tcp/ipv4/ipv6 协议族实现，同时提供 snmp、dns、ping、telnet 等网络工具，支持以太网环境下 dhcp 动态 ip 地址申请，也支持动态及静态路由表。协议栈还封装实现了一个伯克利套接字（Berkeley sockets）层。该层并没有完全按照 Berkeley sockets 标准设计实现，而是根据以往 socket 编程经验，以方便用户使用、简化用户编码为设计目标，重新声明并定义了一组常见 socket 接口函数。协议栈简化了传统 BSD socket 编程需要的一些繁琐操作，将一些不必要的操作细节改为底层实现，比如 select/poll 模型、阻塞及非阻塞读写操作等。

为了适应单片机系统对内存使用极度变态的苛刻要求，onps 协议栈在设计之初即考虑采用写时零复制（zero copy）技术。用户层数据在向下层协议传递过程中，协议栈采用缓存链表（buffer list）技术将它们链接到一起，直至将其发送出去，均无须任何内存复制操作。另外，协议栈采用 buddy 算法提供安全、可靠的动态内存管理功能，以期最大限度地提高协议栈运行过程中的内存利用率并尽可能地减少内存碎片。

不同于本世纪 00 到 10 年代初，单片机的应用场景中 ucosii 等 rtos 尚未大规模普及，前后台系统还大行其道的时代，现如今大部分的应用场景下开发人员选择使用 rtos 已成为主流。因此，协议栈在设计之初即不支持前后台模式，其架构设计建立在时下流行的 rtos（RT-Thread、ucosii/iii 等）之上。协议栈移植的主要工作也就自然是针对不同 rtos 编写相关 os 适配层功能函数了。当然，如果你有着极其特定的应用场景，需要将 onps 栈移植到采用前后台模式的单片机上，我的建议是保留 tcp/udp 之下协议层的通讯处理逻辑，调整上层的系统架构使其适应目标系统运行模式。

## 3 移植

整个移植说明按照两个主要应用场景展开：一，使用有线以太网网络联网的场景；二，使用 4g/5g 模块拨号联网的场景。前者需要在实际的以太网卡上实现 tcp/ip 通讯；后者则需要利用 ppp 栈构建一个虚拟的点对点网卡来实现 tcp/ip 通讯。其实这两个应用场景本质上也没什么区别，ip 层以上的处理逻辑完全一致。之所以分开进行讲解，主要是因为拨号联网的场景需要实现 ppp 栈，而以太网网络则是实现 ethernet 协议。两者在底层实现上有着很大的区别，与目标系统进行适配的移植工作并不一致，所以需要单独拿出来分别进行讲解。

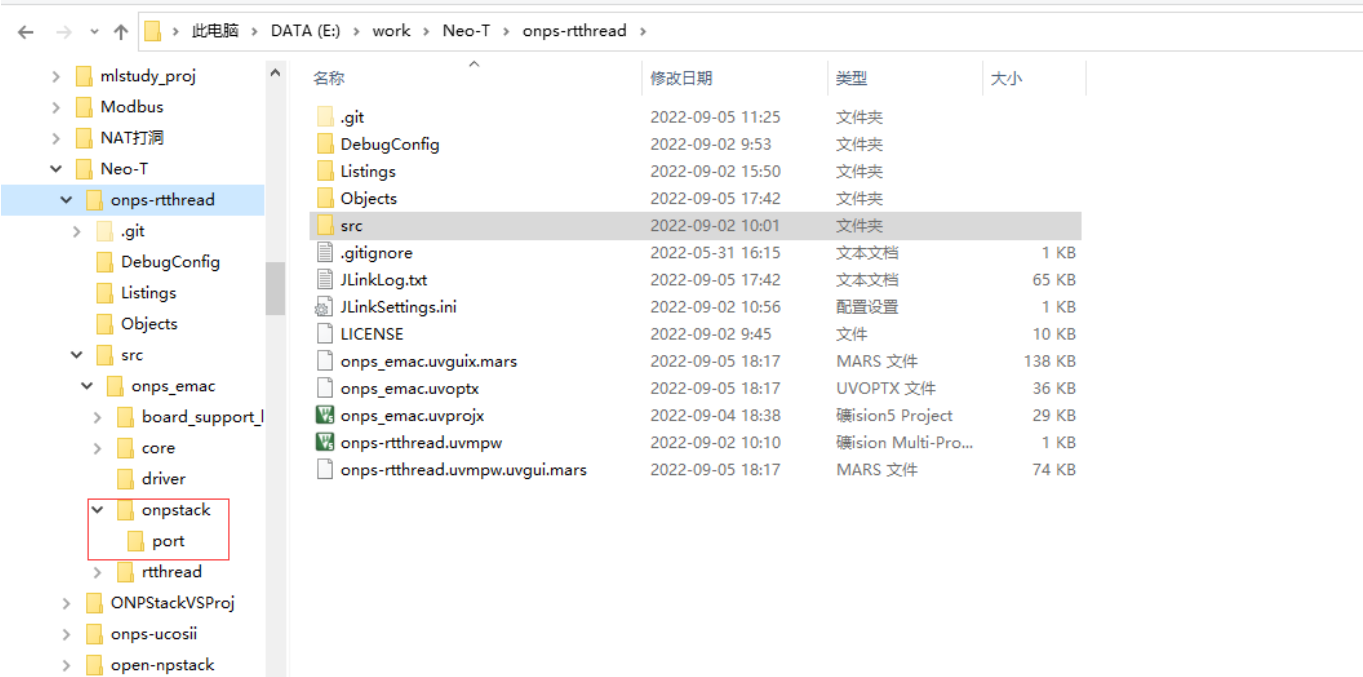
我熟悉并使用过的 rtos 除了 ucousii 就是 rt-thread，所以本移植说明的目标 os 为 ucousii 及 rt-thread。如果你的目标系统是其它 rtos，比如 FreeRTOS，只要这个 rtos 提供信号量、互斥锁操作机制那就没什么问题，参照这个说明你照样可以轻松完成移植的相关工作。接下来的移植说明将针对这两个实时操作系统展开。另外，移植说明使用的 IDE 为 Keil V5 MDK。

### 3.1 以太网应用场景下的移植

动手之前，你应该准备一个能够在你的目标板上正常进行任务调度的 rtos 基础工程（本说明以 rt-thread、ucousii 两个目标系统作为演示工程）。同时你的目标板最好能够提供 printf 输出调试信息的功能。这项功能非常重要，对于排查移植过程中出现的各种问题很有帮助。目前市面上常见的单片机开发板，比如正点原子的 stm32 系列都提供了该功能，所以接下来的移植工作假设你的板子支持 printf 函数。当然，实在不提供也不会影响移植工作，只是会影响排查问题的效率。

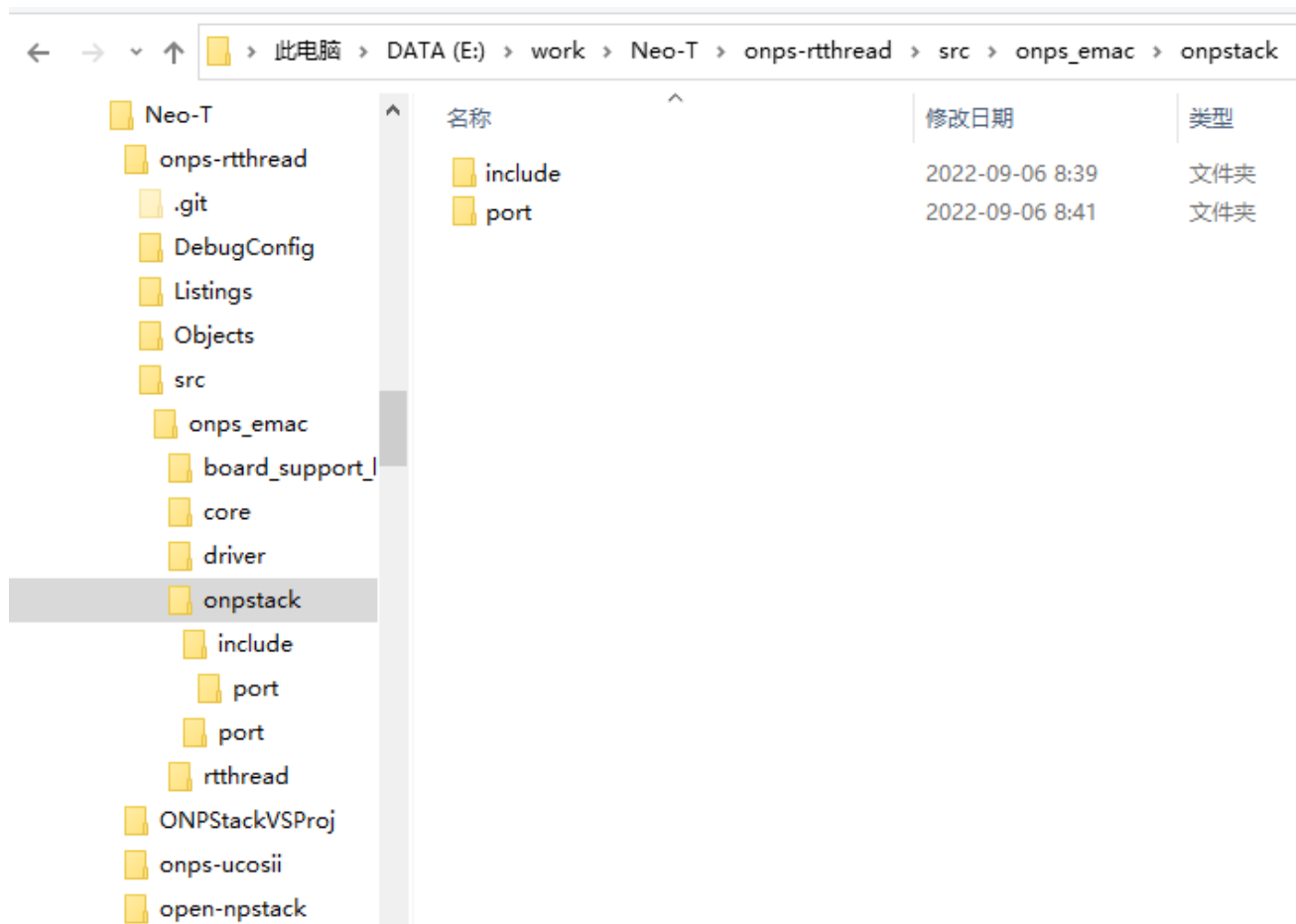
#### 3.1.1 将 onps 栈代码添加到工程

首先在你的 IDE 工程所在的目录下新建一个 onpstack 目录，然后在该目录下再建立一个 port 目录，与协议栈移植相关的文件均放在该目录下。建立完毕后 IDE 工程目录的组织情况如下：



从 gitee 或 github 下载 onps 协议栈代码到本地，下载地址参见[第 6 节](#)。源码下载到本地后将 port

目录下的 `os_adapter.c` 文件复制到你的 IDE 工程目录下，目标地址就是刚才建立的 `onpstack/port`，然后再把 `port` 目录下的 `include` 文件夹复制到刚才建立的 `onpstack` 目录下：



这样与目标系统相关的文件就准备好了，我们直接将这几个文件添加到工程中即可。接下来再把协议栈用到的核心文件添加到工程中：

```

bsd/: socket.c
ethernet/: arp.c ethernet.c
    ip/: icmp.c ip.c tcp.c tcp_link.c tcp_options.c udp.c udp_link.c
    mmu/: buddy.c buf_list.c
    netif/: netif.c route.c
include/: bsd/socket.h ethernet/arp.h ethernet/arp_frame.h ethernet/ethernet.h ethernet/ethernet_frame.h
    ethernet/ethernet_protocols.h ip/icmp.h ip/icmp_frame.h ip/ip.h ip/ip_frame.h ip/tp.h ip/tcp_frame.h
    ip/tcp_link.h ip/tcp_options.h ip/udp.h ip/udp_frame.h ip/udp_link.h mmu/buddy.h mmu/buf_list.h
    netif/netif.h netif/route.h one_shot_timer.h onps.h onps_errors.h onps_input.h onps_utils.h protocols,h
one_shot_timer.c
onps_entry.c
onps_errors.c
onps_input.c
onps_utils.c

```

协议栈其它文件属于可选模块，与系统配置项有关。比如当我们需要动态地址分配时，就需要把 `ethernet` 文件夹下 “`dhcp_`” 为前缀的文件添加到工程中。以此类推，目标系统需要哪个模块，就把其对应文件添加到工程中。系统可选配置模块及对应文件如下：

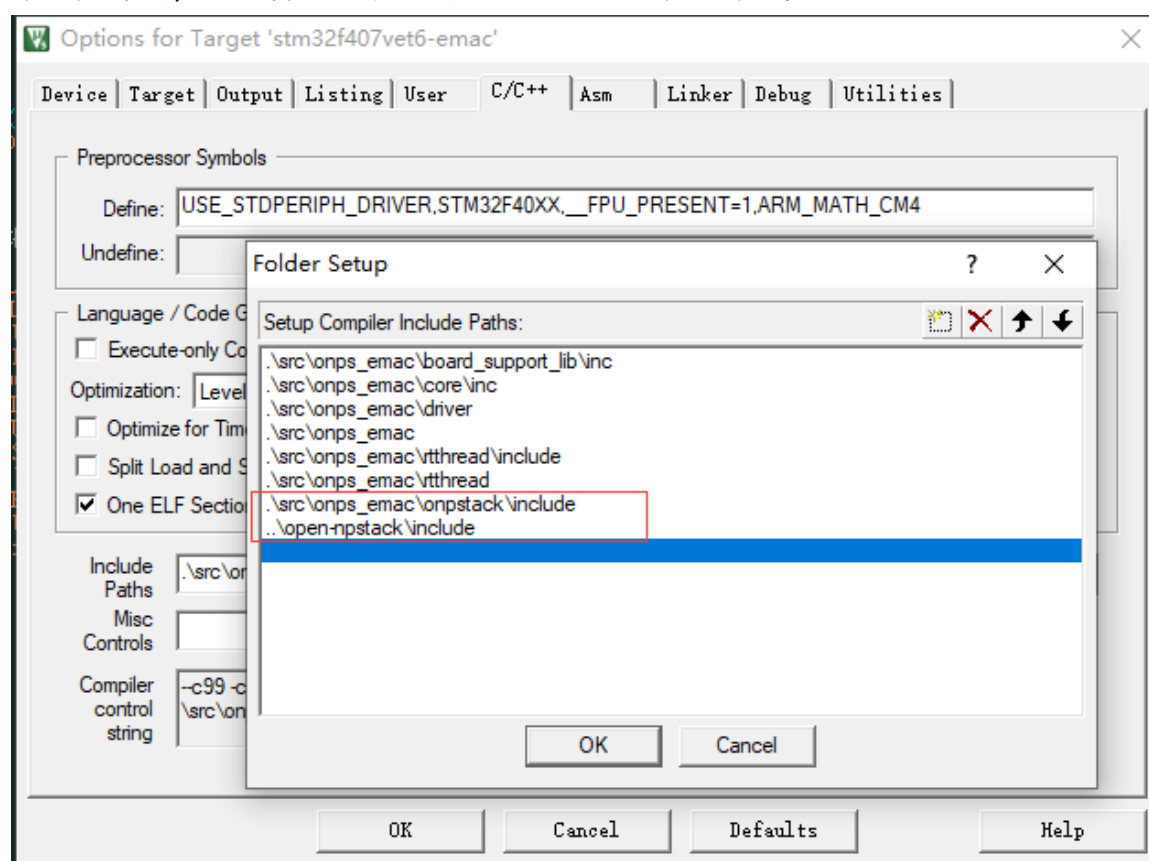
■ **ipv6:** `ethernet/dhcpv6.c(.h)` `ethernet/dhcpv6_frame.h` `ip/ipv6_onfigure.c(.h)` `ip/icmpv6.c(.h)`

ip/icmpv6\_frame.h

- **dhcp:** ethernet/dhcp\_client.c(.h) ethernet/dhcp\_client\_by\_timer.c(.h) ethernet/dhcp\_frame.h
- **net\_tools**
  - **dns:** net\_tools/dns.c(.h)
  - **ping:** net\_tools/ping.c(.h)
  - **sntp:** net\_tools/sntp.c(.h)
  - **telnet:** net\_tools/telnet.c(.h) net\_tools/telnet\_srv.c(.h) net\_tools/telnet\_client.c(.h)  
net\_tools/net\_virtual\_terminal.c(.h)

其中 telnet\_client.c(.h) 文件还可以根据需求裁剪掉。有关裁剪 onps 栈的详细说明请参阅《onps 栈移植手册》第一节“onps 栈的配置及裁剪”。

添加完源码，最后再把协议栈的 include 路径添加到工程中：



如此，整个基础工程准备完毕，接下来就进入实际的移植工作了。

### 3.1.2 sys\_config.h——配置协议栈

这个文件是协议栈的配置文件，我们可以根据目标系统的具体情况对协议栈进行裁剪，减少或增加对系统资源的占用率。所有配置项均有详细地注释，同时还有《onps 栈移植手册》供参考，不再赘言。惟一需要特别说明的是 SUPPORT\_PPP 和 SUPPORT\_ETHERNET 宏，这两个宏用于显式地打开或关闭协议栈对 ppp、以太网的支持。显然，针对我们当前的移植工作，我们需要 SUPPORT\_PPP 宏置 0，关闭 ppp；SUPPORT\_ETHERNET 宏置 1，打开以太网模块。



### 3.1.3 datatype.h——基础数据类型定义

首先，要实现一个字节对齐的宏定义：

```
#define PACKED __attribute__((packed)) /* 实现字节对齐
#define PACKED_FIELD(x) PACKED x      /* 指定单个变量字节对齐
#define PACKED_BEGIN                  /* 字节对齐开始，armcc 编译器不需要
#define PACKED_END                    /* 字节对齐结束，armcc 编译器不需要
```

以上字节对齐的宏定义是针对我使用的 IDE 提供的 armcc 编译器定义的，其它编译器要根据其手册做出对应修改。这个宏是整个协议栈能否正常运转的关键，所有与网络传输相关的通讯协议均须字节对齐，其它对齐方式无法实现正常网络通讯。所以，必须确保该宏能正常工作。

另外一部分工作就是基础数据类型定义：

```
/* 系统常用数据类型定义(不同的编译器版本，各数据类型的位宽亦不同，请根据后面注释选择相同位宽的类型定义)
typedef unsigned long long ULONGLONG; /* 64 位无符号长整型
typedef long long          LONGLONG;  /* 64 位有符号长整型
typedef signed long        LONG;      /* 32 位的有符号长整型
typedef unsigned long      ULONG;     /* 32 位的无符号长整型
typedef float              FLOAT;     /* 32 位的浮点型
typedef double             DOUBLE;    /* 64 位的双精度浮点型
typedef signed int         INT;       /* 32 位的有符号整型
typedef unsigned int       UINT;     /* 32 位的无符号整型
typedef signed short       SHORT;    /* 16 位的有符号短整型
typedef unsigned short     USHORT;   /* 16 位的无符号短整型
typedef char               CHAR;     /* 8 位有符号字节型
typedef unsigned char      UCHAR;    /* 8 位无符号字节型
typedef unsigned int       in_addr_t; /* internet 地址类型
```

### 3.1.4 编写 OS 适配层相关代码

对于 os 适配层，主要的移植工作就几块：1) 提供多任务（线程）建立函数；2) 提供系统级的秒级、毫秒级延时函数及运行时长统计函数；3) 提供同步（互斥）锁相关操作函数；4) 提供信号量操作函数；5) 提供一组临界区保护也就是中断禁止/使能函数。os 适配层的移植工作涉及 os\_datatype.h、os\_adapter.h、os\_adapter.c 三个文件。

#### 1) os\_datatype.h

这个文件要完成的主要工作就是声明两个与目标操作系统相关的数据类型：互斥锁、信号量，不同的 rtos 其定义是不同的，在这里我分别给出 rt-thread 和 ucspi 下这两个数据类型的声明样例。首先是 rt-thread 下的：

```
#include <rtthread.h>
typedef struct rt_mutex *HMUTEX;          /* 线程同步锁句柄
#define INVALID_HMUTEX (struct rt_mutex *)0 /* 无效的线程同步锁句柄

typedef struct rt_semaphore *HSEM;        /* 信号量，适用与不同线程间通讯
#define INVALID_HSEM (struct rt_semaphore *)0 /* 无效的信号量句柄
```

ucspi 下的：

```
#include "ucos_ii.h"
#include "os_cpu.h"
#include "os_cfg.h"
typedef OS_EVENT* HMUTEX;          /* 线程同步锁句柄 */
#define INVALID_HMUTEX (OS_EVENT*)0 /* 无效的线程同步锁句柄 */

typedef OS_EVENT* HSEM;           /* 信号量，适用与不同线程间通讯 */
#define INVALID_HSEM (OS_EVENT*)0 /* 无效的信号量句柄 */
```

除了以上两个数据类型，还需要定义几个 c 语言编程常用的宏及协议栈用到的两个特殊数据类型：

```
#ifndef NULL
#define NULL ((void *)0)
#endif

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

typedef unsigned int BOOL; /* bool 型变量，如果你的目标系统已经定义，这里就不需要定义了，注释或删除本行即可 */

/* socket 编程中常用的 internet 地址类型 */
#ifndef s_addr
struct in_addr
{
    in_addr_t s_addr;
};
#define s_addr s_addr
#endif
```

## 2) os\_adapter.h

这个文件要调整地方不多。首先需要调整的地方就是 STCB\_PSTACKTHREAD 结构体。它用于建立协议栈内部工作线程。线程入口函数在不同的 os 上可能存在些微的差别。我们只需根据这些差别调整该结构体的相应字段即可。不过对于 rt-thread 和 ucosii 来说，线程入口函数的声明类型虽然完全相同，但这两个系统提供的线程（任务）建立函数的结构却有所不同，所以需要为 STCB\_PSTACKTHREAD 结构体分别针对这两个系统增加几个字段。首先是 rt-thread 下的：

```
typedef struct _STCB_PSTACKTHREAD_ { /* 协议栈内部工作线程控制块，其用于线程建立 */
    void(*pfunThread)(void *pvParam); /* 指向线程入口函数的指针，请根据 os 线程入口函数的实际定义修改该字段 */
    void *pvParam; /* 传递给线程入口函数的用户参数，请根据 os 线程入口函数的实际定义增加或减少用户参数字段 */
    const CHAR *pszThreadName; /* 线程名称 */
    UINT unStackSize; /* 栈大小 */
    UCHAR ubPrio; /* 线程优先级 */
    UINT unTimeSlice; /* 单次调度该线程能够运行的最长时间片 */
};
```

ucosii 下需要增加的字段:

```
typedef struct _STCB_PSTACKTHREAD_ { /* 协议栈内部工作线程控制块, 其用于线程建立
    void(*pfunThread)(void *pvParam); /* 指向线程入口函数的指针, 请根据 os 线程入口函数的实际定义修改该字段
    void *pvParam; /* 传递给线程入口函数的用户参数, 请根据 os 线程入口函数的实际定义增加或减少用户参数字段
    OS_STK *pstkTop; /* 栈顶地址
    UCHAR ubPrio; /* 线程优先级
};
```

这个结构体增加的字段将在编码实现 `os_thread_onpstack_start()` 函数时使用。

接着我们再提供一组临界区保护函数, 这组临界区保护函数的实现目的就是工作线程一旦进入临界区, 中断及 OS 的任务调度都不能打断当前代码的执行, 直至线程退出临界区。要想达成这个目的其实很简单, 进入临界区之前关中断, 退出后开中断即可。首先给出 `rt-thread` 下的实现代码:

```
extern rt_base_t rt_hw_interrupt_disable(void);
extern void rt_hw_interrupt_enable(register rt_base_t temp);
#define os_critical_init() register rt_base_t temp; /* 临界区初始化
#define os_enter_critical() temp = rt_hw_interrupt_disable(); /* 进入临界区 (关中断)
#define os_exit_critical() rt_hw_interrupt_enable(temp); /* 退出临界区 (开中断)
```

ucosii 下的实现代码:

```
#define os_critical_init() OS_CPU_SR cpu_sr = 0; /* 临界区保护函数初始化
#define os_enter_critical() OS_ENTER_CRITICAL(); /* 进入临界区 (关中断)
#define os_exit_critical() OS_EXIT_CRITICAL(); /* 退出临界区 (开中断)
```

需要注意的是, 虽然这个文件要完成的移植工作量较小, 但这个文件非常重要。它声明的所有与 `os` 相关的接口函数均需要实现。它是我们完成 `os` 适配层移植工作的关键索引文件。

### 3) `os_adapter.c`

这个文件的核心工作就是编码实现 `os_adapter.h` 文件声明的所有与 `os` 相关的接口函数。首先是几个系统级的时间函数 `os_sleep_secs()`、`os_sleep_ms()`、`os_get_system_secs()`、`os_get_system_msecs()`, 其用于协议栈延时及计时。对于前两个延时函数, 实现起来一点都不难, 直接调用 `os` 提供的延时函数就行了。

`rt-thread`:

```
/* 当前线程休眠指定的秒数, 参数 unSecs 指定要休眠的秒数
void os_sleep_secs(UINT unSecs)
{
    rt_thread_mdelay(unSecs * 1000);
}

/* 当前线程休眠指定的毫秒数, 单位: 毫秒
void os_sleep_ms(UINT unMSecs)
{
    rt_thread_mdelay(unMSecs);
}
```

ucosii:

```
/* 当前线程休眠指定的秒数, 参数 unSecs 指定要休眠的秒数
void os_sleep_secs(UINT unSecs)
{
    OSTimeDlyHMSM(0, 0, unSecs, 0);
}

/* 当前线程休眠指定的毫秒数, 单位: 毫秒
void os_sleep_ms(UINT unMSecs)
{
    OSTimeDlyHMSM(0, 0, 0, unMSecs);
}
```

os\_get\_system\_secs() 函数的实现相对麻烦一些, 它其实与 os 关系并不大, 而是与你的 os 调度定时器中断有关。我们需要在定时器中断中实现秒级及毫秒级计数, 然后再提供一个读取这个秒级计数的函数即可。在我的目标工程中, 我的定时器中断函数是在 sys\_timer.c 文件中实现的, 因此我在这个文件头部定义了两个 32 位无符号整型变量, 然后在定时器中断中分别按秒、毫秒递增:

```
static volatile uint32_t lo_unElapsedSecs, lo_unElapsedMSecs, lo_unCurSysticks;

/* systick 中断服务函数, 使用 rt-thread/ucosii 时内核调度中断
void SysTick_Handler(void)
{
    .....

    /* 增加的秒级计时代码
    lo_unCurSysticks++;
    lo_unElapsedMSecs += 1000 / OS_TICKS_PER_SEC; /* rt-thread 下将 OS_TICKS_PER_SEC 换为 RT_TICK_PER_SECOND
    if(!(lo_unCurSysticks % OS_TICKS_PER_SEC))
        lo_unElapsedSecs++;
    .....
}

/* 返回系统定时器自启动以来的工作时长, 单位: 秒
uint32_t GetElapsedSecs(void)
{
    return lo_unElapsedSecs;
}

/* 返回系统定时器自启动以来的工作时长, 单位: 毫秒
uint32_t GetElapsedMSecs(void)
{
    return lo_unElapsedMSecs;
}
```

回到 os\_adapter.c 文件中, 实现 os\_get\_system\_secs() 及 os\_get\_system\_msecs() :

```
/* 获取系统启动以来已运行的秒数 (从 0 开始)
UINT os_get_system_secs(void)
```

```
{
    return GetElapsedSecs();
}

/** 获取系统启动以来已运行的毫秒数（从 0 开始）
UINT os_get_system_msecs(void)
{
    return GetElapsedMsecs();
}
```

这个函数无论目标 os 是啥，以上代码通用。接下来就是编码实现协议栈要用到的互斥锁函数了。首先就是 rt-thread 下的实现代码：

```
/** 线程同步锁初始化，成功返回同步锁句柄，失败则返回 INVALID_HMUTEX
HMUTEX os_thread_mutex_init(void)
{
    HMUTEX hMutex = rt_mutex_create("rt-mutex", RT_IPC_FLAG_FIFO);
    if(RT_NULL != hMutex)
        return hMutex;

    return INVALID_HMUTEX; /** 初始失败要返回一个无效句柄
}

/** 线程同步区加锁
void os_thread_mutex_lock(HMUTEX hMutex)
{
    rt_mutex_take(hMutex, RT_WAITING_FOREVER);
}

/** 线程同步区解锁
void os_thread_mutex_unlock(HMUTEX hMutex)
{
    rt_mutex_release(hMutex);
}

/** 删除线程同步锁，释放该资源
void os_thread_mutex_uninit(HMUTEX hMutex)
{
    rt_mutex_delete(hMutex);
}
```

ucosii 下互斥锁的实现代码相对麻烦一些，因为它需要显式地指定优先级继承优先级（PIP），而且每创建一个新的互斥锁，都需要指定一个不同的 PIP。所以，我们需要提前预留出一块 PIP 资源给协议栈使用：

```
#define MTX_PIP_START_PRIO 20 /** 协议栈用到的互斥锁优先级继承优先级（PIP），每建立一个互斥锁，优先级递减
static CHAR l_bTHMtxCurPIP = MTX_PIP_START_PRIO;

/** 线程同步锁初始化，成功返回同步锁句柄，失败则返回 INVALID_HMUTEX
```

```
HMUTEX os_thread_mutex_init(void)
{
    OS_CPU_SR cpu_sr = 0u;
    UCHAR ubErr;
    CHAR bTHMtxCurPIP;

    if(1_bTHMtxCurPIP <= 0) /* 如果预留的 PIP 用完了就无法再建立一个有效的互斥锁了
        return INVALID_HMUTEX;

    /* 优先级递减，所以一定确保预留的 PIP 够用
    OS_ENTER_CRITICAL();
    bTHMtxCurPIP = 1_bTHMtxCurPIP--;
    OS_EXIT_CRITICAL();

    return OSMutexCreate(bTHMtxCurPIP, &ubErr);
}

/* 线程同步区加锁
void os_thread_mutex_lock(HMUTEX hMutex)
{
    UCHAR ubErr;
    OSMutexPend(hMutex, 0, &ubErr);
}

/* 线程同步区解锁
void os_thread_mutex_unlock(HMUTEX hMutex)
{
    OSMutexPost(hMutex);
}

/* 删除线程同步锁，释放该资源
void os_thread_mutex_uninit(HMUTEX hMutex)
{
    UCHAR ubErr;
    OS_CPU_SR cpu_sr = 0;

    /* 回收占用的宝贵的 PIP 资源
    OS_ENTER_CRITICAL();
    1_bTHMtxCurPIP++;
    OS_EXIT_CRITICAL();

    OSMutexDel(hMutex, OS_DEL_ALWAYS, &ubErr);
}
```

接着就是信号量函数，rt-thread 下的实现代码如下：

```
/* 信号量初始化，参数 unInitVal 指定初始信号量值， unCount 指定信号量最大数值
HSEM os_thread_sem_init(UINT unInitVal, UINT unCount)
```

```
{
    unCount = unCount; /* 避免编译器警告

    HSEM hSem = rt_sem_create("rt-dsem", unInitVal, RT_IPC_FLAG_FIFO);
    if(RT_NULL != hSem)
        return hSem;

    return INVALID_HSEM; /* 初始失败要返回一个无效句柄
}

/* 等待信号量到达，参数 unWaitSecs 指定要等待的超时时间（单位为秒）：
/* 0，一直等下去直至信号量到达，收到信号则返回值为 0，出错则返回值为-1；
/* 其它，等待指定时间，如果指定时间内信号量到达，则返回值为 0，超时则返回值为 1，出错则返回值为-1
INT os_thread_sem_pend(HSEM hSem, INT nWaitSecs)
{
    if(nWaitSecs)
        nWaitSecs = nWaitSecs * RT_TICK_PER_SECOND;
    else
        nWaitSecs = RT_WAITING_FOREVER;

    rt_err_t lRtnVal = rt_sem_take(hSem, nWaitSecs);
    if(RT_EOK == lRtnVal)
        return 0;
    else if(-RT_ETIMEOUT == lRtnVal)
        return 1;
    else
        return -1;
}

/* 投递信号量
void os_thread_sem_post(HSEM hSem)
{
    rt_sem_release(hSem);
}

/* 信号量去初始化，释放该资源
void os_thread_sem_uninit(HSEM hSem)
{
    rt_sem_delete(hSem);
}
```

#### ucosii 下的信号量实现代码：

```
/* 信号量初始化，参数 unInitVal 指定初始信号量值， unCount 指定信号量最大数值
HSEM os_thread_sem_init(UINT unInitVal, UINT unCount)
{
    unCount = unCount; /* 避免编译器警告
    return OSSemCreate((USHORT)unInitVal);
```

```

}

/** 等待信号量到达，参数 unWaitSecs 指定要等待的超时时间（单位为秒）：
/** 0，一直等下去直至信号量到达，收到信号则返回值为 0，出错则返回值为-1；
/** 其它，等待指定时间，如果指定时间内信号量到达，则返回值为 0，超时则返回值为 1，出错则返回值为-1
INT os_thread_sem_pend(HSEM hSem, INT nWaitSecs)
{
    UCHAR ubErr;
    OSSemPend(hSem, (UINT)(nWaitSecs * OS_TICKS_PER_SEC), &ubErr);
    if(OS_ERR_NONE == ubErr)
        return 0;
    else if(OS_ERR_TIMEOUT == ubErr)
        return 1;
    else
        return -1;
}

/** 投递信号量
void os_thread_sem_post(HSEM hSem)
{
    OSSemPost(hSem);
}

/** 信号量去初始化，释放该资源
void os_thread_sem_uninit(HSEM hSem)
{
    UCHAR ubErr;
    OSSemDel(hSem, OS_DEL_ALWAYS, &ubErr);
}

```

接下来要实现的是一个重量级的函数 `os_thread_onpstack_start()`，用于协议栈内部工作线程的建立。对于以太网的应用场景来说，协议栈内部工作线程有两个，一个是 one-shot 类型的定时器计数线程；另外一个是在 tcp 底层发送线程。这两个线程已经添加到了 `os_adapter.c` 文件在头部位置声明的工作线程列表中。该列表是一个静态存储时期的本地变量：

```
lr_stcbaPStackThread
```

其类型就是咱们在前面讨论过并重新定义过的 `STCB_PSTACKTHREAD` 结构体。由于不同的 OS 下，结构体成员有所不同，因此添加到 `lr_stcbaPStackThread` 列表中的成员值需要单独定义。首先是 `rt-thread` 下的：

```

/** 协议栈内部工作线程列表
#define THOSTIMERCOUNT_PRIO      22      /** onps 栈 one-shot 定时器计数线程优先级
#define THOSTIMERCOUNT_STK_SIZE  256 * 4 /** 栈大小
#define THOSTIMERCOUNT_TIMESLICE 10      /** 分配给 one-shot 定时器计数线程的最长运行时间片

#if SUPPORT_SACK
    #define THTCPHANDLER_PRIO      21 /** onps 栈 tcp 协议底层发送线程
    #define THTCPHANDLER_STK_SIZE  256 * 4
    #define THTCPHANDLER_TIMESLICE 5

```



```

#endif

const static STCB_PSTACKTHREAD lr_stcbaPStackThread[] = {
    { thread_one_shot_timer_count, /* one-shot 定时器计数线程入口函数，协议栈内部工作线程
      (void *)0,
      "OSTimerCnt",
      THOSTIMERCOUNT_STK_SIZE,
      THOSTIMERCOUNT_PRIO,
      THOSTIMERCOUNT_TIMESLICE },

#ifdef SUPPORT_SACK
    { thread_tcp_handler, /* tcp 协议底层发送线程入口函数，协议栈内部工作线程
      (void *)0,
      "TcpHandler",
      THTCPHANDLER_STK_SIZE,
      THTCPHANDLER_PRIO,
      THTCPHANDLER_TIMESLICE },
#endif
};

```

上述定义，除了 STCB\_PSTACKTHREAD:: pfunThread、STCB\_PSTACKTHREAD:: pvParam 两个字段的值不能改变外，其它字段值可以根据自己的实际情况进行调整。ucosii 下的实现代码：

```

/* 协议栈内部工作线程列表
#define THOSTIMERCOUNT_PRIO    22  /* onps 栈定时器计数任务优先级
#define THOSTIMERCOUNT_STK_SIZE 256 /* 栈大小
__align(8) OS_STK THOSTIMERCOUNT_STK[THOSTIMERCOUNT_STK_SIZE];

#ifdef SUPPORT_SACK
    #define THTCPHANDLER_PRIO    21  /* onps 栈 tcp 协议底层发送任务优先级
    #define THTCPHANDLER_STK_SIZE 256
    __align(8) OS_STK THTCPHANDLER_STK[THTCPHANDLER_STK_SIZE];
#endif

const static STCB_PSTACKTHREAD lr_stcbaPStackThread[] = {
    { thread_one_shot_timer_count,
      (void *)0,
      (OS_STK *)&THOSTIMERCOUNT_STK[THOSTIMERCOUNT_STK_SIZE - 1],
      THOSTIMERCOUNT_PRIO },

#ifdef SUPPORT_SACK
    { thread_tcp_handler, (void *)0, (OS_STK *)&THTCPHANDLER_STK[THTCPHANDLER_STK_SIZE - 1], THTCPHANDLER_PRIO },
#endif
};

```

如果我们在 sys\_config.h 文件中置位 SUPPORT\_SACK 宏，tcp 底层发送线程就会被加入到协议栈。该线程的主要工作是从协议栈为每个 tcp 链路建立的发送缓存中顺序读取用户数据，封装成 tcp 报文后交给 ip 层发送出去，

完成 tcp sack（选择性确认）机制要求的所有工作。关于 tcp sack 请参阅《onps 栈移植手册》1.6 节，不再赘述。

接下来实现线程启动函数 `os_thread_onpstack_start()`。rt-thread 下：

```
/* 启动协议栈内部工作线程
void os_thread_onpstack_start(void *pvParam)
{
    pvParam = pvParam; /* 避免编译器警告

    /* 建立工作线程
    rt_thread_t tid;
    INT i;
    for (i = 0; i < sizeof(lr_stcbaPStackThread) / sizeof(STCB_PSTACKTHREAD); i++)
    {
        tid = rt_thread_create(lr_stcbaPStackThread[i].pszThreadName,
                               lr_stcbaPStackThread[i].pfunThread,
                               RT_NULL,
                               lr_stcbaPStackThread[i].unStackSize,
                               lr_stcbaPStackThread[i].ubPrio,
                               lr_stcbaPStackThread[i].unTimeSlice);

        if(tid != RT_NULL)
            rt_thread_startup(tid);
    }
}
```

ucosii 下：

```
/* 启动协议栈内部工作线程
void os_thread_onpstack_start(void *pvParam)
{
    pvParam = pvParam; /* 避免编译器警告

    /* 建立工作线程
    INT i;
    for (i = 0; i < (INT)(sizeof(lr_stcbaPStackThread) / sizeof(STCB_PSTACKTHREAD)); i++)
        OSTaskCreate(lr_stcbaPStackThread[i].pfunThread,
                     lr_stcbaPStackThread[i].pvParam,
                     lr_stcbaPStackThread[i].pstkTop,
                     lr_stcbaPStackThread[i].ubPrio);
}
```

至此，OS 适配层的移植工作完成。

### 3.1.5 编写网卡驱动并注册网卡

移植样例用的目标板是 STM32F407VET6 开发板，它提供一个 rj45 口。由于 F407VET6 这个 MCU 已经集成了一个以太网（ETH/MAC）外设，所以我们可以直接把 STM32 官方发布的 mac 驱动集成到我们的工程中（`emac.c`、`emac.h` 两个文件），

然后再根据协议栈的具体要求修改这个驱动即可完成这部分移植工作：

### 1) emacs.h

这个文件要完成的主要工作就是定义 MAC 地址、指定网卡名称、打开或关闭 dhcp 动态地址请求等这几项内容：

```
#ifndef EMAC_H
#define EMAC_H

#ifdef SYMBOL_GLOBALS
    #define EMAC_EXT
#else
    #define EMAC_EXT extern
#endif /* SYMBOL_GLOBALS

/** 如果你的以太网环境存在 dhcp 服务器，该宏置 1，以太网卡将自动从 dhcp 服务器获取 ip 地址、网关、dns 等配置信息
** 置 0，需要你把 ip 地址、网关、dns 等配置信息手动添加到驱动代码中
#define DHCP_REQ_ADDR_EN 1

#define NETIF_ETH_NAME      "eth0" /* ethernet 网卡名称，根据实际情形请自行修改
#define DP83848_PHY_ADDRESS 0x01   /* phy 地址，请根据目标板原理图修改该值

/** mac 地址定义，根据实际情形可自行修改
#define MAC_ADDR0_DEFAULT 0x4E
#define MAC_ADDR1_DEFAULT 0x65
#define MAC_ADDR2_DEFAULT 0x6F
#define MAC_ADDR3_DEFAULT 0x22
#define MAC_ADDR4_DEFAULT 0x06
#define MAC_ADDR5_DEFAULT 0x01

/** 外部引用的网卡驱动函数声明
EMAC_EXT BOOL  emac_init(void);
EMAC_EXT INT  emac_send(SHORT sBufListHead, UCHAR *pubErr);
EMAC_EXT void ETH_IRQHandler(void);
#endif
```

### 2) emacs.c

这个文件要完成的工作其实很复杂，好在官方提供的驱动代码已经替我们完成了绝大部分的工作，我们只需把网卡驱动融合进协议栈就可以了。首先我们把协议栈入口头文件 include 到 emacs.c 文件中：

```
#include "onps.h"
```

rt-thread 还需要把 os 的入口头文件 include 到 emacs.c 中：

```
#include <rtthread.h>
```

在 emacs.c 文件的头部声明一个只读的静态存储时期的私有变量，保存 emacs.h 头文件中指定的 mac 地址：

```
/** mac 地址
static const uint8_t lr_ubaMacAddr[ETH_MAC_ADDR_LEN] =
{ MAC_ADDR0_DEFAULT, MAC_ADDR1_DEFAULT, MAC_ADDR2_DEFAULT, MAC_ADDR3_DEFAULT, MAC_ADDR4_DEFAULT, MAC_ADDR5_DEFAULT };
```

头文件 `emac.h` 中声明的三个函数构成了网卡驱动的骨架：`emac_init()` 函数完成网卡的初始配置及注册功能；`emac_send()` 提供通讯报文的发送功能；`ETH_IRQHandler()` 则是网卡接收中断服务函数，用于读取到达的通讯报文。前文说过，这三个函数的基本处理逻辑来自于 STM32 官方发布的 bsp，我们只是在此基础上增加了与协议栈相关的处理代码。接下来移植工作的重点也是这部分新增加的代码，首先是 `emac_init()`：

```
/* 网卡初始化
BOOL emac_init(void)
{
    /* mac 引脚配置、使能 mac 时钟、配置 mac 等工作，可以照搬官方给的代码，然后根据你的需求调整相关配置位：
    /* 如果开启 ipv6 支持，就需要禁止网卡的组播帧过滤功能：
    stMac.ETH_MulticastFramesFilter = ETH_MulticastFramesFilter_None;
    /* 反之，则使能组播帧过滤以提升网络处理性能
    stMac.ETH_MulticastFramesFilter = ETH_MulticastFramesFilter_Perfect;
    /* 详细的实现代码可从 gitee 或 github 上拉取（参见文档结尾处）
    .....
    .....

    /* 等待完成 ethernet 网卡配置
    while(ETH_ERROR == ETH_Init(&stMac, DP83848_PHY_ADDRESS))
        os_sleep_ms(10); /* 在这里用上了前面已经移植好的 os 适配层函数：毫秒级延时函数
                           /* emac_init() 函数一定是在 os 已启动且协议栈加载完毕后再调用

    /* 使能接收中断
    ETH_DMAITConfig(ETH_DMA_IT_NIS | ETH_DMA_IT_R, ENABLE);
    /* 配置 mac 地址
    ETH_MACAddressConfig(ETH_MAC_Address0, lr_ubaMacAddr);
    /* 初始话发送、接收描述符链表：Chain Mode
    ETH_DMATxDscChainInit(DMATxDscTab, &Tx_Buff[0][0], ETH_TXBUFNB);
    ETH_DMARxDscChainInit(DMARxDscTab, &Rx_Buff[0][0], ETH_RXBUFNB);

    /* 发送报文的校验和插入操作旁路
    INT i;
    for(i=0; i<ETH_TXBUFNB; i++)
        ETH_DMATxDscChecksumInsertionConfig(&DMATxDscTab[i], ETH_DMATxDsc_ChecksumByPass);

    /* 关键的一步：启动以太网卡之前一定要先将其注册到协议栈 EN_ONPSERR enErr;
    ST_IPv4 stIPv4;
    #if !DHCP_REQ_ADDR_EN
        /* 分配一个静态地址，请根据自己的具体网络情形设置地址
        stIPv4.unAddr = inet_addr_small("192.168.0.4");
        stIPv4.unSubnetMask = inet_addr_small("255.255.255.0");
        stIPv4.unGateway = inet_addr_small("192.168.0.1");
        stIPv4.unPrimaryDNS = inet_addr_small("1.2.4.8");
        stIPv4.unSecondaryDNS = inet_addr_small("8.8.8.8");
        stIPv4.unBroadcast = inet_addr_small("192.168.0.255");
    #else
        /* 地址清零，为 dhcp 客户端申请动态地址做好准备
        memset(&stIPv4, 0, sizeof(stIPv4));
```

```
#endif

/* 注册网卡，也就是将网卡添加到协议栈
l_pstNetifEth = ethernet_add(NETIF_ETH_NAME,
                             lr_ubaMacAddr,
                             &stIPv4,
                             emac_send,
                             start_thread_ethernet_ii_recv,
                             &l_pstNetifEth,
                             &enErr);

if(!l_pstNetifEth)
{
#if SUPPORT_PRINTF
    printf("ethernet_add() failed, %s\r\n", onps_error(enErr));
#endif
    return FALSE;
}

/* 使能 mac 及 dma 接收与发送，其实就是启动网卡，开始工作
ETH_Start();

#if DHCP_REQ_ADDR_EN
/* 启动一个 dhcp 客户端，从 dhcp 服务器申请一个动态地址
if(dhcp_req_addr(l_pstNetifEth, &enErr))
{
#if SUPPORT_PRINTF
    printf("dhcp request ip address successfully.\r\n");
#endif
}
else
{
#if SUPPORT_PRINTF
    printf("dhcp request ip address failed, %s\r\n", onps_error(enErr));
#endif
}
#endif

return TRUE;
}
```

上面给出的蓝色代码就是与协议栈相关的代码，主要是完成了两块工作：一，注册网卡到协议栈；二，申请分配一个动态地址，如果需要的话。注册网卡的工作是由协议栈提供的 `ethernet_add()` 函数完成的，其详细说明如下：

#### 功能

注册 ethernet 网卡到协议栈，只有如此协议栈才能正常使用该网卡进行数据通讯。

#### 函数原型

PST\_NETIF ethernet\_add(const CHAR \*pszIfName,

```

        const UCHAR ubaMacAddr[ETH_MAC_ADDR_LEN],
        PST_IPV4 pstIPv4,
        PFUN_ETHMAC_SEND pfunEmacSend,
        void (*pfunStartTHEmacRecv)(void *pvParam),
        PST_NETIF *ppstNetif,
        EN_ONPSERR *penErr
    );

```

### 参数

pszIfName: 网卡名称

ubaMacAddr: 网卡 mac 地址

pstIPv4: 指向 ST\_IPV4 结构体的指针, 这个结构体保存用户指定的 ip 地址、网关、dns、子网掩码等配置信息

pfunEmacSend: 函数指针, 指向发送函数, 函数原型为 `INT(* PFUN_ETHMAC_SEND)(SHORT sBufListHead, UCHAR *pubErr)`, 这个指针指向的其实就是我们接下来就要讲到的 `emac_send()` 函数

pfunStartTHEmacRecv: 函数指针, 协议栈使用该函数启动网卡接收线程

ppstNetif: 二维指针, 协议栈成功注册网卡后 `ethernet_add()` 函数会返回一个 `PST_NETIF` 指针给调用者, 这个参数指向这个指针, 其最终会被协议栈通过 `pvParam` 参数传递给 `pfunStartTHEmacRecv` 指向的函数

penErr: 如果注册失败, `ethernet_add()` 函数会返回一个错误码, 这个参数用于接收这个错误码

### 返回值

注册成功, 返回一个 `PST_NETIF` 类型的指针, 后续的报文收发均用到这个指针; 注册失败则返回 `NULL`。具体错误信息参见 `penErr` 参数携带的错误码。

`emac_init()` 函数的实现代码中, 我们调用 `ethernet_add()` 完成网卡注册时有两个地方需要特别说明: 一个是 `l_pstNetifEth`; 另一个是 `start_thread_ethernet_ii_recv()`。前一个用于接收注册成功后返回的 `PST_NETIF` 指针; 后一个则是需要提供一个线程启动函数, 启动协议栈内部的以太网接收线程。

`PST_NETIF` 指针非常重要, 它是网卡能够正常工作的关键。报文收发均用到这个指针。它的生命周期应该与协议栈的生命周期相同。因此这个指针变量应该是一个静态存储时期的变量, 并且网卡的接收、发送函数均能访问:

```
static PST_NETIF l_pstNetifEth = NULL; /* 协议栈返回的 netif 结构
```

对于 `start_thread_ethernet_ii_recv()` 函数, 我们需要根据不同的 `rtos` 分别编写不同的代码实现。首先是 `rt-thread`:

```

#define THETHIIRECV_PRIO      21      /* ethernet 网卡接收线程（任务）优先级
#define THETHIIRECV_STK_SIZE  384 * 4 /* 接收线程栈大小, 这个栈要相对大一些, 太小会报错
#define THETHIIRECV_TIMESLICE 10      /* 单次任务调度线程能够工作的最大时间片
static void start_thread_ethernet_ii_recv(void *pvParam)
{
    rt_thread_t tid = rt_thread_create("EthRcv",
                                        thread_ethernet_ii_recv,
                                        pvParam,
                                        THETHIIRECV_STK_SIZE,
                                        THETHIIRECV_PRIO,
                                        THETHIIRECV_TIMESLICE);

    if(tid != RT_NULL)
        rt_thread_startup(tid);
}

```

}

其中 `thread_ethernet_ii_recv()` 函数完成实际的以太网层的报文接收及处理工作。这是一个线程入口函数，由协议栈提供。其轮询等待网卡接收中断 `ETH_IRQHandler()` 函数发送的报文到达信号，收到信号则立即读取并处理到达的报文。

ucosii 下的实现代码：

```
#define THETHIIRECV_PRIO      21  /* ethernet 网卡接收线程（任务）优先级
#define THETHIIRECV_STK_SIZE 384 /* 接收线程栈大小
__align(8) OS_STK THETHIIRECV_STK[THETHIIRECV_STK_SIZE];
static void start_thread_ethernet_ii_recv(void *pvParam)
{
    OSTaskCreate(thread_ethernet_ii_recv,
                pvParam,
                (OS_STK *)&THETHIIRECV_STK[THETHIIRECV_STK_SIZE - 1],
                THETHIIRECV_PRIO);
}
```

`emac_init()` 函数中还有个地方需要特别交待下——网卡组播过滤功能的选择。IPv6 地址配置需要用到组播通讯，所以网卡驱动必须要禁止组播过滤，否则无法完成无状态 ipv6 地址配置。反之，当我们不需要协议栈支持 ipv6 时，就可以使能组播过滤，抛弃组播报文，减少通讯负载，提升网络处理性能。

由于协议栈采用缓存链表机制向下层协议传递数据，最后传递给网卡发送函数 `emac_send()`。因此我们需要从缓存链表读取数据然后再发送到网络上（无论是否置位 `SUPPORT_SACK` 宏，上层协议最终都会以缓存链表的形式将数据传递给网卡驱动）。另外，发送函数的原型定义还要符合协议栈的要求，因为我们在进行网卡注册时还要向协议栈注册发送函数的入口地址（其原型定义 `PFUN_EMAC_SEND` 请参阅前文）。满足上述要求的 `emac_send()` 函数的实现代码如下：

```
/* 完成实际的报文发送工作
INT emac_send(SHORT sBufListHead, UCHAR *pubErr)
{
    SHORT sNextNode = sBufListHead;
    UCHAR *pubData;
    USHORT usDataLen;
    UCHAR *pubDMATxBuf = (UCHAR *)DMATxDescToSet->Buffer1Addr;
    __IO ETH_DMADESCTypeDef *DMATxNextDesc = DMATxDescToSet;
    UINT unHasCpyBytes = 0, unRemainBytes, unCpyBytes;

    /* 首先看看报文长度是否超出了 DMA 发送缓冲区的限制
    /* buf_list_get_len() 函数由协议栈提供，其遍历整个 buf list 链表计算整个 ethernet 报文的长度
    UINT unEthPacketLen = buf_list_get_len(sBufListHead);
    if(unEthPacketLen > ETH_TX_BUF_SIZE * ETH_TXBUFNB)
    {
        /* 太长了，无法发送超长报文，直接报错返回
        if(pubErr)
            *((EN_ONPSERR *)pubErr) = ERRPACKETTOOLARGE;
        return -1;
    }
```

```
/* 遍历所有节点并将它们逐个放入 DMA 缓冲区
__lblGetNextNode:
pubData = (UCHAR *)buf_list_get_next_node(&sNextNode, &usDataLen); /* 获取下一个节点
if (NULL == pubData) /* 返回空意味着已经取出完毕, 现在可以启动 DMA 发送了
{
    /* 官方驱动代码
    ETH_Prepate_Transmit_Descriptors((USHORT)unEthPacketLen);
    return (INT)unEthPacketLen;
}

unRemainBytes = (UINT)usDataLen; /* 当前节点携带的数据长度

/* 将当前节点携带的数据复制到 dma 发送缓冲区
__lblCpy:
/* 当前 DMA 缓冲区链表节点是否已满, 如果已满则移动到下一个节点继续接收
if(unHasCpyBytes >= ETH_TX_BUF_SIZE)
{
    /* 指向环形链表的下一个节点, 不用考虑节点是否为空的问题, 因为这个链表是一个全封闭的环形链表
    DMATxNextDesc = (ETH_DMADESCTypeDef *) (DMATxNextDesc->Buffer2NextDescAddr);
    pubDMATxBuf   = (UCHAR *)DMATxNextDesc->Buffer1Addr;
    unHasCpyBytes = 0;
}

/* 计算当前 dma 缓冲区节点还能够 copy 的数据长度
unCpyBytes = unRemainBytes < (ETH_TX_BUF_SIZE - unHasCpyBytes) ? unRemainBytes : (ETH_TX_BUF_SIZE - unHasCpyBytes);

/* 复制到 dma 缓冲区
memcpy(pubDMATxBuf + unHasCpyBytes, pubData + (usDataLen - unRemainBytes), unCpyBytes);
unRemainBytes -= unCpyBytes;
unHasCpyBytes += unCpyBytes;

/* 剩余字节数为 0, 则读取下一个 buf list 节点
if(!unRemainBytes)
    goto __lblGetNextNode;
else
    goto __lblCpy;
}
```

与 `emac_init()` 函数正好相反, 上面给出的蓝色代码为官方驱动代码, 其它则是与协议栈相关的代码。

关于缓存链表, 其实现机制其实很简单。以 udp 通讯为例, 用户要发送数据到对端, 会直接调用 udp 发送函数, 将数据传递给 udp 层。udp 层收到用户数据后, 为了节省内存, 避免复制, 协议栈直接将用户数据挂接到缓存链表上成为链表的数据节点。接着, udp 层会再申请一个节点把 udp 报文头挂接到数据节点的前面, 组成一个拥有两个节点的完整 udp 报文链表——链表第一个节点挂载 udp 报文头, 第二个节点挂载用户要发送的数据。至此, udp 层的报文封装工作完成, 数据继续向 ip 层传递。ip 层会继续申请一个节点把 ip 报文头挂接到 udp 报文头节点的前面, 组成一个拥有三个节点的完整 ip 报文链表。以此类推直至送达 `emac_send()` 函数, 完成最终的发送。`emac_send()` 函数的工作就是按照上述机制顺序取出链表节点携带的报文数据, 复制到网卡的 dma 发送缓冲区, 然后启动 dma 完成发送。



移植拼图的最后一块为网卡接收中断服务子函数 ETH\_IRQHandler()：

```
void ETH_IRQHandler(void)
{
    __IO ETH_DMADESCTypeDef *DMARxNextDesc;
    UCHAR *pubPacket;
    EN_ONPSERR enErr;

    /** 进入中断，前者为 rt-thread 下的，注释掉的是 ucossii 下的，请根据目标系统自行选择使用哪个函数
    rt_interrupt_enter()/* OSIntEnter() */;
    {
        /** 处理所有到达的报文
        while(ETH_CheckFrameReceived())
        {
            /** 获取到达的一帧完整的 ethernet ii 协议报文
            FrameTypeDef stRcvdFrame = ETH_Get_Received_Frame();

            /** 利用协议栈 buddy 模块动态申请一块内存用于保存到达的报文
            pubPacket = (UCHAR *)buddy_alloc(sizeof(ST_SLINKEDLIST_NODE) + stRcvdFrame.length, &enErr);
            if(pubPacket)
            {
                /** 根据协议栈要求，刚才申请的内存按照 PST_SLINKEDLIST_NODE 链表节点方式组织刚刚收到的报文
                PST_SLINKEDLIST_NODE pstNode = (PST_SLINKEDLIST_NODE)pubPacket;
                pstNode->uniData.unVal = stRcvdFrame.length;
                memcpy(pubPacket + sizeof(ST_SLINKEDLIST_NODE), (UCHAR *)stRcvdFrame.buffer, stRcvdFrame.length);

                /** 将上面组织好的报文节点放入接收链表，这个接收链表由协议栈管理，ethernet_put_packet() 函数由协议栈提供
                /** thread_ethernet_ii_rcv() 接收线程负责等待 ethernet_put_packet() 函数投递的信号并读取这个链表
                ethernet_put_packet(1_pstNetifEth, pstNode);
            }
            else
            {
                #if SUPPORT_PRINTF && DEBUG_LEVEL
                printf("<EIRQ> %s\r\n", onps_error(enErr));
                #endif
            }

            /** Release descriptors to DMA
            /** Check if frame with multiple DMA buffer segments
            if (DMA_RX_FRAME_infos->Seg_Count > 1)
                DMARxNextDesc = DMA_RX_FRAME_infos->FS_Rx_Desc;
            else
                DMARxNextDesc = stRcvdFrame.descriptor;

            /** Set Own bit in Rx descriptors: gives the buffers back to DMA
            for (int i=0; i<DMA_RX_FRAME_infos->Seg_Count; i++)
            {
                DMARxNextDesc->Status = ETH_DMARxDesc_OWN;
```

```

    DMARxNextDesc = (ETH_DMADESCTypeDef *) (DMARxNextDesc->Buffer2NextDescAddr);
}

/* Clear Segment_Count
DMA_RX_FRAME_infos->Seg_Count = 0;

/* When Rx Buffer unavailable flag is set: clear it and resume reception
if ((ETH->DMASR & ETH_DMASR_RBUS) != (u32)RESET)
{
    /* Clear RBUS ETHERNET DMA flag
    ETH->DMASR = ETH_DMASR_RBUS;
    /* Resume DMA reception
    ETH->DMARPDR = 0;
}
}

/* Clear the Eth DMA Rx IT pending bits
ETH_DMAClearITPendingBit(ETH_DMA_IT_R);
ETH_DMAClearITPendingBit(ETH_DMA_IT_NIS);
}
rt_interrupt_leave() /* OSIntExit() */; /* 离开中断,同样前者为 rt-thread 下的, 后者为 ucousii 下的
}

```

蓝色代码部分为协议栈相关的代码，其余均为官方驱动代码。

其中 `buddy_alloc()` 函数在功能上与 c 语言的标准库函数 `malloc()` 完全相同，都是动态分配一块指定大小的内存给调用者使用，使用完毕后再由用户通过 `buddy_free()` 函数释放。这两个函数由协议栈的内存管理（mmu）模块提供。

`ethernet_put_packet()` 函数需要重点解释一下。这个函数由协议栈提供。它完成的工作非常重要，它在网卡接收中断 `ETH_IRQHandler()` 与协议栈以太网接收线程 `thread_ethernet_ii_recv()` 之间搭建了一个数据流通的“桥”。接收中断收到报文后按照协议栈要求，将报文封装成 `ST_SLINKEDLIST_NODE` 类型的链表节点，然后传递给 `ethernet_put_packet()` 函数。该函数将立即把传递过来的节点挂载到由协议栈管理的以太网接收链表的尾部，然后投递一个“有新报文到达”的信号量。前面咱们在 `emac_init()` 函数中启动的以太网接收线程 `thread_ethernet_ii_recv()` 会轮询等待这个信号量。一旦信号到达，接收线程将立即读取链表并取出报文交给协议栈处理。至此，移植工作完成。

在编写协议栈测试代码之前，针对 `rt-thread` 还有几项特殊的配置工作要做，否则协议栈无法正常工作。打开 `rtconfig.h` 文件，修改其中的几项配置：

```

// <o>Alignment size for CPU architecture data access
// <i>Default: 4
#define RT_ALIGN_SIZE 8 /* 4 改为 8，协议栈用到了 printf() 函数，这个函数要求栈必须是 8 字节对齐（参见 ucousii），否
                        /* 则系统会触发 HardFault_Handler 异常，当然你的系统如果不支持 printf()，此处无须调整

// <o>the max length of object name<2-16>
// <i>Default: 8
#define RT_NAME_MAX 16 /* 缺省值 8 个字节有点短，协议栈内部工作线程名称超过 8 个，所以这里调整到 16 个字节

// <o>the stack size of main thread<1-4086>

```

```
// <i>Default: 512
#define RT_MAIN_THREAD_STACK_SIZE 428 * 4 /* 主线程堆栈相对大一些，后面的 tcp 服务器测试用例用到的栈空间较大

/* 使能 rt-thread 堆，开启 rt-thread 对信号量、互斥锁的支持，同时简化线程操作（不再单独申请数组建立栈空间）
#define RT_USING_SEMAPHORE
#define RT_USING_MUTEX
#define RT_USING_HEAP

/* 关闭 rt-thread 的 console 口，直接注释掉即可
// #define RT_USING_CONSOLE
```

上述配置宏最重要的就是 RT\_ALIGN\_SIZE。这个宏很关键，其用于指定线程栈的对齐方式：4 字节还是 8 字节对齐。正常情况下，如果我们不使用特殊的 c 库函数比如 printf()，缺省的 4 字节对齐方式就够用了，反之则必须指定 8 字节对齐。

### 3.1.6 dhcp 动态地址申请测试

测试之前一定要确保你的网络中存在一个 dhcp 服务器，能够接受 dhcp 客户端的动态地址分配申请。然后，在 main.c 文件中先把协议栈提供的统一入口头文件及网卡驱动头文件添加进来：

```
#include "onps.h"
#include "emac.h"
```

接着再把协议栈及网卡驱动加载代码添加到该文件中。注意，rt-thread 与 ucspi 下的添加位置是不同的。首先是 rt-thread 下的：

```
int main(void)
{
    EN_ONPSERR enErr;

    /* printf、rtc 等底层硬件的初始化相关工作
    .....

    /* 加载协议栈
    if(open_npstack_load(&enErr))
    {
#if SUPPORT_PRINTF
        printf("The open source network protocol stack is loaded successfully. \r\n");
#endif

        /* 加载网卡驱动：网卡初始化、注册网卡、启动 dhcp 客户端动态申请 ip 地址
        if(emac_init())
            goto __lblStart;
    }
    else
    {
#if SUPPORT_PRINTF
        printf("The open source network protocol stack failed to load, %s\r\n", onps_error(enErr));
#endif
    }
```

```

}

/* 协议栈加载失败则不再继续正常的工作流程，在这里死循环即可
while(TRUE)
{
    /* 可以在这里增加一个错误指示灯之类的相关代码
    .....

    rt_thread_mdelay(1000);
}
__blStart:
/* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
.....

/* 进入主线程的主逻辑处理循环
while(TRUE)
{
    /* 保留你原先的代码即可，当前测试这里不需要添加任何代码
    .....
}
}

```

ucosii 下的处理代码与上面完全相同，但添加位置不是在 main() 函数中。我们需要把上述代码添加到由 mian() 函数启动的主任务中，由主任务完成协议栈及网卡驱动的加载工作：

```

int main(void)
{
    .....
    OSInit();
    /* 协议栈及网卡驱动的加载工作由 THMain 任务完成
    OSTaskCreate(THMain, (void *)0, (OS_STK *)&THMAIN_STK[THMAIN_STK_SIZE - 1], THMAIN_PRIO);
    OSStart();
}

/* 主任务
static void THMain(void *pvData)
{
    .....

    /* 参照 rt-thread 下的逻辑添加顺序将相同的代码添加在此即可
    .....
}

```

回到先前编辑过的 emac.h 文件中，把 DHCP\_REQ\_ADDR\_EN 宏打开：

```
#define DHCP_REQ_ADDR_EN 1 /* 置 1，打开 dhcp 动态 ip 地址申请
```

sys\_config.h 文件中，把调试级别调整到 2，如此才能在控制台看到动态地址申请的详细信息：

```
#define DEBUG_LEVEL 2
```

编译并下载到你的目标板上，如果你的板子支持 printf 功能，稍等几秒（动态地址申请需要点时间），你将在控制台看到如下输出：

```
The open source network protocol stack is loaded successfully.
<eth0> added to the protocol stack
[inet 0.0.0.0, netmask 0.0.0.0, broadcast 0.0.0.0, Primary DNS 0.0.0.0, Secondary DNS 0.0.0.0]
Add/Update network interface <eth0> to routing table
[destination default, gateway 192.168.0.1, genmask 255.255.255.0]
    ip addr 192.168.0.3
    sub net mask 255.255.255.0
    gateway 192.168.0.1
    primary dns srv 1.2.4.8
    secondary dns srv 114.114.114.114
    lease time 60
dhcp request ip address successfully.
```

为了测试 dhcp 客户端的稳定性，我们可以把地址租期调整为 1 分钟（注意是在 dhcp 服务器端调整租期），而不是缺省的 24 小时。这样协议栈将每隔 30 秒发送一次续租请求（为什么是 30 秒请参考 dhcp 相关 rfc 文档）。让其长时间运行，测试 dhcp 客户端续租功能的稳定性。无论续租成功还是失败，控制台均会输出一条日志信息：

```
The open source network protocol stack is loaded successfully.
<eth0> added to the protocol stack
[inet 0.0.0.0, netmask 0.0.0.0, broadcast 0.0.0.0, Primary DNS 0.0.0.0, Secondary DNS 0.0.0.0]
Add/Update network interface <eth0> to routing table
[destination default, gateway 192.168.0.1, genmask 255.255.255.0]
    ip addr 192.168.0.3
    sub net mask 255.255.255.0
    gateway 192.168.0.1
    primary dns srv 1.2.4.8
    secondary dns srv 114.114.114.114
    lease time 60
dhcp request ip address successfully.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
```

从实际测试情况看，连续测试 3 天以上，续租功能非常稳定，未出现续租未成功的情形。当然，测试样例所在的网络环境，dhcp 服务器的负担并不是很重，仅管理着几台 dhcp 客户端。不过，不用担心，当续租未成功时，比如地址已被占用或者 dhcp 服务器因未知原因拒绝续租等，协议栈会重新申请地址。这一切都是自动进行的，用户无须干预。此种情形下，一定要确保你的应用场景不会因 ip 地址的改变而受到影响。否则请关闭 dhcp 客户端，采用静态地址。

### 3.1.7 ipv6 地址配置测试

由于 ppp 接口能否实现 ipv6 通讯的关键并不是由协议栈配置决定的而是由当地移动运营商决定的。所以，协议栈当前暂不支持 ppp 链路上的 ipv6 通讯，详细说明请参阅《onps 栈移植手册》1.6 节。另外，鉴于 ipv6 地址配置的复杂性，协议栈目前仅支持状态和无状态地址自动配置，不支持静态配置。因此，此项测试工作必须是在以太网环境下进行，且网络中的路由器支持 ipv6。如果路由器还支持 dhcpv6 地址分配，我们还可以测试有状态地址配置，反之仅能够测试无状态地址配置。

首先 sys\_config.h 文件置位 SUPPORT\_IPV6 宏，然后把 ipv6 相关的文件添加到工程中：

```
ethernet/dhcpv6.c(.h) ethernet/dhcpv6_frame.h ip/ipv6_onfigure.c(.h) ip/icmpv6.c(.h) ip/icmpv6_frame.h
```

编译、下载、重启系统，一切顺利的话，稍等片刻，控制台输出如下：

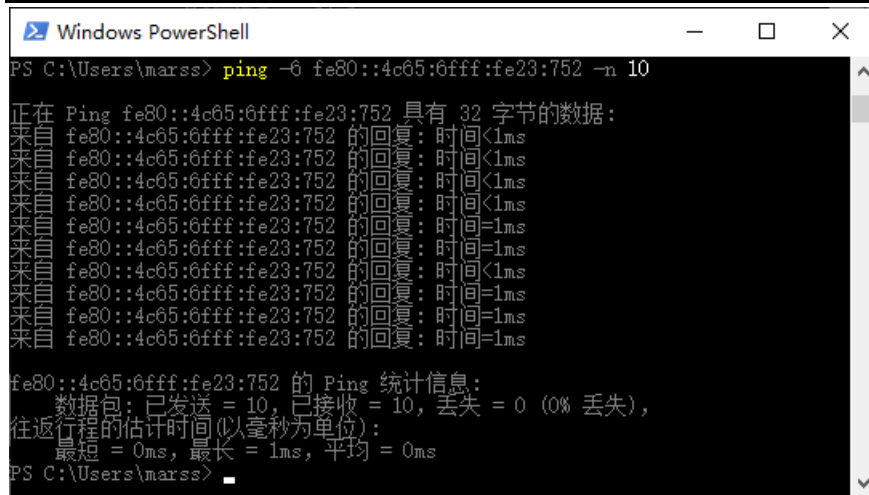
共得到三个地址：本地链路地址、无状态配置地址、DHCPv6 分配的地址（有状态配置）。除本地链路地址外，其余两个地址均存在生存计时（Valid lifetime）。协议栈根据计时值确定地址状态：选用（Preferred）、弃用（Deprecated）、无效（Invalid）。选用和弃用状态的地址可以继续使用，无效状态地址将被放弃，重新配置。地址生存计时时长等于“选用”时长加“弃用”时长，先“选用”后“弃用”，顺序计时，最后计时归零，地址无效。具体详情请参阅相关资料，不赘言。

### 3.1.8 ping 测试

首先测试 ping 开发板，这一步什么都不用做。先是 ping ipv4 地址，如下图所示：

ping ipv6 地址：





```
Windows PowerShell
PS C:\Users\marss> ping -6 fe80::4c65:6fff:fe23:752 -n 10

正在 Ping fe80::4c65:6fff:fe23:752 具有 32 字节的数据:
来自 fe80::4c65:6fff:fe23:752 的回复: 时间<1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间<1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间<1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间<1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间=1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间=1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间<1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间=1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间=1ms
来自 fe80::4c65:6fff:fe23:752 的回复: 时间=1ms

fe80::4c65:6fff:fe23:752 的 Ping 统计信息:
    数据包: 已发送 = 10, 已接收 = 10, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 0ms, 最长 = 1ms, 平均 = 0ms
PS C:\Users\marss>
```

无论是 ipv4 还是 ipv6 地址，均没有丢包，响应时间维持在 1 毫秒以内。

接下来测试开发板 ping 外部 ip 地址。首先使能协议栈对 ping 测试工具的支持，sys\_config.h 文件中：

```
#define NETTOOLS_PING 1 /* 使能 ping 测试
```

添加测试代码到 main.c 文件中：

```
#include "net_tools/ping.h" /* ping 工具头文件

/* 回调函数，收到目标地址的应答报文后 ping 工具会调用这个函数完成用户的特定处理逻辑
/* 针对这个测试，在这里就是简单地打印出了应答报文的内容以及 ping 的响应时间
static void ping_recv_handler(USHORT usIdentifier, /* ping 的标识 id，响应报文与探测报文这个 id 应该一致
                               void *pvFromAddr, /* 响应报文的源地址
                               USHORT usSeqNum, /* 响应报文序号，其与探测报文一致
                               UCHAR *pubEchoData, /* 响应报文携带的响应数据，其与探测报文一致
                               UCHAR ubEchoDataLen, /* 响应报文携带的数据长度
                               UCHAR ubTTL, /* ttl 值
                               UCHAR ubElapsedMSecs, /* 响应时长，单位：秒，从发送探测报文开始计时到收到响应报文结束计时
                               void *pvParam)
{
    CHAR szSrcAddr[20];
    struct in_addr stInAddr;
    stInAddr.s_addr = *(UINT *)pvFromAddr;
#if SUPPORT_PRINTF
#if PRINTF_THREAD_MUTEX
    os_thread_mutex_lock(o_hMtxPrintf);
#endif
    printf("<Fr>%s, recv %d bytes, ID=%d, Sequence=%d, Data='%s', TTL=%d, time=%dms\r\n",
           inet_ntoa_safe(stInAddr, szSrcAddr), /* 这是一个线程安全的 ip 地址转 ascii 字符串函数
           (UINT)ubEchoDataLen,
           usIdentifier,
           usSeqNum,
           pubEchoData,
           (UINT)ubTTL,
           (UINT)ubElapsedMSecs);
```

```
#if PRINTF_THREAD_MUTEX
    os_thread_mutex_unlock(o_hMtxPrintf);
#endif
#endif
}
```

开启 ping 测试的代码需要根据不同的 OS 添加到不同位置：rt-thread 下依然是添加到 main() 函数中，ucosii 依然是添加到 THMain 主任务。

```
_lblStart:
/* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
.....

/* 启动 ping 测试
USHORT usSeqNum = 0;
UINT unErrCount = 0;
INT nPing = ping_start(AF_INET, &enErr);
if(nPing < 0)
{
    /* 启动失败，输出一条日志信息
    printf("ping_start() failed, %s\r\n", onps_error(enErr));
}

/* 进入主线程的主逻辑处理循环
while(TRUE)
{
    /* 你原先的处理逻辑代码
    .....
    if(nPing >= 0)
    {
        /* ping 目标地址
        INT nRtnVal = ping(nPing, "192.168.0.2", usSeqNum++, 64, GetElapsedMSecs, ping_recv_handler, NULL, 3, NULL, &enErr);
        if(nRtnVal <= 0) /* ping 返回一个错误
        {
            /* 累计 ping 错误数
            unErrCount++;

            /* 控制台打印当前错误数
            os_thread_mutex_lock(o_hMtxPrintf);
            printf("no reply received, the current number of errors is %d, current error: %s\r\n",
                unErrCount,
                nRtnVal ? onps_error(enErr) : "recv timeout");
            os_thread_mutex_unlock(o_hMtxPrintf);
        }
    }
}
```



蓝色字体部分为新添加的测试代码。`ping_start()` 函数的调用非常简单，其功能就是开启 ping 测试。结束 ping 测试需要调用 `ping_end()` 函数，否则 ping 测试会一直占用协议栈资源。这几个函数的使用说明如下：

#### 函数原型

```
INT ping_start(INT family, EN_ONPSERR *penErr);
```

#### 功能

启动 ping 测试。注意，启动后你可以随时更换目标地址，不必拘泥于一个固定的目标地址。

#### 参数

family: 地址族类型，IPv4 地址其值为 `AF_INET`，IPv6 地址为 `AF_INET6`

penErr: 如果启动失败，该参数用于接收具体的错误码

#### 返回值

成功，返回当前启动的 ping 测试任务的句柄；失败，返回值小于 0，具体错误信息参看 `penErr` 保存的错误码。

—

#### 函数原型

```
void ping_end(INT nPing);
```

#### 功能

结束 ping 测试，释放占用的协议栈资源。

#### 参数

nPing: `ping_start()` 函数返回的 ping 测试句柄

#### 返回值

无

—

#### 函数原型

```
INT ping(INT nPing,
         const CHAR *pszDstAddr,
         USHORT usSeqNum,
         UCHAR ubTTL,
         UINT(*pfunGetCurMsecs)(void),
         PFUN_PINGRCVHANDLER pfunRcvHandler,
         PFUN_PINGERRHANDLER pfunErrHandler,
         UCHAR ubWaitSecs,
         void *pvParam,
         EN_ONPSERR *penErr);
```

#### 功能

ping 目标地址并等待接收对端的响应报文。其功能与通用的 ping 测试工具完全相同。

#### 参数



编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(I) 帮助(H)

开发板->pc 响应时间: 9604-9435=169微妙

pc->开发板 响应时间: 5162-4521=641微妙

Time	Source	Destination	Protocol	Length	Info
248 113.367631	192.168.0.3	192.168.0.2	ICMP	74	Echo (ping) reply id=0x0001, seq=59/15104, ttl=64
249 114.159435	192.168.0.3	192.168.0.2	ICMP	78	Echo (ping) request id=0x0002, seq=55/14080, ttl=64
250 114.159604	192.168.0.2	192.168.0.3	ICMP	78	Echo (ping) reply id=0x0002, seq=55/14080, ttl=64
251 114.384521	192.168.0.2	192.168.0.3	ICMP	74	Echo (ping) request id=0x0001, seq=60/15360, ttl=64
252 114.385162	192.168.0.3	192.168.0.2	ICMP	74	Echo (ping) reply id=0x0001, seq=60/15360, ttl=64
253 115.171070	192.168.0.3	192.168.0.2	ICMP	78	Echo (ping) request id=0x0002, seq=56/14336, ttl=64
254 115.171328	192.168.0.2	192.168.0.3	ICMP	78	Echo (ping) reply id=0x0002, seq=56/14336, ttl=64
255 115.407382	192.168.0.2	192.168.0.3	ICMP	74	Echo (ping) request id=0x0001, seq=61/15616, ttl=64
256 115.408496	192.168.0.3	192.168.0.2	ICMP	74	Echo (ping) reply id=0x0001, seq=61/15616, ttl=64
257 116.182809	192.168.0.3	192.168.0.2	ICMP	78	Echo (ping) request id=0x0002, seq=57/14592, ttl=64

开发板 ping 外部 ipv6 地址的处理逻辑与 ipv4 完全相同, 测试代码仅有微小区别。首先是用户自定义的接收回调函数: ipv6 下的 echo 响应报文没有 ttl 值; 地址转换函数改为 inet6\_ntoa(), 具体代码参见样例工程, 不再赘述。另外一处区别就是 ping\_start() 函数选择 AF\_INET6 地址族。仅此两处, 再无其它。直接 ping 目标主机的 ipv6 地址, 结果如下:

需要特别说明的是，开发板 ping 全球单播地址（2 开头，即首字节最高三位为 001）时，主机如果是 windows 系统，公用网络防火墙要设置策略允许开发板 ipv6 地址 ping 主机，否则无法 ping 通。当然最简单的做法是 ping 期间关闭公用网络防火墙。

### 3.1.9 dns 测试

这个测试需要为网卡设定好能够访问互联网的网关、DNS 服务器地址等配置信息。当然如果采用 dhcp 动态地址申请的方式能够得到这些信息那就更省事了。首先，sys config.h 文件中使能 dns 查询：

```
#define NETTOOLS_DNS_CLIENT 1 /* 使能 dns 查询客户端
```

依然是 main.c 文件，先把 dns 查询工具的头文件 include 进来：

```
#include "net_tools/dns.h"
```

然后 rt-thread 下 main() 函数, uc0s11 下 THMain 主任务中添加 dns 查询测试代码:

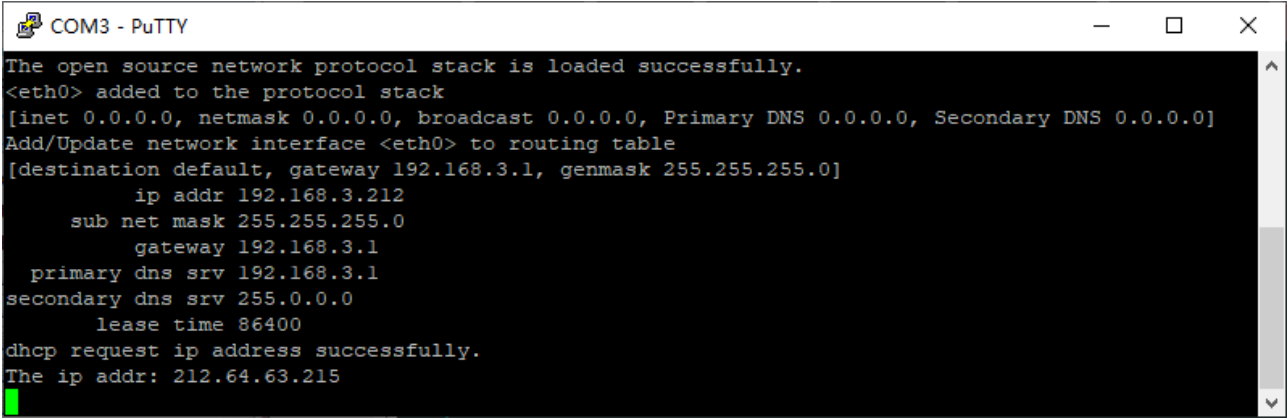
```
__lblStart:
/* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
.....

/* dns 查询测试
in_addr_t unPrimaryDNS, unSecondaryDNS;
INT nDnsClient = dns_client_start(&unPrimaryDNS, &unSecondaryDNS, 3, &enErr);
if(nDnsClient < 0)
{
    /* dns 客户端启动失败, 输出一条错误日志
    printf("%s\r\n", onps_error(enErr));
}
else
{
    /* 发送查询请求并等待 dns 服务器的应答
    in_addr_t unIp = dns_client_query(nDnsClient, unPrimaryDNS, unSecondaryDNS, "gitee.com", &enErr);
    if(unIp) /* 查询成功
    {
        CHAR szAddr[20];
        printf("The ip addr: %s\r\n", inet_ntoa_safe_ext(unIp, szAddr));
    }
    else
        printf("%s\r\n", onps_error(enErr)); /* 查询失败

    /* 结束 dns 查询, 释放占用的协议栈资源
    dns_client_end(nDnsClient);
}

/* 进入主线程的主逻辑处理循环
while(TRUE)
{
    /* 你原先的处理逻辑代码
    .....
}
```

蓝色字体部分为新添加的 dns 查询代码。编译并下载到开发板。为了省事, 我直接用的是 dhcp 从宽带路由器申请的动态地址, 这样就不用单独设置网关和 dns 服务器了。开发板完成 dns 查询后的控制台输出日志如下:



从日志可以看出，我的查询报文是发到路由器的，因为网关地址和 dns 服务器地址相同。一般家庭网关都是如此配置。这其实与直接发到实际的 dns 服务器并没什么不同。我们发送的 dns 查询请求到达家庭网关后，网关会替我们完成实际的查询操作，然后把查询结果反馈给我们。从底层协议看，网关反馈的查询结果与直接向 DNS 服务器查询得到结果报文完全一致。协议栈并不关心响应报文是从哪里发出的，只要能够在规定时间内等到响应报文并且报文结构符合协议标准即可。

与前面给出的 ping 测试工具差不多，dns 查询工具同样提供了一组简单的 api 函数用于实现域名查询。这一组函数包括 dns\_client\_start()、dns\_client\_end() 以及 dns\_client\_query()。其详细的使用说明如下：

函数原型

INT dns\_client\_start(in\_addr\_t \*punPrimaryDNS, in\_addr\_t \*punSecondaryDNS, CHAR bRcvTimeout, EN\_ONPSERR \*penErr);

功能

启动一个域名查询客户端。

参数

punPrimaryDNS: 指针类型，用于接收主域名服务器地址。协议栈会选择缺省路由绑定的网卡，并得到网卡携带的 dns 服务器地址用于接下来的域名查询，该参数即用于保存主 dns 服务器的地址

punSecondaryDNS: 指针类型，与上同，用于接收次域名服务器地址

bRcvTimeout: 查询超时时间，单位：秒

penErr: 如果启动失败，该参数用于接收具体的错误码

返回值

成功，返回当前启动的 dns 客户端的句柄；失败，返回值小于 0，具体错误信息参看 pennErr 保存的错误码。

—

函数原型

void dns\_client\_end(INT nClient);

功能

结束 dns 客户端，释放占用的协议栈资源。

参数

nClient: dns\_client\_start() 函数返回的 dns 客户端句柄

返回值



无

—

### 函数原型

```
in_addr_t dns_client_query(INT nClient,
                           in_addr_t unPrimaryDNS,
                           in_addr_t unSecondaryDNS,
                           const CHAR *pszDomainName,
                           EN_ONPSERR *penErr);
```

### 功能

发送 dns 查询请求，并等待服务器的响应报文，功能与通用的 dns 客户端完全相同。

### 参数

nClient: dns\_client\_start() 函数返回的 dns 客户端句柄  
 punPrimaryDNS: 主域名服务器地址，其值为 dns\_client\_start() 函数得到的主域名服务器地址  
 punSecondaryDNS: 次域名服务器地址，其值为 dns\_client\_start() 函数得到的次域名服务器地址  
 pszDomainName: 指针类型，指向要查询的域名  
 penErr: 如果查询失败，该参数用于接收具体的错误码

### 返回值

成功，返回域名对应的 ip 地址；失败，返回值为 0，具体错误信息参看 penErr 保存的错误码。

## 3.1.10 sntp 网络校时测试

要进行这个测试依然要确保你的开发板在物理层能够访问互联网，同时你的开发板支持 rtc，并提供一组 rtc 操作函数，包括读取、设置系统当前时间等 api。首先，sys\_config.h 文件中使能 sntp 网络校时：

```
#define NETTOOLS_Sntp 1 /* 使能 sntp 客户端
```

依然是 main.c 文件，先把 sntp 网络校时工具的头文件 include 进来：

```
#include "net_tools/sntp.h"
```

然后 rt-thread 下 main() 函数，ucosii 下 THMain 主任务中添加 sntp 网络校时测试代码：

```
__lblStart:
/** 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
.....

/** 先设定个不合理的时间，以测试网络校时功能是否正常,由 rtc 驱动提供，负责修改系统当前时间
RTCSetsysTime(22, 9, 5, 17, 42, 30);

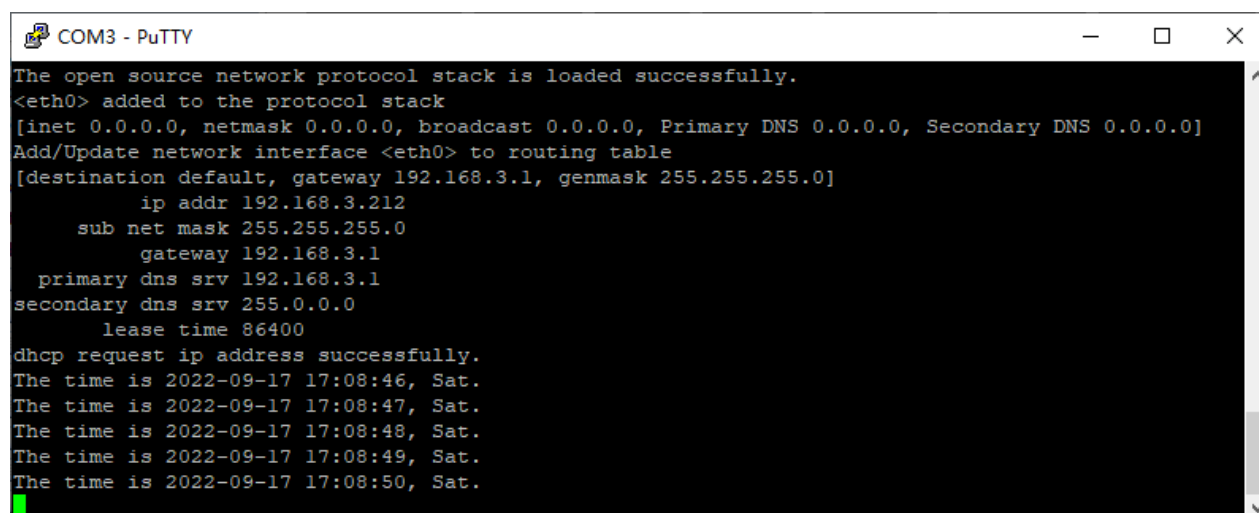
/** 开启网络校时，sntp_update_by_ip() 与 sntp_update_by_dns() 均可使用
ST_DATETIME stDateTime;
char szWeekDay[8];
//if(sntp_update_by_ip("52.231.114.183", NULL, RTCSetSystemUnixTimestamp, 8, &enErr)) /** ntp 服务器地址直接校时
if(sntp_update_by_dns("time.windows.com", Time, RTCSetSystemUnixTimestamp, 8, &enErr)) /** ntp 服务器域名方式校时
{
```

```
/* 获取系统时间，检查校时结果
RTCGetSysTime(&stDateTime);
/* 将数字表示的周几转为英文简写
GetSystemWeekEngShort(stDateTime.ubWeekDay, szWeekDay);

/* 控制台输出当前系统时间
printf("The time is %d-%02d-%02d %02d:%02d:%02d, %s\r\n",
    stDateTime.usYear,
    stDateTime.ubMonth,
    stDateTime.ubDay,
    stDateTime.ubHour,
    stDateTime.ubMin,
    stDateTime.ubSec,
    szWeekDay);
}
else
{
    //printf("sntp_update_by_ip() failed, %s\r\n", onps_error(enErr));
    printf("sntp_update_by_dns() failed, %s\r\n", onps_error(enErr));
}

/* 进入主线程的主逻辑处理循环
while(TRUE)
{
    /* 你原先的处理逻辑代码
    .....
}
```

依然是蓝色字体部分为 sntp 网络校时测试代码。测试结果如下：



我们把时间设定在了 2022 年 9 月 5 日 17 点 42 分 30 秒，经过网络校时后，时间被准确校正到了当前时间。测试代码用到了 rtc 模块提供的一组 api。其中 RTCSetSysTime() 用于设置系统时间。RTCSetSystemUnixTimestamp() 函数同样也是设置系统时间，只不过是通过 unix 时间戳进行设置。RTCGetSysTime() 函数用于读取当前系统时间。这组函数的具体实现代码可以从文末给出的 gitee 地址上拉取（[移植样例工程](#)）。相较于 ping 及 dns 工具，sntp 网络校时工具只提供了一

个接口函数 `sntp_update_by_xx()` 即可完成校时。我们可以通过 `ntp` 服务器地址也可以通过 `ntp` 服务器域名进行校时。函数的详细使用说明如下：

#### 函数原型

```
BOOL sntp_update_by_ip(const CHAR *pszNtpSrvIp,
                      time_t(*pfunTime)(void),
                      void(*pfunSetSysTime)(time_t),
                      CHAR bTimeZone,
                      EN_ONPSERR *penErr);
```

#### 功能

发送一个校时请求到 `pszNtpSrvIp` 参数指定的 `ntp` 服务器，并等待服务器的响应报文，完成校时操作。

#### 参数

`pszNtpSrvIp`: `ntp` 服务器 ip 地址  
`pfunTime`: 函数指针，与 c 库函数 `time()` 功能及原型相同，返回自 1970 年 1 月 1 日 0 时 0 分 0 秒以来经过的秒数，可以为空  
`pfunSetSysTime`: 函数指针，通过 unix 时间戳设置系统当前时间，由 `sntp_update_by_xx()` 内部调用，收到正确的响应报文后调用该函数设置系统时间  
`bTimeZone`: 时区，例如东 8 区，其值为 8；西 8 区其值为 -8  
`penErr`: 如果校时失败，该参数用于接收具体的错误码

#### 返回值

校时成功，返回 `TRUE`；失败，返回 `FALSE`，具体错误信息参看 `penErr` 保存的错误码。

`sntp_update_by_dns()` 函数与 `sntp_update_by_ip()` 函数除了第一个入口参数变成了域名外，其它完全相同，不再赘述。

## 3.1.11 tcp 客户端测试

在协议栈源码工程下，存在一个用 `vs2015` 建立的 `TcpServerForStackTesting` 工程。其运行在 `windows` 平台下，模拟实际应用场景下的 `tcp` 服务器。当 `tcp` 客户端连接到服务器后，服务器会立即下发一个 1100 多字节长度的控制报文到客户端。之后在整个 `tcp` 链路存续期间，服务器会每隔一段随机的时间（90 秒到 120 秒之间）下发控制报文到客户端，模拟实际应用场景下服务器主动下发指令、数据到客户端的情形。客户端则连续上发数据报文到服务器，服务器回馈一个应答报文给客户端。客户端如果收不到该应答报文则会立即重发，直至收到应答报文或超过重试次数后重连服务器。总之，整个测试场景的设计目标就是完全契合常见的商业应用需求，以此来验证协议栈的核心功能指标是否完全达标。

用 `vs2015` 打开这个工程，配置管理器指定目标平台为 `x64`。`main.cpp` 及 `tcp_helper.h` 文件的头部定义了服务器的地址类型、端口号以及报文长度等信息：

```
#define SUPPORT_IPV6      1    /* 使能或禁止 tcp 服务器是否支持 ipv6 地址（tcp_helper.h 文件头部） */
#define SRV_PORT          6410 /* 服务器端口 */
#define LISTEN_NUM        10   /* 最大监听数 */
#define RCV_BUF_SIZE      2048 /* 接收缓冲区容量 */
#define PKT_DATA_LEN_MAX 1200 /* 报文携带的数据最大长度，凡是超过这个长度的报文都将被丢弃 */
```

如果我们想测试 `ipv6` 通讯，将 `SUPPORT_IPV6` 宏置位即可。客户端将使用 `ipv6` 地址连接服务器。我们可以依据实际情形调整上述配置并利用这个模拟服务器测试 `tcp` 客户端的通讯功能。

鉴于我们只是为了测试协议栈 `tcp` 通讯的可靠性，服务器无须支持过多的 `tcp` 并发连接，所以服务器采用了简单的 `select` 模型而非复杂的 `iocp` 模型设计实现。Build 整个工程，得到测试服务器进程后回到我们的移植测试工程。依然



是 main.c 文件，主线程（任务）下启动一个新的线程（任务）用于 tcp 客户端测试。rt-thread 下线程启动代码如下：

```
.....

#define THTCPCLT_PRIO      26      /* tcp 客户端测试线程优先级
#define THTCPCLT_STK_SIZE  384 * 4 /* tcp 客户端测试线程栈大小
#define THTCPCLT_TIMESLICE 10      /* 单次调度 tcp 客户端测试线程能够运行的最长时间片

.....

int main(void)
{
    .....

    rt_thread_t tid;

    .....

    __lblStart:
    /* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
    .....

    /* 启动 tcp 客户端测试任务
    tid = rt_thread_create("THTcpClnt", THTcpClnt, RT_NULL, THTCPCLT_STK_SIZE, THTCPCLT_PRIO, THTCPCLT_TIMESLICE);
    if(tid != RT_NULL)
        rt_thread_startup(tid);
    .....
}
```

ucos ii 下线程启动代码添加到 THMain 主任务中：

```
.....

#define THTCPCLT_PRIO      32 /* tcp 客户端测试任务优先级
#define THTCPCLT_STK_SIZE  384 /* tcp 客户端测试任务栈大小
__align(8) OS_STK THTCPCLT_STK[THTCPCLT_STK_SIZE]; /* tcp 客户端测试任务栈
.....

static void THMain(void *pvData)
{
    .....

    __lblStart:
    /* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
    .....

    /* 启动 tcp 客户端测试任务
    OSTaskCreate(THTcpClnt, (void *)0, (OS_STK *)&THTCPCLT_STK[THTCPCLT_STK_SIZE - 1], THTCPCLT_PRIO);
    .....
}
```

接下来搭建主测试框架，也就是 THTcpClnt 线程（任务）。为了突出如何利用协议栈提供的 socket api 编写 tcp 客户端这条主线，接下来仅给出通讯相关的代码，非关键的业务逻辑及容错相关的代码这里不再给出。具体的代码实现还是直接拉取 gitee 上的样例工程（参见[第 6 节](#)）：

```
/* 提前申请一块静态存储时期的缓冲区用于 tcp 客户端的接收和发送，因为接收和发送的报文都比较大，所以不使用动态申请的方式
#define RCV_BUF_SIZE      1300 /* 接收缓冲区容量
#define PKT_DATA_LEN_MAX 1200 /* 报文携带的数据最大长度，凡是超过这个长度的报文都将被丢弃
static UCHAR l_ubaRcvBuf[RCV_BUF_SIZE]; /* 接收缓冲区
```

```
static UCHAR l_ubaSndBuf[sizeof(ST_COMMUPKT_HDR)+PKT_DATA_LEN_MAX]; /* 发送缓冲区，ST_COMMUPKT_HDR 为通讯报文头部结构体

/* tcp 客户端测试线程（任务），其连接 tcp 测试服务器，与服务器进行双向通讯
static void THTcpClt(void *pvUserParam)
{
    EN_ONPSERR enErr;
    SOCKET hSocket = INVALID_SOCKET;
    CHAR bIsConnected = FALSE; /* 连接成功标识

    /* 主循环进行以下几项工作：1）上传数据；2）等待应答；3）接收控制指令；4）回馈控制指令应答报文
    while(TRUE)
    {
        /* 尚未分配一个 socket，先申请一个 socket
        if(INVALID_SOCKET == hSocket)
        {
            bIsConnected = FALSE; /* 重新分配 socket 意味着当前 tcp 链路已经断开了，连接成功标识置 FALSE

            /* 分配一个 socket
            hSocket = socket(AF_INET, SOCK_STREAM, 0, &enErr);

            /* 返回了一个无效的 socket
            if(INVALID_SOCKET == hSocket)
            {
                printf("<1>socket() failed, %s\r\n", onps\_error(enErr));
                continue;
            }
        }

        /* 如果尚未连接服务器则先连接服务器
        if(!bIsConnected)
        {
            /* 连接成功则 connect() 函数返回 0，非 0 值则连接失败
            if(!connect(hSocket, "192.168.0.2", 6410, 10))
            {
                /* 等待接收服务器应答或控制报文的时长（即 recv() 函数的等待时长），单位：秒。0 不等待；大于 0 等待指定秒数；-1 一直
                /* 等待直至数据到达或报错。设置成功返回 TRUE，否则返回 FALSE。这里我们设置 recv() 函数不等待
                /* 注意，只有连接成功后才可设置这个接收等待时长
                if(!socket\_set\_rcv\_timeout(hSocket, 0, &enErr))
                    printf("socket_set_rcv_timeout() failed, %s\r\n", onps\_error(enErr));

                /* 更新连接成功标识
                bIsConnected = TRUE;
            }
            else /* 连接失败
                printf("connect 192.168.0.2:6410 failed, %s\r\n", onps\_get\_last\_error(hSocket, NULL));
        }
    }
}
```

```

/* 接收, 前面已经设置 recv() 函数不等待, 有数据则读取数据后立即返回, 无数据则立即返回
INT nRcvBytes = recv(hSocket, l_ubaRcvBuf, sizeof(l_ubaRcvBuf));
if(nRcvBytes > 0)
{
    /* 收到报文, 处理之, 报文有两种: 一种是应答报文; 另一种是服务器主动下发的控制报文
    .....
}

/* 发送数据报文到服务器
/* 首先封装要发送的数据报文, PST_COMMUPKT_HDR 其类型为指向 ST_COMMUPKT_HDR 结构体的指针, 这个结构体是我们自定义的
/* 通讯报文头部结构, 该结构体的具体实现细节参看下文
PST_COMMUPKT_HDR pstHdr = (PST_COMMUPKT_HDR)l_ubaSndBuf;
.....

/* 发送上面已经封装好的数据报文
INT nPacketLen = sizeof(ST_COMMUPKT_HDR) + pstHdr->usDataLen + 1; /* 要发送的数据报文长度, 参考样例工程源码
INT nSndBytes = send(hSocket, l_ubaSndBuf, nPacketLen, 3);
if(nSndBytes != nPacketLen) /* 与实际要发送的数据不相等的话就意味着发送失败了
{
    printf("<err>sent %d bytes failed, %s\r\n", nPacketLen, onps\_get\_last\_error(hSocket, &enErr));

    /* 关闭 socket, 断开当前 tcp 连接, 释放占用的协议栈资源
    close(hSocket);
    hSocket = INVALID_SOCKET;
}
}
}

```

蓝色字体部分即为协议栈提供的编写 tcp 客户端用到的 socket api。红色字体注释部分则为编写 tcp 客户端的几个关键步骤:

- 1) 调用 socket 函数, 申请一个数据流(tcp)类型的 socket;
- 2) connect() 函数建立 tcp 连接;
- 3) recv() 函数等待接收服务器下发的应答及控制报文;
- 4) send() 函数将封装好的数据报文发送给服务器;
- 5) close() 函数关闭 socket, 断开当前 tcp 连接;

与传统的 socket 编程相比, 除了上述几个函数的原型与 Berkeley sockets 标准有细微的差别, 在功能及使用方式上没有任何改变。之所以对函数原型进行调整, 原因是传统的 socket 编程模型比较繁琐——特别是阻塞/非阻塞的设计很不简洁, 需要一些看起来很“突兀”地额外编码, 比如 select 操作。在设计协议栈的 socket 模型时, 考虑到类似 select 之类的操作细节完全可以借助 rtos 的信号量机制将其封装到底层实现, 从而达成简化用户编码, 让 socket 编程更加简洁、优雅的目的。因此, 最终呈现给用户的协议栈 socket 模型部分偏离了 Berkeley 标准。

前面说过, 我们设计的这个 tcp 客户端用于真实模拟实际应用场景下的 tcp 通讯, 所以 tcp 通讯报文也是按照真实场景设计的。通讯报文有头部结构, 有数据, 有校验, 还有尾部标志, 也有专门的应答报文设计。通讯报文头部结构的原型如下:

```

#define PKT_FLAG 0xEE /* 通讯报文的头部和尾部标志
typedef struct _ST_COMMUPKT_HDR_ { /* 数据及控制指令报文头部结构
    CHAR bFlag;           /* 报文头部标志, 其值参看 PKT_FLAG 宏

```

```

CHAR bCmd;          /* 指令, 0 为数据报文, 1 为控制指令报文
CHAR bLinkId;       /* tcp 链路标识, 当存在多个 tcp 链路时, 该字段用于标识这是哪一个链路
UINT unSeqNum;      /* 报文序号
UINT unTimestamp;   /* 报文被发送时刻的 unix 时间戳
USHORT usDataLen;   /* 携带的数据长度
USHORT usChecksum;  /* 校验和 (crc16), 覆盖除头部和尾部标志字符串之外的所有字段
} PACKED ST_COMMUPKT_HDR, *PST_COMMUPKT_HDR;

typedef struct _ST_COMMUPKT_ACK_ { /* 数据即控制指令应答报文结构
    ST_COMMUPKT_HDR stHdr; /* 报文头
    UINT unTimestamp;      /* unix 时间戳, 其值为被应答报文携带的时间戳
    CHAR bLinkId;          /* tcp 链路标识, 其值为被应答报文携带的链路标识
    CHAR bTail;            /* 报文尾部标志, 其值参看 PKT_FLAG 宏
} PACKED ST_COMMUPKT_ACK, *PST_COMMUPKT_ACK;

```

真实场景下, 单个 tcp 报文携带的数据长度的上限基本在 1K 左右, 所以在我们的测试中, 单个通讯报文的长度也设定在这个范围内。开发板循环上报服务器的数据报文的长度 900 多字节, 服务器下发开发板的控制报文长度 1100 多字节。

pc 上开启 tcp 测试服务器, 把我们的 tcp 测试代码更新到开发板上, 我们会分别在服务器控制台和开发板控制台看到如下输出信息:

```

E:\work\Neo-T\ONPStackVSPProj\Release\TcpServerForStackTesting.exe
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212092, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212093, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212094, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212095, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212096, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212097, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212098, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212099, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212100, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212101, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212102, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212103, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212104, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212105, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212106, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212107, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212108, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212109, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212110, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212111, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212112, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212113, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212114, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212115, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212116, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212117, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212118, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212119, the data length is 900 bytes
1#2022-09-20 10:52:34#>recved the uploaded packet, cmd = 0x00, SeqNum = 212120, the data length is 900 bytes

```

服务器侧控制台输出



```
COM3 - PuTTY
The open source network protocol stack is loaded successfully.
<eth0> added to the protocol stack
[inet 0.0.0.0, netmask 0.0.0.0, broadcast 0.0.0.0, Primary DNS 0.0.0.0, Secondary DNS 0.0.0.0]
Add/Update network interface <eth0> to routing table
[destination default, gateway 192.168.0.1, genmask 255.255.255.0]
    ip addr 192.168.0.3
    sub net mask 255.255.255.0
    gateway 192.168.0.1
    primary dns srv 1.2.4.8
    secondary dns srv 114.114.114.114
    lease time 60
dhcp request ip address successfully.
#l#>connect 192.168.0.2:6410 successfully!
l#2022-09-20 10:47:48#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 1, data length is 1128 bytes
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
l#2022-09-20 10:49:19#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 1, data length is 1128 bytes
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
<0> The ip address 192.168.0.3 of the NIC eth0 has been successfully renewed, The lease period is 60 seconds.
```

开发板侧控制台输出

开发板一旦连接到服务器，服务器会在第一时间（也就是服务器在收到 SYN ACK 报文后）下发一个携带 1128 字节长数据的控制报文给开发板，开发板也会同时开启连续上报数据报文到服务器的任务。双方同时进行，不给对方任何反应时间，以此来测试开发板 tcp 栈的性能。同时为了测试 tcp 栈极限情况下的处理能力，开发板会在收到服务器的应答报文后立即发送下一组数据报文，让协议栈连续工作。实际测试情况看，在开发板未开启看门狗的情况下，无论是把服务器部署在局域网内还是部署在互联网上，连续测试 5 天以上均未出现通讯中断或者开发板宕机的情形。

我们还可以把测试服务器改为 ipv6 地址类型。目标板置位 SUPPORT\_IPV6，然后把 socket 地址类型改为 AF\_INET6，测试一下 ipv6 地址族的通讯可靠性：

```
COM3 - PuTTY
The open source network protocol stack (ver 1.1.0.230726) is loaded successfully.
<eth0> added to the protocol stack
[inet 0.0.0.0, netmask 0.0.0.0, broadcast 0.0.0.0, Primary DNS 0.0.0.0, Secondary DNS 0.0.0.0]
Add/Update network interface <eth0> to routing table
[destination default, gateway 192.168.3.1, genmask 255.255.255.0]
    ip addr 192.168.3.76
    sub net mask 255.255.255.0
    gateway 192.168.3.1
    primary dns srv 192.168.3.1
    secondary dns srv 0.0.0.0
    lease time 86400
dhcp request ip address successfully.
#0#>connect [2408:8215:41a:f9d8:b7c3:216d:d435:6clf]:6410 failed, Addressing failure, default route does not exist
#1#>connect 47.92.239.107:6410 successfully!
l#2023-08-04 16:58:51#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 49, data length is 1128 bytes
Successfully configured address 2408:8215:41a:f9d8:4c65:6fff:fe23:753 based on the prefix advertised by server fe80::d6a1:48ff:fe96:6b3b
#0#>connect [2408:8215:41a:f9d8:b7c3:216d:d435:6clf]:6410 failed, tcp connection timeout

Configuration results of ipv6 for Ethernet adapter eth0:
    Link-local address: fe80::4c65:6fff:fe23:752
    configured address: 2408:8215:41a:f9d8:4c65:6fff:fe23:753 (Preferred, Valid lifetime 7200)
    configured address: 2408:8215:41a:f9d0:d4a1:4896:6b3b:2 (Preferred, DHCPv6, Valid lifetime 7209)
    default router<0>: fe80::d6a1:48ff:fe96:6b3b (High, Lifetime 1752)
    dns srv<0>: fe80::d6a1:48ff:fe96:6b3b
#0#>connect [2408:8215:41a:f9d8:b7c3:216d:d435:6clf]:6410 successfully!
0#2023-08-04 16:59:23#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 0, data length is 1128 bytes
1#2023-08-04 17:00:29#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 49, data length is 1128 bytes
0#2023-08-04 17:00:59#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 0, data length is 1128 bytes
1#2023-08-04 17:02:04#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 49, data length is 1128 bytes
0#2023-08-04 17:02:31#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 0, data length is 1128 bytes
1#2023-08-04 17:03:47#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 49, data length is 1128 bytes
0#2023-08-04 17:04:06#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 0, data length is 1128 bytes
1#2023-08-04 17:05:20#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 49, data length is 1128 bytes
```

样例工程实际测试了两路 tcp 客户端，一路 ipv4 地址族，另外一路 ipv6，上面给出的控制台输出可以很清晰地看到这一点。实际测试结果看，通讯可靠稳定。下图记录了协议栈连续工作几天时间上传一亿两千多万个报文的情况（每 1000 个报文输出一次，每个报文携带 900 字节的用户数据），期间未出现任何通讯故障：

```

tcpserverforstacktesting
1#2023-07-24 20:17:30#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125511000, 900 bytes
1#2023-07-24 20:17:31#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125512000, 900 bytes
1#2023-07-24 20:17:32#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125513000, 900 bytes
1#2023-07-24 20:17:33#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125514000, 900 bytes
1#2023-07-24 20:17:34#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125515000, 900 bytes
1#2023-07-24 20:17:35#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125516000, 900 bytes
1#2023-07-24 20:17:36#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125517000, 900 bytes
1#2023-07-24 20:17:37#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125518000, 900 bytes
1#2023-07-24 20:17:38#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125519000, 900 bytes
1#2023-07-24 20:17:39#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125520000, 900 bytes
1#2023-07-24 20:17:40#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125521000, 900 bytes
1#2023-07-24 20:17:41#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125522000, 900 bytes
1#2023-07-24 20:17:42#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125523000, 900 bytes
1#2023-07-24 20:17:43#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125524000, 900 bytes
1#2023-07-24 20:17:44#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125525000, 900 bytes
1#2023-07-24 20:17:45#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125526000, 900 bytes
1#2023-07-24 20:17:45#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125527000, 900 bytes
1#2023-07-24 20:17:46#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125528000, 900 bytes
1#2023-07-24 20:17:47#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125529000, 900 bytes
1#2023-07-24 20:17:48#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125530000, 900 bytes
1#2023-07-24 20:17:49#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125531000, 900 bytes
1#2023-07-24 20:17:50#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125532000, 900 bytes
1#2023-07-24 20:17:51#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125533000, 900 bytes
1#2023-07-24 20:17:52#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125534000, 900 bytes
1#2023-07-24 20:17:53#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125535000, 900 bytes
1#2023-07-24 20:17:54#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125536000, 900 bytes
1#2023-07-24 20:17:55#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125537000, 900 bytes
1#2023-07-24 20:17:56#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125538000, 900 bytes
1#2023-07-24 20:17:57#>Uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 125539000, 900 bytes

```

tcp 客户端测试代码用到的一组 socket api 的使用说明参见本文[第 4 节——socket 使用说明](#)，不再详述。在这里只须单独说明一下 send() 函数，因为这个函数很特殊，涉及 tcp 栈的底层实现。由于我们的栈是运行在资源受限的单片系统中，所以在设计 tcp 通讯时，我们在最初版本中先实现了核心通讯功能，主要目的就是为了节省内存。前面我们在介绍协议栈采用写时零复制技术时也说过这个原因。虽然我们不支持 tcp 层的重传，但并不代表 tcp 传输不再可靠。我们把 tcp 层的重传机制移到了用户层，由用户来实现重传。在 tcp 层，协议栈采用了相对低效的问答式发送机制。一问一答，确保报文到达对端才允许用户发送下一组，否则会通知用户发送失败。我们用部分性能的牺牲换取了传输得可靠性。由于我们采用的是写时零复制技术，所以问答式发送机制的最终结果会实时反映到用户层。这是因为要想避免数据复制，就必须确保用户层数据的生存周期等于协议栈的问答周期。数据维持到问答周期结束，用户才可以选择继续发送下一组数据还是重发数据（重传机制上移到用户层）。也就是说当用户调用 send() 函数发送数据后，协议栈 tcp 层会确保收到对端的 tcp ack 报文后才会通知用户层的 send() 函数返回实际发送的字节数，否则 send() 函数会一直等待（阻塞模式）或返回 0（非阻塞模式，代表发送中，尚未收到对端确认到达的报文），或者返回 -1 告知用户发送失败（超时或其它系统错误）。换句话说，send() 函数的返回值实时反映了当前数据发送的状态。用户层可以在第一时间知道当前发送的报文是否已实际到达对端。而不是像传统的 socket 编程那样，send() 函数返回实际发送的字节数仅代表数据已经被送达协议栈的发送缓冲区，但实际是否已被发送或者是否发送成功，通过 send() 函数是无法得知的。所以，对于我们刚刚编写的 tcp 测试客户端来说，我们完全可以不必等待服务器下发的应答报文，而是仅仅判定 send() 函数返回的字节数等于实际报文长度即可——因为这代表着数据已被成功送达服务器。这反而还省事了。实际应用场景下，我们完全可以利用 send() 函数的这个特点去掉用户层的应答环节，只发送数据到服务器，无须服务器应答，从而节省数据应答带来的额外带宽及性能开销。

实现基本通讯功能的 tcp 可以满足问答式通讯应用场景，但那些诸如 ftp 等需要大块数据传输的场景 tcp 栈的传输性能就会严重不足。所以我们在 1.1 版本提供了一个具备完整能力的 tcp，节省内存的同时力求为用户提供更多的选择。如果目标系统内存相对富裕，比如 80~128K 字节以上，系统配置文件置位 SUPPORT\_SACK 宏，我们将获得完整能力的 tcp 栈。协议栈会为每一个 tcp 链路建立一个发送缓存，以实现乱序、丢失包的选择性重传。同时，协议栈采用延迟应答机制进一步提升通讯载荷效率比。有关 SUPPORT\_SACK 宏以及延迟应答机制的相关说明请参阅《onps 栈移植手册》1.6 节。此时的 send() 函数仅负责将用户数据送达 tcp 缓存，不再等待对端应答，其返回值为缓存区接收的实际数据长度。此时，用户层代码在调用 send() 函数时要依据返回值来确定用户数据是否已全部送达 tcp 缓存。如果返回值小于实际要发送的数据长度，则意味着缓存已满，需要稍等片刻后再次调用 send() 函数继续发送，如下所示：



```

.....

/* 发送上面已经封装好的数据报文
INT nPacketLen = sizeof(ST_COMMUPKT_HDR) + pstHdr->usDataLen + 1; /* 要发送的数据报文长度，参考样例工程源码
INT nSndBytes, nHasSendBytes = 0;
while(nHasSendBytes < nPacketLen)
{
    nSndBytes = send(hSocket, &l_ubaSndBuf[nHasSendBytes], nPacketLen - nHasSendBytes, 0);
    if(nSndBytes < 0)
    {
        printf("<err>sent %d bytes failed, %s\r\n", nPacketLen, onps\_get\_last\_error(hSocket, &nErr));

        /* 关闭 socket，断开当前 tcp 连接，释放占用的协议栈资源
        close(hSocket);
        hSocket = INVALID_SOCKET;
    }
    else
        nHasSendBytes += nSndBytes;
}
.....

```

我们可以对比下 SUPPORT\_SACK 宏置位与复位时的通讯速度区别。先是 SUPPORT\_SACK 宏未置位，采用问答式通讯时：

```

管理员: C:\Windows\system32\cmd.exe - TcpServerForStackTesting.exe
0#2023-08-04 22:12:45#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 17000, 900 bytes
0#2023-08-04 22:13:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 18000, 900 bytes
0#2023-08-04 22:13:25#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 19000, 900 bytes
0#2023-08-04 22:13:43#>sent control command to peer, cmd = 0x01, ClientID = 1, the data length is 1128 bytes
0#2023-08-04 22:13:42#>Acknowledge packet, AackedLinkId = 0, ClientID = 1, AackedTimestamp <2023-08-04 22:13:43>
0#2023-08-04 22:13:45#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 20000, 900 bytes
0#2023-08-04 22:14:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 21000, 900 bytes
0#2023-08-04 22:14:25#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 22000, 900 bytes
0#2023-08-04 22:14:45#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 23000, 900 bytes
0#2023-08-04 22:15:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 24000, 900 bytes
0#2023-08-04 22:15:19#>sent control command to peer, cmd = 0x01, ClientID = 1, the data length is 1128 bytes
0#2023-08-04 22:15:18#>Acknowledge packet, AackedLinkId = 0, ClientID = 1, AackedTimestamp <2023-08-04 22:15:19>
0#2023-08-04 22:15:25#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 25000, 900 bytes
0#2023-08-04 22:15:45#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 26000, 900 bytes
0#2023-08-04 22:16:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 27000, 900 bytes
0#2023-08-04 22:16:25#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 28000, 900 bytes
0#2023-08-04 22:16:45#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 29000, 900 bytes
0#2023-08-04 22:16:53#>sent control command to peer, cmd = 0x01, ClientID = 1, the data length is 1128 bytes
0#2023-08-04 22:16:52#>Acknowledge packet, AackedLinkId = 0, ClientID = 1, AackedTimestamp <2023-08-04 22:16:53>
0#2023-08-04 22:17:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 30000, 900 bytes
0#2023-08-04 22:17:25#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 31000, 900 bytes
0#2023-08-04 22:17:45#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 32000, 900 bytes
0#2023-08-04 22:18:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 33000, 900 bytes
0#2023-08-04 22:18:24#>sent control command to peer, cmd = 0x01, ClientID = 1, the data length is 1128 bytes
0#2023-08-04 22:18:23#>Acknowledge packet, AackedLinkId = 0, ClientID = 1, AackedTimestamp <2023-08-04 22:18:24>
0#2023-08-04 22:18:25#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 34000, 900 bytes
0#2023-08-04 22:18:45#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 35000, 900 bytes
0#2023-08-04 22:19:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 36000, 900 bytes
0#2023-08-04 22:19:25#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 37000, 900 bytes
0#2023-08-04 22:19:45#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 38000, 900 bytes
0#2023-08-04 22:19:57#>sent control command to peer, cmd = 0x01, ClientID = 1, the data length is 1128 bytes
0#2023-08-04 22:19:56#>Acknowledge packet, AackedLinkId = 0, ClientID = 1, AackedTimestamp <2023-08-04 22:19:57>
0#2023-08-04 22:20:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 39000, 900 bytes
0#2023-08-04 22:20:25#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 40000, 900 bytes

```

测试用的 tcp 服务器位于带宽为 1Mbps 的阿里云上，协议栈平均 20 秒上传 1000 个数据包，每个包携带 900 字节的用户数据。

当 SUPPORT\_SACK 宏置位时：

```

管理员: C:\Windows\system32\cmd.exe - TcpServerForStackTesting.exe
1#2023-08-04 22:28:17#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 3000, 900 bytes
1#2023-08-04 22:28:24#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 4000, 900 bytes
1#2023-08-04 22:28:31#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 5000, 900 bytes
1#2023-08-04 22:28:38#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6000, 900 bytes
1#2023-08-04 22:28:44#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 7000, 900 bytes
1#2023-08-04 22:28:51#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 8000, 900 bytes
1#2023-08-04 22:28:58#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 9000, 900 bytes
1#2023-08-04 22:29:05#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 10000, 900 bytes
1#2023-08-04 22:29:12#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 11000, 900 bytes
1#2023-08-04 22:29:19#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 12000, 900 bytes
1#2023-08-04 22:29:26#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 13000, 900 bytes
1#2023-08-04 22:29:34#>sent control command to peer, cmd = 0x01, ClientID = 1, the data length is 1128 bytes
1#2023-08-04 22:29:33#>Acknowledge packet, AackedLinkId = 1, ClientID = 1, AackedTimestamp <2023-08-04 22:29:34>
1#2023-08-04 22:29:33#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 14000, 900 bytes
1#2023-08-04 22:29:40#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 15000, 900 bytes
1#2023-08-04 22:29:47#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 16000, 900 bytes
1#2023-08-04 22:29:54#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 17000, 900 bytes
1#2023-08-04 22:30:01#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 18000, 900 bytes
1#2023-08-04 22:30:08#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 19000, 900 bytes
1#2023-08-04 22:30:15#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 20000, 900 bytes
1#2023-08-04 22:30:22#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 21000, 900 bytes
1#2023-08-04 22:30:29#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 22000, 900 bytes
1#2023-08-04 22:30:37#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 23000, 900 bytes
1#2023-08-04 22:30:44#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 24000, 900 bytes
1#2023-08-04 22:30:51#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 25000, 900 bytes
1#2023-08-04 22:30:59#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 26000, 900 bytes
1#2023-08-04 22:31:06#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 27000, 900 bytes
1#2023-08-04 22:31:10#>sent control command to peer, cmd = 0x01, ClientID = 1, the data length is 1128 bytes
1#2023-08-04 22:31:09#>Acknowledge packet, AackedLinkId = 1, ClientID = 1, AackedTimestamp <2023-08-04 22:31:10>
1#2023-08-04 22:31:13#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 28000, 900 bytes
1#2023-08-04 22:31:20#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 29000, 900 bytes
1#2023-08-04 22:31:27#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 30000, 900 bytes
1#2023-08-04 22:31:35#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 31000, 900 bytes
1#2023-08-04 22:31:43#>Uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 32000, 900 bytes

```

基本 7 秒 1000 组，速度差异还是很明显的。如果我们重新设计应用层协议（目前应用层也是一问一答机制），采用类似 ftp 的大块数据传输机制，速度将会有更加明显的差异。

### 3.1.12 tcp 服务器测试

常见的 tcp 服务器要完成的工作无外乎就是接受连接请求，接收客户端上发的数据，下发应答或控制报文，清除不活跃的客户端以释放其占用的系统资源。我们要编写的这个测试服务器也要完成这些工作。当然，为了突出如何利用协议栈提供的 socket api 编写 tcp 服务器这条主线，我们还是只给出主要逻辑代码，非关键业务及容错代码请参考 gitee 上的[样例工程](#)。我们把 tcp 服务器分为两部分实现：一部分在主线程（任务）完成启动 tcp 服务器、等待接受连接请求、清除不活跃客户端的工作；另一部分单独建立一个线程（任务）完成读取客户端数据并下发应答报文的工作。首先编写主线程（任务代码），依然是在 main.c 文件，rt-thread 下添加的测试代码如下：

```

.....

#define THTCPSRV_PRIO      27      /* tcp 服务器数据读取线程优先级 */
#define THTCPSRV_STK_SIZE  256 * 4 /* tcp 服务器数据读取线程栈大小 */
#define THTCPSRV_TIMESLICE 10      /* 分配给 tcp 服务器数据读取线程的时间片 */

#define LTCPSRV_PORT       6411    /* tcp 测试服务器端口 */
#define LTCPSRV_BACKLOG_NUM 5      /* 排队等待接受连接请求的客户端数量 */

```



```
static SOCKET l_hSockSrv; /* tcp 服务器 socket, 这是一个静态存储时期的变量, 因为服务器数据接收线程也要使用这个变量
.....

/* 启动 tcp 服务器
SOCKET tcp_server_start(USHORT usSrvPort, USHORT usBacklog)
{
    EN_ONPSERR enErr;
    SOCKET hSockSrv;

    do {
        /* 申请一个 socket
        hSockSrv = socket(AF_INET, SOCK_STREAM, 0, &enErr);
        if(INVALID_SOCKET == hSockSrv)
            break;

        /* 绑定地址和端口, 功能与 Berkeley sockets 提供的 bind() 函数相同
        if(bind(hSockSrv, NULL, usSrvPort))
            break;

        /* 启动监听, 同样与 Berkeley sockets 提供的 listen() 函数相同
        if(listen(hSockSrv, usBacklog))
            break;

        /* 可以在这里添加客户端管理相关的初始代码, 用于管理连接上来的客户端, 以实现不活跃的客户端清除功能
        .....

        return hSockSrv;
    } while(FALSE);

    /* 执行到这里意味着前面出现了错误, 无法正常启动 tcp 服务器了
    if(INVALID_SOCKET != hSockSrv)
        close(hSockSrv);
    printf("%s\r\n", onps\_error(enErr));

    /* tcp 服务器启动失败, 返回一个无效的 socket 句柄
    return INVALID_SOCKET;
}

int main(void)
{
    .....
    __lblStart:
    /* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
    .....

    /* 启动 tcp 服务器
    l_hSockSrv = tcp_server_start(LTCPSRV_PORT, LTCPSRV_BACKLOG_NUM);
    if(INVALID_SOCKET != l_hSockSrv)
```

```

{
    tid = rt_thread_create("THTcpSrv", THTcpSrv, RT_NULL, THTCPSRV_STK_SIZE, THTCPSRV_PRIO, THTCPSRV_TIMESLICE);
    if(tid != RT_NULL)
        rt_thread_startup(tid);
}

.....

/* 进入主线程的主逻辑处理循环
while(TRUE)
{
    /* 你原先的处理逻辑代码
    .....

    /* 接受连接请求
    SOCKET hSockClt = accept(l_hSockSrv, &unCltIP, &usCltPort, 1, &enErr);
    if(INVALID_SOCKET != hSockClt)
    {
        /* 将其放入客户端的管理链表，以便能够检测并清除不活跃的客户端
        .....
    }
    else
        printf("accept() failed, %s\r\n", onps\_error(enErr));

    /*调用 close() 函数清除不活跃的客户端
    .....
}
}

```

ucos ii 下要添加的代码除了 tcp 数据读取任务的建立代码有区别外，其它与 rt-thread 下相同。因此，这里只给出数据读取任务的建立代码，其它不再给出：

```

.....

#define THTCPSRV_PRIO      31  /* tcp 服务器数据读取任务优先级
#define THTCPSRV_STK_SIZE 256 /* tcp 服务器数据读取任务栈大小
__align(8) OS_STK THTCPSRV_STK[THTCPSRV_STK_SIZE]; /* tcp 服务器数据读取任务栈
.....

static void THMain(void *pvData)
{
    .....
    _lblStart:
    /* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
    .....

    /* 启动 tcp 服务器
    l_hSockSrv = tcp_server_start(LTCPSRV_PORT, LTCPSRV_BACKLOG_NUM);
    if(INVALID_SOCKET != l_hSockSrv)
        OSTaskCreate(THTcpSrv, (void *)0, (OS_STK *)&THTCPSRV_STK[THTCPSRV_STK_SIZE - 1], THTCPSRV_PRIO);
    .....
}

```

接下来完成数据读取线程（任务）THTcpSrv，依然是只给出与协议栈相关的主要处理逻辑代码，其它参见 gitee 上的[样例工程](#)：

```
/* 完成 tcp 服务器的数据读取工作
static void THTcpSrv(void *pvData)
{
    SOCKET hSockClt;
    EN_ONPSERR enErr;
    INT nRcvBytes;
    UCHAR ubaRcvBuf[256];

    while(TRUE)
    {
        /* 协议栈利用 rtos 提供的信号量实现了一个 poll 模型，当有一个及以上的 tcp 客户端数据到达，均会触发一个信号到用户层，
        /* 我们通过 tcpsrv_recv_poll() 函数等待这个信号。这个函数的第二个参数值 1 表示这个函数最长等待 1 秒，1 秒之内有任意一个
        /* 或多个客户端数据到达则立即返回最先到达的这个客户端的 socket，继续调用这个函数则继续返回下一个客户端 socket，直至
        /* 返回一个无效的 socket 才意味着当前所有已送达的数据均已读取完毕，已经没有任何客户端有新数据到达了
        hSockClt = tcpsrv\_recv\_poll(l_hSockSrv, 1, &enErr);
        if(INVALID_SOCKET != hSockClt) /* 有效的 socket
        {
            /* 注意这里一定要尽量读取完毕该客户端的所有已到达的数据，因为每个客户端只有新数据到达时才会触发一个信号到用户层
            /* ，如果你没有读取完毕就只能等到该客户端送达下一组数据时再读取了，这可能会导致数据处理延迟问题
            while(TRUE)
            {
                /* 读取数据
                nRcvBytes = recv(hSockClt, ubaRcvBuf, 256);
                if(nRcvBytes > 0)
                {
                    /* 原封不动的回送给客户端，利用回显来模拟服务器回馈应答报文的场景
                    send(hSockClt, ubaRcvBuf, nRcvBytes, 1);
                }
                else /* 已经读取完毕
                {
                    if(nRcvBytes < 0)
                    {
                        /* 协议栈底层报错，这里需要增加你的容错代码处理这个错误并打印错误信息
                        printf("%s\r\n", onps\_get\_last\_error(hSocket, NULL));
                    }
                    break;
                }
            }
        }
        else /* 无效的 socket
        {
            /* 返回一个无效的 socket 时需要判断是否存在错误，如果不存在则意味着 1 秒内没有任何数据到达，否则打印这个错误
            if(ERRNO != enErr)
                printf("tcpsrv_recv_poll() failed, %s\r\n", onps\_error(enErr));
```

```
}  
}  
}
```

按照我们的传统习惯，给出的测试代码蓝色字体部分为协议栈提供的 socket api。红色字体的注释部分则为编写 tcp 服务器的几个主要步骤：

- 1) 调用 socket 函数，申请一个数据流(tcp)类型的 socket；
- 2) bind() 函数绑定一个 ip 地址和端口号；
- 3) listen() 函数启动监听；
- 4) accept() 函数接受一个 tcp 连接请求；
- 5) 调用 tcpsrv\_recv\_poll() 函数利用协议栈提供的 poll 模型（非传统的 select 模型）等待客户端数据到达；
- 6) 调用 recv() 函数读取客户端数据并处理之，直至所有数据读取完毕返回第 5 步，获取下一个已送达数据的客户端 socket；
- 7) 定期检查不活跃的客户端，调用 close() 函数关闭 tcp 链路，释放客户端占用的协议栈资源；

与传统的 tcp 服务器编程并没有两样，处理流程包括用到的 socket api 亦基本相同，这里就不再啰嗦了。上述代码中涉及到的 socket api 的使用说明同样参见本文[第 4 节——socket 使用说明](#)。这里单独说一下测试代码中提到的 poll 模型。

poll 模型利用了 rtos 的信号量机制。当某个 tcp 服务器端口有一个或多个客户端有新的数据到达时，协议栈会立即投递一个或多个信号到用户层。注意，协议栈投递信号的数量取决于新数据到达的次数（tcp 层每收到一个携带数据的 tcp 报文记一次），与客户端数量无关。用户通过 tcpsrv\_recv\_poll() 函数得到这个信号，并得到最先送达数据的客户端 socket，然后读取该客户端送达的数据。注意这里一定要把所有数据读取出来。因为信号被投递的唯一条件就是有新的数据到达。没有信号，tcpsrv\_recv\_poll() 函数无法得到一个有效的客户端 socket，那么剩余数据就只能等到该客户端再次送达新数据时再读了。

其实，poll 模型的运作机制非常简单。tcp 服务器每收到一组新的数据，就会将该数据所属的客户端 socket 放入接收队列尾部，然后投信号。所以，数据到达、获取 socket 与投递信号是一系列的连锁反应，且一一对应。tcpsrv\_recv\_poll() 函数则在用户层接着完成连锁反应的后续动作：等信号、摘取接收队列首部节点、取出首部节点保存的 socket、返回该 socket 以告知用户立即读取数据。非常简单明了，没有任何拖泥带水。从这个运作机制我们可以看出：

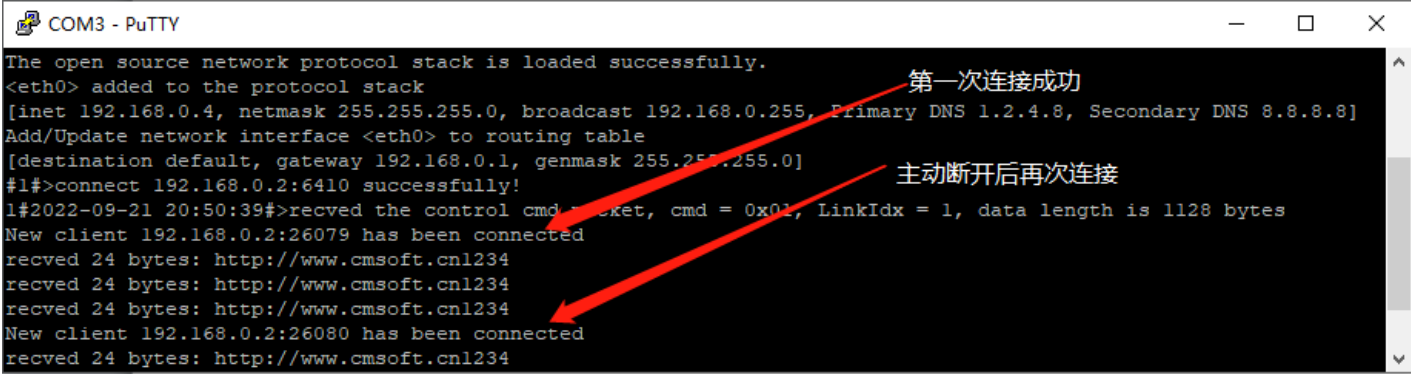
- 1) poll 模型的运转效率取决于 rtos 的信号量处理效率；
- 2) tcpsrv\_recv\_poll() 函数每次返回的 socket 有可能是同一个客户端的，也可能是不同客户端；
- 3) 单个客户端已送达的数据长度与信号并不一一对应，一一对应的是该客户端新数据到达的次数与信号投递的次数，所以当数据读取次数小于信号数时，存在读取数据长度为 0 的情形；
- 4) tcpsrv\_recv\_poll() 函数返回有效的 socket 后，尽量读取全部数据到用户层进行处理，否则会出现剩余数据无法读取的情形，如果客户端不再上发新的数据的话；

协议栈还为用户提供了另外一种数据读取模式：主动读取模式。在某些应用场景中，用户需要主动读取客户端到达的数据而不是被动等待 poll 模型的通知，此时就需要通过调用 tcpsrv\_set\_recv\_mode() 函数显式地设置服务器为主动读取模式。有关这个函数的详情请参阅《onps 栈 API 接口手册》“Berkeley sockets”一节。这里唯一需要交待的地方是，对这个函数的调用一定要在 listen() 函数之后，因为 listen() 函数会将读取模式缺省设置为 poll 被动等待模型。

实际测试之前，我们还需要把 emac.h 文件中的 DHCP\_REQ\_ADDR\_EN 宏关闭，配置网卡为静态地址，而不是动态获取。毕竟 tcp 服务器的 ip 地址才是对外提供有效服务的关键所在。我们把测试代码更新到开发板上，然后在网上找个网络调试助手之类的工具软件，用它作为 tcp 客户端连接开发板上的测试服务器：



开发板控制台的日志输出如下：



此时，为了测试 tcp 栈的可靠性和稳定性，开发板上还同时运行着上一节添加的 tcp 客户端。pc 上的 tcp 测试服务器也开着。所以我们可以看到上面开发板的输出日志中包含着我们熟悉的内容：

```
#1#>connect 192.168.0.2:6410 successfully!  
1#2022-09-21 20:50:39#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 1, data length is 1128 bytes
```

从实际测试情况看，网络调试助手开启连续发送，周期 100 毫秒，板子上的 tcp 服务器一直很稳定的回显收到的报文给网络调试助手。同时开发板上的 tcp 客户端也很稳定的与 pc 上的 tcp 测试服务器进行大数据量通讯。有关 tcp 服务器开发的细节可参考协议栈 telnet 服务器源码。

### 3.1.13 udp 通讯测试

相比 tcp 编程，udp 编程相对简单很多。为 udp 绑定一个固定端口其就可以作为服务器使用，反之则作为一个客户端使用。下面给出的测试代码依然是突出主线，忽略非关键业务及容错代码。我们为 udp 测试单独建立一个线程（任务）。依然是 main.c 文件，rt-thread 下 udp 测试任务的建立代码如下：

```

.....

#define THUDPCOMMU_PRIO      28      /* udp 通讯测试线程优先级
#define THUDPCOMMU_STK_SIZE  200 * 4 /* udp 通讯测试线程栈大小
#define THUDPCOMMU_TIMESLICE 5       /* udp 通讯测试线程的运行时间片
.....

int main(void)
{
    .....
    __lblStart:
    /* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
    .....
    /* 启动 udp 通讯测试线程
    tid = rt_thread_create("THUdpCommu", THUdpCommu, RT_NULL, THUDPCOMMU_STK_SIZE, THUDPCOMMU_PRIO, THUDPCOMMU_TIMESLICE);
    if(tid != RT_NULL)
        rt_thread_startup(tid);
    .....
}

```

ucosii 下:

```

.....

#define THUDPCOMMU_PRIO      33 /* udp 通讯测试任务优先级
#define THUDPCOMMU_STK_SIZE 200 /* udp 通讯测试任务栈大小
__align(8) OS_STK THUDPCOMMU_STK[THUDPCOMMU_STK_SIZE]; /* udp 通讯测试任务栈
.....

static void THMain(void *pvData)
{
    .....
    __lblStart:
    /* 在这里可以启动其它需要协议栈及网卡驱动加载成功后才能启动的线程
    .....
    /* 启动 udp 通讯测试任务
    OSTaskCreate(THUdpCommu, (void *)0, (OS_STK *)&THUDPCOMMU_STK[THUDPCOMMU_STK_SIZE - 1], THUDPCOMMU_PRIO);
    .....
}

```

udp 通讯测试线程（任务）THUdpCommu 的主要处理逻辑如下:

```

#define RUDPSRV_IP    "192.168.0.2" /* 远端 udp 服务器的地址
#define RUDPSRV_PORT 6416          /* 远端 udp 服务器的端口
#define LUDPSRV_PORT 6415          /* 本地 udp 服务器的端口

/* udp 通讯用缓冲区（接收和发送均使用）
static UCHAR l_ubaUdpBuf[256];

static void THUdpCommu(void *pvData)
{
    EN_ONPSERR enErr;

```

```
SOCKET hSocket = INVALID_SOCKET;
while(TRUE)
{
    /* 尚未分配一个 socket, 先申请一个 socket
    if(INVALID_SOCKET == hSocket)
    {
        /* 分配一个 socket
        hSocket = socket(AF_INET, SOCK_DGRAM, 0, &enErr);

        /* 返回了一个无效的 socket
        if(INVALID_SOCKET == hSocket)
        {
            printf("<1>socket() failed, %s\r\n", onps\_error(enErr));
            continue;
        }
    }
    #if 0
        /* 如果是想建立一个 udp 服务器, 这里需要调用 bind() 函数绑定地址和端口
        if(bind(hSocket, NULL, LUDPSRV_PORT))
        {
            printf("bind() failed, %s\r\n", onps\_get\_last\_error(hSocket, NULL));

            /* 关闭 socket 释放占用的协议栈资源
            close(hSocket);
            hSocket = INVALID_SOCKET;

            continue;
        }
    #else
        /* 建立一个 udp 客户端, 在这里可以调用 connect() 函数绑定一个固定的目标服务器, 接下来就可以直接使用 send() 函数发送
        /* 数据, 当然在这里你也可以什么都不做 (不调用 connect()), 但接下来你需要使用 sendto() 函数指定要发送的目标地址
        if(connect(hSocket, RUDPSRV_IP, RUDPSRV_PORT, 0))
        {
            printf("connect %s:%d failed, %s\r\n", RUDPSRV_IP, RUDPSRV_PORT, onps\_get\_last\_error(hSocket, NULL));

            /* 关闭 socket 释放占用的协议栈资源
            close(hSocket);
            hSocket = INVALID_SOCKET;

            continue;
        }
    #endif

    /* 与 tcp 客户端测试一样, 接收数据之前要设定 udp 链路的接收等待的时间, 单位: 秒, 这里设定 recv() 函数等待 1 秒
    if(!socket\_set\_rcv\_timeout(hSocket, 1, &enErr))
        printf("socket_set_rcv_timeout() failed, %s\r\n", szNowTime, onps\_error(enErr));
}

/* 发缓冲区填充一段字符串然后得到其填充长度
```



```
..... 填充

INT nSendDataLen = strlen((const char *)l_ubaUdpBuf); /* 填充的字符串的长度
/* 调用 send() 函数发送数据
/* 如果实际发送长度与字符串长度不相等则说明发送失败
if(nSendDataLen != send(hSocket, l_ubaUdpBuf, nSendDataLen, 0))
    printf("send failed, %s\r\n", onps\_get\_last\_error(hSocket, NULL));

/* 接收对端数据之前清 0，以便本地能够正确输出收到的对端回馈的字符串
memset(l_ubaUdpBuf, 0, sizeof(l_ubaUdpBuf));
/* 调用 recv() 函数接收数据，如果想知道对端地址调用 recvfrom() 函数，在这里 recv() 函数为阻塞模式，最长阻塞 1 秒(如
/* 果未收到任何 udp 报文的话)
INT nRcvBytes = recv(hSocket, l_ubaUdpBuf, sizeof(l_ubaUdpBuf));
if(nRcvBytes > 0)
    printf("recv %d bytes, Data = <%s>\r\n", nRcvBytes, (const char *)l_ubaUdpBuf);
else
{
    /* 小于 0 则意味着 recv() 函数报错
    if(nRcvBytes < 0)
    {
        printf("recv failed, %s\r\n", onps\_get\_last\_error(hSocket, NULL));

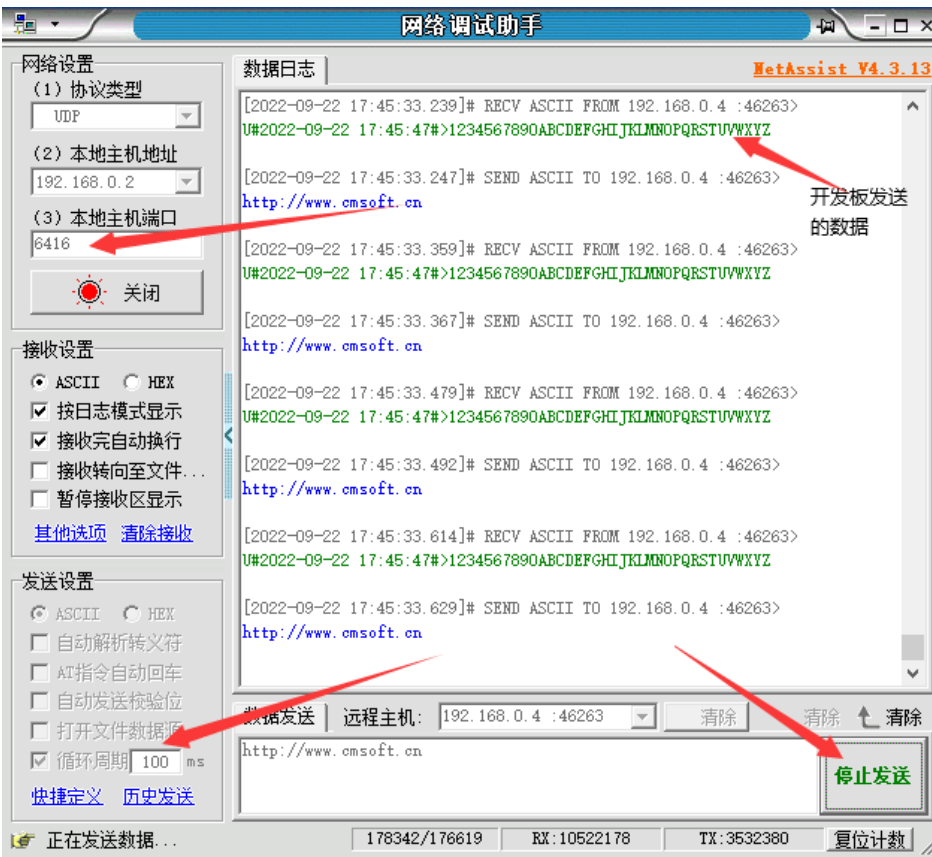
        /* 关闭 socket 释放占用的协议栈资源
        close(hSocket);
        hSocket = INVALID_SOCKET;
    }
}
}
```

udp 通讯编程依然遵循了传统习惯，主要编程步骤还是那些：

- 1) 调用 socket 函数，申请一个 SOCK\_DGRAM(udp)类型的 socket；
- 2) 如果想建立服务器，调用 bind() 函数；想与单个目标地址通讯，调用 connect() 函数；与任意目标地址通讯则什么都不用做；
- 3) 调用 send() 或 sendto() 函数发送 udp 报文；
- 4) 调用 recv() 或 recvfrom() 函数接收 udp 报文；
- 5) close() 函数关闭 socket 释放当前占用的协议栈资源；

把程序更新到开发板，然后打开网络调试助手，指定 udp 端口为 6416：





开发板控制台的日志输出如下：

```
The open source network protocol stack is loaded successfully.
<eth0> added to the protocol stack
[inet 192.168.0.4, netmask 255.255.255.0, broadcast 192.168.0.255, Primary DNS 1.2.4.8, Secondary DNS 8.8.8.8]
Add/Update network interface <eth0> to routing table
[destination default, gateway 192.168.0.1, genmask 255.255.255.0]
#l#>connect 192.168.0.2:6410 failed, tcp connection timeout
#l#>connect 192.168.0.2:6410 successfully!
l#2022-09-22 17:44:33#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 3, data length is 1128 bytes
U#2022-09-22 17:44:49#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-22 17:44:49#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-22 17:44:49#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-22 17:44:49#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-22 17:44:50#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-22 17:44:50#>recv 20 bytes, Data = <http://www.cmsoft.cn>
```

依然是同时进行了大数据量的 tcp 客户端测试，udp 通讯丝毫没收到影响。

3.1.14 telnet 测试

以太网设备在联网过程遇到的最多问题莫过于地址冲突、路由修改等。对于那些没有人机交互接口的终端设备来说，这些都是不好解决的问题。为此，协议栈提供了一个轻型 telnet 服务为用户提供网络层的人机交互接口。有关如何使能 telnet 服务及移植、命令扩展相关的说明请参阅《onps 栈移植手册》1.2 节。

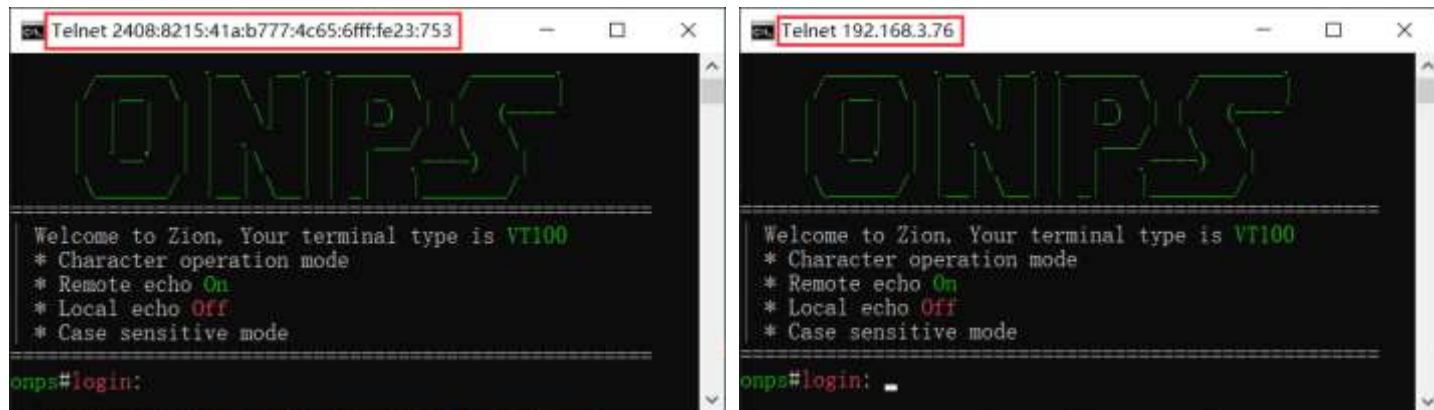
```
telnet 对外提供服务的端口在 telnet_srv.h 文件中定义：
#define TELNETSRV_PORT 23 /* telnet 服务器缺省端口

我们可以根据实际需求调整该端口。这个文件还定义了 telnet 客户端的超时时间：
/* 定义 telnet 客户端最长多少秒可以无任何操作，超过这个时间服务器会主动断开当前连接
#define TELNETCLT_INACTIVE_TIMEOUT 300

超时时间这个值对于资源受限的系统来说非常重要。它决定了资源被占用的最大时长，超过这个时间资源将被立即回收。
```

在真实应用场景中，telnet 登录用户长时间不进行任何操作的情况是存在的，比如用户操作完毕忘记主动注销登录等。我们称之为“僵尸”客户端。如果服务器不去处理这种情况，资源很快就会被耗尽。TELNETCLT\_INACTIVE\_TIMEOUT 的取值并无统一标准，取值大小直接反应了我们对“僵尸”客户端的容忍程度。

telnet 服务是作为普通服务添加到目标系统中的，具体实现代码请参照《onps 栈移植手册》第 5 节并结合样例工程进行编写。telnet 服务同时支持 ipv4 及 ipv6（SUPPORT\_IPV6 置位），其实际运行效果如下图所示：

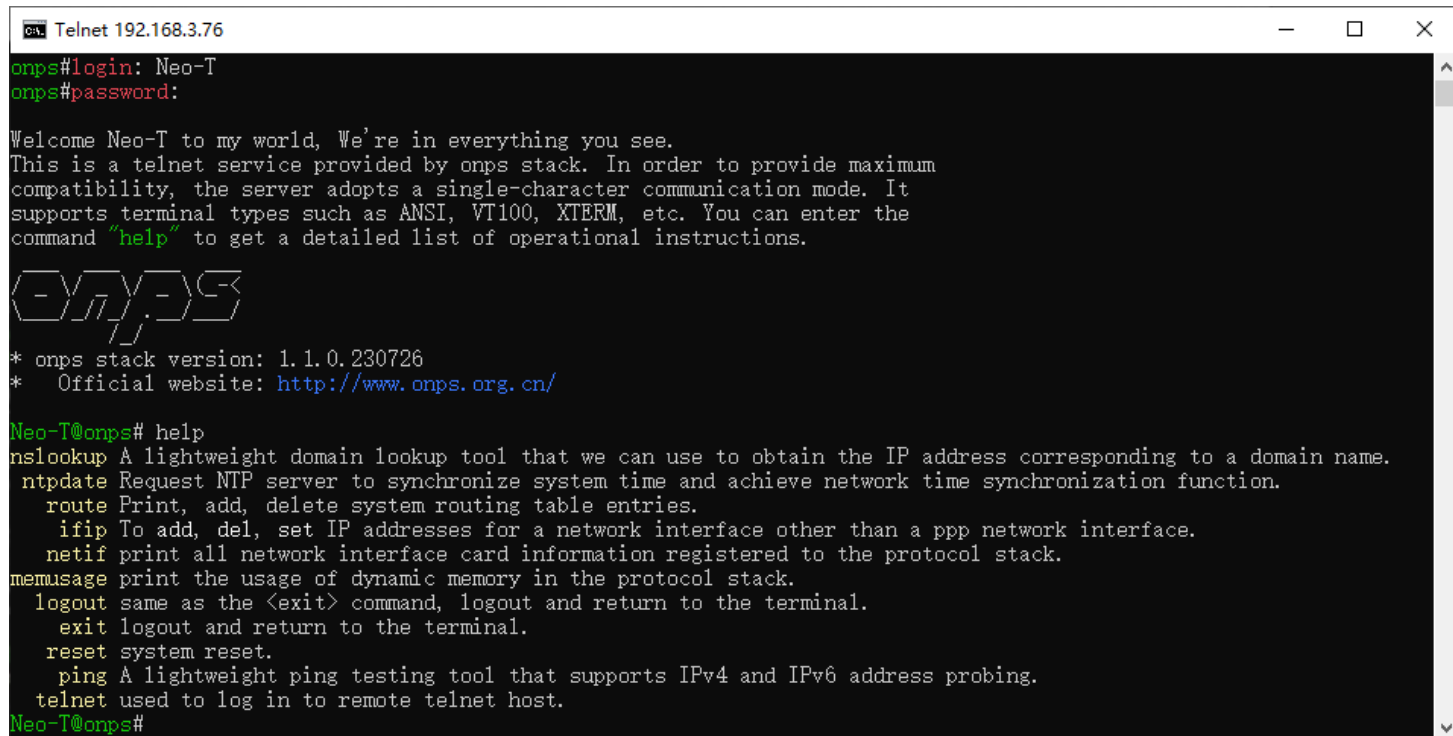


telnet 支持 ipv6 会给我们带来一个意想不到的效果。众所周知，ipv6 是为了解决 ip 地址不足的问题。它甚至可以为每一粒沙子分配一个唯一的 ip 地址。所以，一旦我们在终端系统中支持 ipv6，且终端所在的网络路由器支持 ipv6，那么我们就可以在全球任意位置随时随地远程 telnet 到终端。这对于有远程调参、调试需求的用户来说将是巨大的优势。要想测试这一点其实很简单。用我们的手机作热点，pc 连接手机热点。此时再 telnet 终端 ipv6 地址，如果终端所在的路由器防火墙已放开该端口，我们将再次看到 onps 栈 telnet 登录界面。

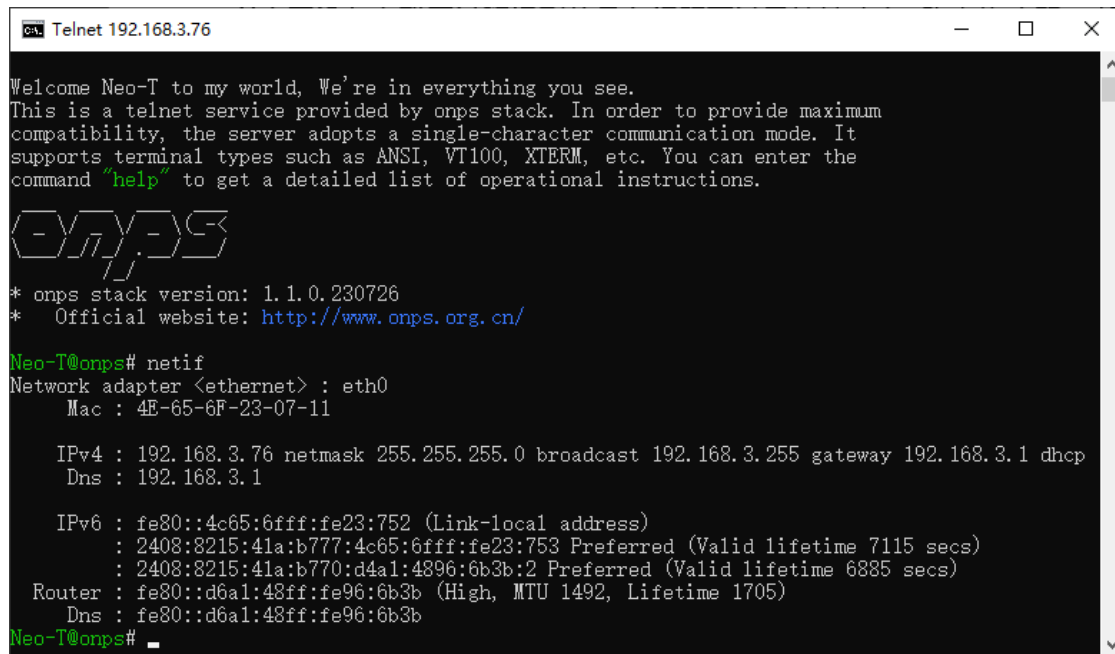
用户登录账户及密码在 net\_virtual\_terminal.h 文件中定义：

```
#define NVT_USER_NAME    "Neo-T"
#define NVT_USER_PASSWD "1964101615"  /* 中国第一颗原子弹爆炸零时：1964 年 10 月 16 日 15 时，请铭记这一历史时刻
```

实际应用场景下，我们可以把这两个信息加密后保存到系统非易失性存储器中，同时增加一个修改登录账户和密码的 NVT 命令。这样更符合商业系统标准。登陆成功后我们可以输入“help”查看目标系统支持的命令列表：



netif 命令用于打印输出网络接口信息：



```

Telnet 192.168.3.76

Welcome Neo-T to my world, We're in everything you see.
This is a telnet service provided by onps stack. In order to provide maximum
compatibility, the server adopts a single-character communication mode. It
supports terminal types such as ANSI, VT100, XTERM, etc. You can enter the
command "help" to get a detailed list of operational instructions.

onps
* onps stack version: 1.1.0.230726
* Official website: http://www.onps.org.cn/

Neo-T@onps# netif
Network adapter <ethernet> : eth0
    Mac : 4E-65-6F-23-07-11

    IPv4 : 192.168.3.76 netmask 255.255.255.0 broadcast 192.168.3.255 gateway 192.168.3.1 dhcp
    Dns : 192.168.3.1

    IPv6 : fe80::4c65:6fff:fe23:752 (Link-local address)
           : 2408:8215:41a:b777:4c65:6fff:fe23:753 Preferred (Valid lifetime 7115 secs)
           : 2408:8215:41a:b770:d4a1:4896:6b3b:2 Preferred (Valid lifetime 6885 secs)
    Router : fe80::d6a1:48ff:fe96:6b3b (High, MTU 1492, Lifetime 1705)
    Dns : fe80::d6a1:48ff:fe96:6b3b
Neo-T@onps#
```

其它一些网络配置命令诸如修改 ip 地址的 ifip 、增删路由表的 route 等，只要不携带任何参数，就能够在控制台得到帮助信息，具体使用方法不再赘述。

对于 ping 命令，协议栈仅实现了 ip 地址直接 ping，未实现 ping 域名。如果有实际需求，可以直接修改 ping.c 文件下的 nvt\_cmd\_ping\_entry() 函数，增加域名转换处理。ntpdate 命令支持域名及 ip 地址直接校时。telnet 命令则是一个轻型的客户端工具，其用于登录其它提供 telnet 服务的主机，如 linux 或 windows。它的存在使得运行协议栈的终端设备可以作为 telnet 服务的中继代理登录其所在内部网络的任意一台主机，实现 telnet 网络穿透。如下所示：

```

Telnet 192.168.3.76

=====
  ONPS
=====
Welcome to Zion, Your terminal type is VT100
* Character operation mode
* Remote echo On
* Local echo Off
* Case sensitive mode
=====
onps#login: Neo-T
onps#password:

Welcome Neo-T to my world, We're in everything you see.
This is a telnet service provided by onps stack. In order to provide maximum
compatibility, the server adopts a single-character communication mode. It
supports terminal types such as ANSI, VT100, XTERM, etc. You can enter the
command "help" to get a detailed list of operational instructions.

=====
  ONPS
=====
* onps stack version: 1.1.0.230726
* Official website: http://www.onps.org.cn/

Neo-T@onps# telnet 192.168.3.68
Connecting to telnet server ...
Ubuntu 20.04 LTS
rubao login: mars
Password:
Welcome to Ubuntu 20.04 LTS (GNU/Linux 4.4.0-19041-Microsoft x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sat Aug  5 17:07:00 CST 2023

System load:          0.52
Usage of /home:        unknown
Memory usage:         76%
Swap usage:           0%
Processes:            10
Users logged in:      0
IPv4 address for eth0: 192.168.0.68
IPv4 address for eth0: 192.168.1.68
IPv4 address for eth0: 192.168.3.68
IPv6 address for eth0: 2408:8215:41a:b777:198:5d47:9af5:e1c
IPv6 address for eth0: 2408:8215:41a:b770:d4a1:4896:6b3b:3
IPv6 address for eth0: 2408:8215:41a:b777:eddd:2d80:6c63:d769

0 updates can be installed immediately.
0 of these updates are security updates.

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Thu Jul 20 18:32:06 CST 2023 from ::ffff:192.168.3.73 on pts/0
mars@rubao:~$ ls -al
total 20
drwxr-xr-x 1 mars mars 4096 Jul 20 21:46 .
drwxr-xr-x 1 root root 4096 May 16 2020 ..
-rw-r----- 1 mars mars 26290 Jul 20 21:46 .bash_history
-rw-r--r-- 1 mars mars 220 May 16 2020 .bash_logout
-rw-r--r-- 1 mars mars 3771 May 16 2020 .bashrc
drwx----- 1 mars mars 4096 May 25 19:08 .cache
drwx----- 1 mars mars 4096 May 24 2020 .config
drwxr-xr-x 1 mars mars 4096 May 16 2020 .landscape
-rw-rw-rw- 1 mars mars 0 Aug  5 17:05 .motd_shown
-rw-r--r-- 1 mars mars 807 May 16 2020 .profile
drwx----- 1 mars mars 4096 Jan 12 2021 .ssh
-rw-r--r-- 1 mars mars 0 May 16 2020 .sudo_as_admin_successful
-rw-r----- 1 mars mars 13288 Jul 20 18:32 .viminfo
-rw-rw-r-- 1 mars mars 18 Jul 20 18:32 test
mars@rubao:~$
```

既然是代理，当我们成功登录到目标主机后，我们可以像直接登录一样进行操作，协议栈只负责转发不对操作做任何干涉。

我们可以为 telnet 模块增加自己的命令，比如查看运行日志、扩展功能模块等。只要充分发挥我们的想象力，在这个运行框架下，这些都不是问题。有关命令扩展的内容请参阅《onps 栈移植手册》5.3 节。移植样例工程也提供了命令扩展例程，篇幅原因就不再展开了。

## 3.2 ppp 拨号场景下的移植

### 3.2.1 准备工作

我进行移植的硬件平台为 STM32F103RCT6，4G 模块为龙尚 U9300C。目标 rtos 依然是 rt-thread 和 ucosii。移植之前要确保 rtos 能够在目标板上正常运行；连接 4g 模块的串口驱动及 4g 模块均能正常工作；rtc、printf 等模块也能够正常工作。准备工作中最重要的是串口驱动。**要想让 ppp 链路获得最佳性能，驱动一定要采用 dma 方式，传统的单字符中断的读写方式是无法满足大数据量通讯需求的。**另外，串口驱动要提供一组导出函数用于协议栈移植：

- 1) 串口初始化函数，完成引脚、通讯速率、校验方式等配置工作；完成开启串口 dma、开启收发中断等工作；
- 2) 串口重新初始化函数，完成重新初始化串口 dma 的工作（当 ppp 链路出现异常时该函数能够清空 dma 缓冲区，确保重新建立 ppp 链路时，dma 能够以初始状态重新工作）；
- 3) 串口发送函数，能够把数据发送给 4g 模块；
- 4) 串口接收函数，能够等待接收 4g 模块发送的数据，可以指定等待的最大时长，单位：秒；

我手头使用的目标平台把 4g 模块接在了 mcu 的串口 3 上，开启了串口 3 的 dma 支持，驱动对应的 4 个导出函数为：

#### 1) SCP3Init()

串口初始化函数，原型为：

```
BOOL SCP3Init(uint32_t unBaudrate,    /* 波特率
                                     uint16_t usDataBit,    /* 数据位
                                     uint16_t usStopBits,   /* 停止位
                                     uint16_t usParityType, /* 奇偶校验类型
                                     uint16_t usFlowCtlType /* 流控类型
                                     );
```

初始化成功返回 TRUE，否则返回 FALSE；

#### 2) SCP3ReInit()

串口重新初始化函数，原型：

```
void SCP3ReInit(void);
```

#### 3) SCP3Send()

串口发送函数，把数据发送到 4g 模块，原型：

```
uint32_t SCP3Send(unsigned char *pubPacket, /* 数据指针，指向要发送的数据
                                     uint32_t unPacketLen    /* 要发送的数据长度
                                     );
```

返回值为实际发送的字节数；

#### 4) SCP3Recv()

串口接收函数，等待接收 4g 模块发送的数据，原型：

```
uint32_t SCP3Recv(unsigned char *pubRcvBuf, /* 指向数据接收缓冲区的指针
        uint32_t unRcvBufLen,      /* 接收缓冲区长度
        int32_t nWaitSecs          /* 指定要等待的最大时长，单位：秒，0 为一直等待；大于 0，等待指定的秒数；
                                   /* 小于 0，不支持
    );
    返回值大于等于 0，为实际收到的字节数；
```

上述函数的具体实现代码参见 gitee 上的[移植样例工程](#)（driver/scp.c）。

移植的第一步依然是把 port 目录下的相关文件放到我们的目标工程中。由于 rtos 相关的信号量、互斥锁之类的函数在 ppp 场景下依然使用，所以我们直接把以太网场景下已经移植好的相关文件拿到 ppp 工程下：

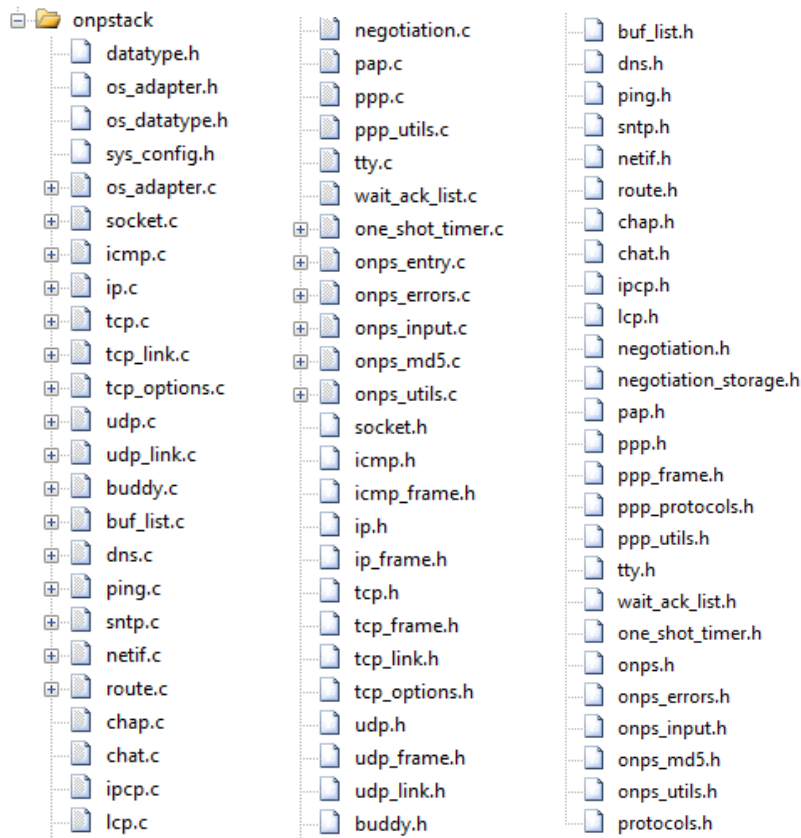
```
include/port/datatype.h、include/port/os_adapter.h、include/port/os_datatype.h、include/port/sys_config.h
port/os_adapter.c
```

把上述几个文件再加协议栈核心文件都添加到我们的 IDE 中：

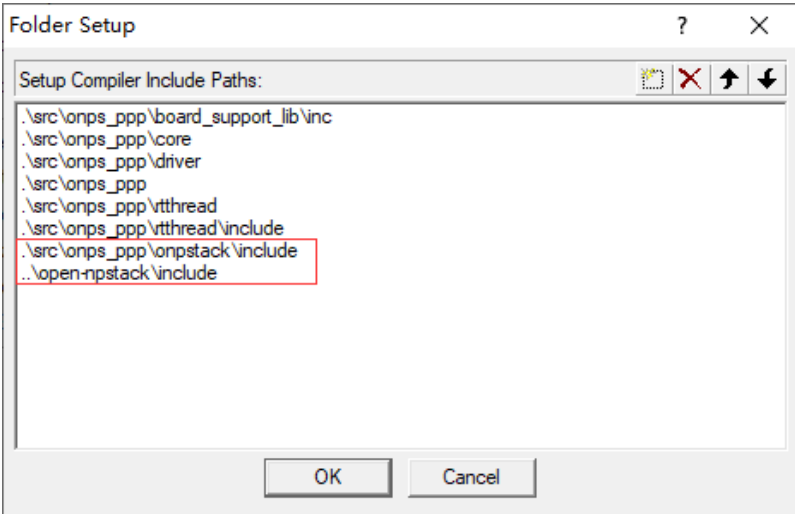
```
bsd/
ppp/
include/ 除 ethernet 相关的头文件外，其余全部添加
ip/
mmu/
net_tools/
netif/
one_shot_timer.c
onps_entry.c
onps_errors.c
onps_input.c
onps_md5.c
onps_utils.c
```

IDE 工程中实际添加后的文件列表如下：





最后，把协议栈的头文件路径添加到工程中：



至此，准备工作完毕。

### 3.2.2 配置协议栈

前面说过，sys\_config.h 这个文件提供了一系列的配置宏用于裁剪、配置协议栈。首先我们使能协议栈的 ppp，关闭以太网：

```
#define SUPPORT_PPP      1 /* 使能 ppp 栈
#define SUPPORT_ETHERNET 0 /* 禁止 ethernet
```

ppp 拨号相关配置信息：



```
#if SUPPORT_PPP

#define APN_DEFAULT          "4gnet"    /* 根据实际情况在这里设置缺省 APN */
#define AUTH_USER_DEFAULT    "card"     /* ppp 认证缺省用户名 */
#define AUTH_PASSWORD_DEFAULT "any_char" /* ppp 认证缺省口令 */

#define PPP_NETLINK_NUM      1 /* 最多支持几路 ppp 链路（系统存在几个 modem 这里就指定几就行） */
#define SUPPORT_ECHO          1 /* 对端是否支持 echo 链路探测 */
#define WAIT_ACK_TIMEOUT_NUM 5 /* 在这里指定连续几次接收不到对端的应答报文就进入协议栈故障处理流程（STACKFAULT），这
                                /* 意味着当前链路已经因严重故障终止了 */

#else

#define PPP_NETLINK_NUM 0

#endif
```

其中 APN\_DEFAULT 宏与 AUTH\_USER\_DEFAULT、AUTH\_PASSWORD\_DEFAULT 宏用于 ppp 拨号阶段，定义了 ppp 拨号时使用的缺省值。这个值仅在你没有配置拨号认证信息时使用（相关配置工作参见 3.2.4 节）。一般来说，如果你办的是移动或联通的普通 sim 卡，这几个宏的缺省值不用修改，直接适用。协议栈在设计上是支持多路 ppp 的，你的目标系统存在几个 4g 模块就将 PPP\_NETLINK\_NUM 宏调整为几。SUPPORT\_ECHO 宏与为你提供服务的移动运营商有关。从我的使用情况看，移动和联通基本都支持 echo 链路探测。如果你对流量及费用不那么在意，建议打开此功能。协议栈会每隔一小段时间周期性检查 ppp 链路是否正常。如果不正常，协议栈会自动重建 ppp 链路。WAIT\_ACK\_TIMEOUT\_NUM 用于指定 ppp 链路层通讯的重试次数。当 lcp、ipcp 层进行 ppp 链路协商时，如果未收到对端的应答，ppp 栈会自动重发。这个宏指定重发的次数。超过这个次数，ppp 栈会自动判定为协商故障，进入故障处理流程。这个值一般不需要修改，除非你想提升或降低协议栈对协商故障的灵敏度。

### 3.2.3 os 适配层——编写 tty 驱动

在 Linux 系统，4g 模块作为一个通讯终端，驱动层会把它当作一个 tty 设备来看待。Linux 下 ppp 栈也是围绕着一个标准的 tty 设备来实现底层通讯逻辑的。至于如何操作这个 4g 模块进行实际的数据收发，tty 层并不关心。所以，onps 栈完全借鉴了这个成功的设计思想，在底层驱动与 4g 模块之间增加了一个 tty 层，将具体的设备操作与上层的业务逻辑进行了剥离。os 适配层首先要做的就是完成 tty 层到底层驱动的封装工作。协议栈利用句柄来唯一的标识一个 tty 设备。在 os\_datatype.h 文件中定义了这个句柄类型：

```
#if SUPPORT_PPP

typedef INT HTTY;          /* tty 终端句柄 */
#define INVALID_HTTY -1    /* 无效的 tty 终端句柄 */

#endif
```

这个句柄类型非常重要，所有与 tty 操作相关的函数都要用到这个句柄类型。我们接下来要编写的 tty 驱动函数也是围绕着这个句柄进行的。打开 os\_adapter.h 文件，添加 tty 驱动函数的原型定义并声明为导出函数：

```
#if SUPPORT_PPP

/* 打开 tty 设备，返回 tty 设备句柄，参数 pszTTYName 指定要打开的 tty 设备的名称 */
OS_ADAPTER_EXT HTTY os_open_tty(const CHAR *pszTTYName);

/* 关闭 tty 设备，参数 hTTY 为要关闭的 tty 设备的句柄 */
OS_ADAPTER_EXT void os_close_tty(HTTY hTTY);

/* 向 hTTY 指定的 tty 设备发送数据，返回实际发送的数据长度 */
/* hTTY：设备句柄 */
```

```
/* pubData: 指针, 指向要发送的数据的指针
/* nDataLen: 要发送的数据长度
OS_ADAPTER_EXT INT os_tty_send(HTTY hTTY, UCHAR *pubData, INT nDataLen);

/* 从参数 hTTY 指定的 tty 设备等待接收数据, 阻塞型
/* hTTY: 设备句柄
/* pubRcvBuf: 指针, 指向数据接收缓冲区的指针, 用于保存收到的数据
/* nRcvBufLen: 接收缓冲区的长度
/* nWaitSecs: 等待的时长, 单位: 秒。0 一直等待; 直至收到数据或报错, 大于 0, 等待指定秒数; 小于 0, 不支持
OS_ADAPTER_EXT INT os_tty_recv(HTTY hTTY, UCHAR *pubRcvBuf, INT nRcvBufLen, INT nWaitSecs);

/* 复位 tty 设备, 这个函数名称体现了 4g 模块作为 tty 设备的特殊性, 其功能从本质上看就是一个 modem, modem 设备出现通讯
/* 故障时, 最好的修复故障的方式就是直接复位, 复位可以修复绝大部分的因软件问题产生的故障
OS_ADAPTER_EXT void os_modem_reset(HTTY hTTY);

#endif
```

转到 `os_adapter.c` 文件, 编码实现这几个 tty 函数:

```
#if SUPPORT_PPP

HTTY os_open_tty(const CHAR *pszTTYName)
{
    /* 避免编译器警告, 如果你的系统存在多个 4g 模块, 那么 pszTTYName 参数用于区分打开哪个串口
    pszTTYName = pszTTYName;

    /* 115200, 8 个有效数据位, 1 个停止位, 无奇偶校验, 无流控
    SCP3Init(115200, DATABIT_8, STOPBITS_1, PARITYTP_NO, FLOWCTLTP_NO);

    /* 由于我的目标系统只有一个 4g 模块, 所以这里返回的 tty 句柄为 0, 如果你的目标系统存在多个, 这里需要你根据参数
    /* pszTTYName 指定的名称来区分是哪个设备, 并据此返回不同的 tty 句柄, 句柄值应从 0 开始自增, 步长为 1
    return 0;
}

/* 在我的目标系统中不需要关闭 tty 设备, 所以这里没有添加任何逻辑代码
void os_close_tty(HTTY hTTY)
{
    hTTY = hTTY; /* 避免编译器警告
}

INT os_tty_send(HTTY hTTY, UCHAR *pubData, INT nDataLen)
{
    /* 避免编译器警告, 如果系统存在多个 tty 设备, 参数 hTTY 用于区分这些设备
    hTTY = hTTY;

    /* 调用对应的串口驱动函数, 发送数据到 4g 模块, 如果存在多个 tty 设备, 请依据参数 hTTY 来确定需要调用哪个
    /* 串口驱动函数发送数据
    return (INT)SCP3Send(pubData, nDataLen);
}
```

```
INT os_tty_recv(HTTY hTTY, UCHAR *pubRcvBuf, INT nRcvBufLen, INT nWaitSecs)
{
    hTTY = hTTY; /* 避免编译器警告 */
    return SCP3Recv(pubRcvBuf, (uint32_t)nRcvBufLen, nWaitSecs);
}

void os_modem_reset(HTTY hTTY)
{
#ifdef SUPPORT_PRINTF
    printf("start reset modem...\r\n");
#endif

    /* 重新初始化串口，主要工作就是清空 dma 缓冲区并重新初始化 dma 通道（具体内容参见 3.2.1 节） */
    SCP3ReInit();

    /* 给 4g 模块的复位引脚发送复位信号，如果你的目标板不支持软件复位 4g 模块，可以省略这一步 */
    M4G_POWER_OFF(); /* 拉低引脚，关闭模块 */
    os_sleep_secs(3); /* 关闭足够的时长 */
    M4G_POWER_ON(); /* 拉高引脚，开启模块 */
    os_sleep_secs(6); /* 同样等待一小段时间，确保模块已进入工作状态 */

#ifdef SUPPORT_PRINTF
    printf("reset modem succeeded.\r\n");
#endif
}
```

### 3.2.4 调整相关系统设置

os\_adapter.c 文件中关于 ppp 部分还有如下几项定义需要根据你的实际目标环境进行配置（参见原始的协议栈源码工程同名文件）：

```
.....

/* 增加 ppp 相关的头文件 */
#ifdef SUPPORT_PPP
#include "scp.h"
#include "ppp/negotiation_storage.h"
#include "ppp/ppp.h"
#endif
.....

#ifdef SUPPORT_PPP
/* 连接 4g 模块的串口名称，有几个 4g 模块，就指定几个，其存储的单元索引应等于 os_open_tty() 返回的对应串口的 tty 句柄值 */
const CHAR *or_psaTTY[PPP_NETLINK_NUM] = { "SCP3" };

/* 指定 ppp 拨号的 apn、用户和密码，系统支持几路 ppp，就需要指定几组拨号信息 */
/* ST_DIAL_AUTH_INFO 结构体保存这几个信息，该结构体的详细内容参见协议栈源码 ppp/ppp.h 文件 */
/* 这里设置的 apn 等拨号认证信息会替代前面说过的 APN_DEFAULT、AUTH_USER_DEFAULT、AUTH_PASSWORD_DEFAULT 等缺省设置 */
```

```

const ST_DIAL_AUTH_INFO or_staDialAuth[PPP_NETLINK_NUM] = {
    { "4gnet", "card", "any_char" },    /* 注意 ppp 账户和密码尽量控制在 20 个字节以内，太长需要需要修改 chap.c
                                           /* 中 send_response() 函数的 szData 数组容量及 pap.c 中 pap_send_auth_request() 函数的
                                           /* ubaPacket 数组的容量，确保其能够封装一个完整的响应报文

    /* 系统存在几路 ppp 链路，就在这里添加几路拨号认证信息 */
};

/* ppp 链路协商的初始协商配置信息，协商成功后这里保存最终的协商结果，ST_PPPNEGORESULT 结构体的详细说明参见下文
ST_PPPNEGORESULT o_staNegoResult[PPP_NETLINK_NUM] = {
    {
        { 0, PPP_MRU, ACCM_INIT, { PPP_CHAP, 0x05 /* CHAP 协议，0-4 未使用，0x05 代表采用 MD5 算法 */ }, TRUE, TRUE, FALSE, FALSE },
        { IP_ADDR_INIT, DNS_ADDR_INIT, DNS_ADDR_INIT, IP_ADDR_INIT, MASK_INIT }, 0
    },

    /* 系统存在几路 ppp 链路，就在这里添加几路的协商初始值，如果不确定，可以将上面预定义的初始值直接复制过来即可 */
};
#endif

```

上面给出的代码做了几件事情：

- 1) include 部分增加了 ppp 相关的头文件；
- 2) 指定 tty 设备连接的串口名称；
- 3) 指定拨号认证信息：apn、用户和密码；
- 4) 指定 ppp 链路协商初始值；

总之，你的目标系统连接了几个 4g 模块，第 2、3、4 件事情就要针对特定的模块分别做一遍，单独指定。这里需要重点说明的是 ppp 链路协商配置信息。这些信息由 ST\_PPPNEGORESULT 结构体保存（参见 negotiation\_storage.h 文件）：

```

typedef struct _ST_PPPNEGORESULT_ {
    struct {
        UINT unMagicNum; /* 幻数（魔术字）
        USHORT usMRU;    /* 最大接收单元，缺省值由 PPP_MRU 宏指定，一般为 1500 字节
        UINT unACCM;     /* ACCM，异步控制字符映射，指定哪些字符需要转义，如果不确定，建议采用 ACCM_INIT 宏指定的缺省值
        struct { /* 保存认证信息的结构体
            USHORT usType;    /* 指定认证类型：chap 或 pap，缺省 chap 认证
            UCHAR ubaData[16]; /* 认证报文携带的数据，不同的协议携带的数据类型不同，一般情况下采用协议栈的缺省值即可
        } stAuth;

        BOOL bIsProtoComp;      /* 是否采用协议域压缩（本地设置项，代表协议栈一侧，协商结果不影响该字段）
        BOOL bIsAddrCtlComp;    /* 是否采用地址及控制域压缩（本地设置项，代表协议栈一侧，协商结果不影响该字段）
        BOOL bIsNegoValOfProtoComp; /* 协议域是否压缩的协商结果值（远端设置项，代表对端是否支持该配置，协商结果影响该字段）
        BOOL bIsNegoValOfAddrCtlComp; /* 地址及控制域是否压缩的协商结果值（远端设置项，同上）
    } stLCP;

    /* 存储 ppp 链路的初始及协商成功后的地址信息
    struct {
        UINT unAddr;          /* ip 地址，初始值由协议栈提供的 IP_ADDR_INIT 宏指定，不要擅自修改
        UINT unPrimaryDNS;    /* 主 dns 服务器地址，初始值由协议栈提供的 DNS_ADDR_INIT 宏指定，不要擅自修改
        UINT unSecondaryDNS;  /* 次 dns 服务器地址，初始值由协议栈提供的 DNS_ADDR_INIT 宏指定，不要擅自修改
        UINT unPointToPointAddr; /* 点对点地址，初始值由协议栈提供的 IP_ADDR_INIT 宏指定，不要擅自修改
    }

```

```

    UINT unSubnetMask;        /* 子网掩码
} stIPCP;
UCHAR ubIdentifier;    /* 标识域，从 0 开始自增，唯一的标识一个 ppp 报文，用于确定应答报文
UINT unLastRcvdSecs; /* 最近一次收到对端报文时的秒数，其用于 ppp 链路故障探测，无需关心，协议栈底层使用
} ST_PPPNEGORESULT, *PST_PPPNEGORESULT;

```

基本上，要调整的地方几乎没有，我们直接采用缺省值即可。最后，我们还需要把 ppp 栈的主处理线程（任务）添加到 os 适配层的工作线程列表中。rt-thread 下相关代码如下：

```

.....
#if SUPPORT_PPP
#define THPPPHANDLER_PRIO      21      /* ppp 栈主处理线程优先级
#define THPPPHANDLER_STK_SIZE  384 * 4 /* ppp 栈主处理线程栈大小
#define THPPPHANDLER_TIMESLICE 10      /* ppp 栈主处理线程运行时间片
#endif
/* 协议栈内部工作线程列表
const static STCB_PSTACKTHREAD lr_stcbaPStackThread[] = {
    { thread_one_shot_timer_count,
      (void *)0,
      "OSTimerCnt",
      THOSTIMERCOUNT_STK_SIZE,
      THOSTIMERCOUNT_PRIO,
      THOSTIMERCOUNT_TIMESLICE },

#if SUPPORT_PPP
    { thread_ppp_handler,    /* ppp 栈主处理线程入口函数，协议栈内部之 ppp 栈工作线程
      (void *)0,
      "THPPPHandler",
      THPPPHANDLER_STK_SIZE,
      THPPPHANDLER_PRIO,
      THPPPHANDLER_TIMESLICE },

#endif
};
.....

```

ucos ii 下相关代码：

```

.....
#if SUPPORT_PPP
#define THPPPHANDLER_PRIO      21      /* ppp 栈主处理任务优先级
#define THPPPHANDLER_STK_SIZE  384      /* ppp 栈主处理任务栈大小
__align(8) OS_STK THPPPHANDLER_STK[THPPPHANDLER_STK_SIZE]; /* ppp 栈主处理任务栈
#endif
/* 协议栈内部工作线程列表
const static STCB_PSTACKTHREAD lr_stcbaPStackThread[] = {
    { thread_one_shot_timer_count,
      (void *)0,
      (OS_STK *)&THOSTIMERCOUNT_STK[THOSTIMERCOUNT_STK_SIZE - 1],
      THOSTIMERCOUNT_PRIO },

```

```
#if SUPPORT_PPP
{ thread_ppp_handler,
  (void *)0,
  (OS_STK *)&THPPHANDLER_STK[THPPHANDLER_STK_SIZE - 1],
  THPPHANDLER_PRIO },
#endif
};
.....
```

新添加的这个 ppp 栈主处理线程（任务）将在协议栈加载时由我们原先已经编码实现的 os 适配层函数 os\_thread\_onpstack\_start() 启动。我们只需将其添加到 os 适配层的工作线程列表中即可，剩下的交由协议栈自动处理。在这里需要特别说明的是传递给 ppp 栈主处理线程（任务）的启动参数——蓝色加粗部分 “(void \*)0”。这个值并不代表启动参数为 NULL，而是代表当前要建立的这个 ppp 链路连接的 tty 设备的句柄值。也就是说其值应与 os\_open\_tty() 返回的 tty 句柄值一致，即：

```
(void *)hTTY /* hTTY 实际是几这里就填几
```

不用关心前面的 “(void \*)”，这段数据类型强制转换代码只是为了避免编译器报错。ppp 链路建立成功后，协议栈会以 “ppp+tty 句柄” 的方式命名该链路，命名时的 tty 句柄值就是通过这个启动参数获得的，所以这个值一定要配置正确。对于[样例工程](#)，os\_open\_tty() 返回的 tty 句柄为 0，则 ppp 链路的名称为 “ppp0”。

至此，os 适配层的移植工作已全部完成。协议栈的移植工作也全部完成。测试之前，我们把协议栈加载到目标系统。首先把协议栈及网络测试工具的头文件引入到系统：

```
.....
#include "onps.h"
#include "net_tools/ping.h"
#include "net_tools/dns.h"
#include "net_tools/sntp.h"
.....
```

把协议栈加载代码添加到目标系统的主线程（任务）中，添加位置与以太网场景下一致，代码逻辑也基本相同：

```
.....
EN_ONPSERR enErr;
if(open_npstack_load(&enErr))
{
    printf("The open source network protocol stack is loaded successfully. \r\n");

    /* 等待 ppp 链路就绪后再启动其它线程（任务）
    while(!netif_is_ready("ppp0"))
        os_sleep_secs(1);
}
.....
```

由于 ppp 链路的建立需要点时间，且存在 ppp 链路建立失败的情形（信号差、欠费等），因此，我们在协议栈加载完毕后，增加了等待 ppp 链路就绪的环节。netif\_is\_ready() 是协议栈提供给用户用于判断指定网卡是否已就绪的接口函数。这个函数的入口参数只有一个——网络接口（网卡）名称。协议栈依据网卡名称查找并确定网卡是否已就绪，就绪则返回 TRUE，否则返回 FALSE。对于 ppp 栈，已经建立好的 ppp 链路本质上就是一个普通的网卡。我们在前面说过，样例工程中 ppp 网卡的名称为 “ppp0”。所以在这里我们指定 netif\_is\_ready() 函数的入口参数值为 “ppp0”，循环等待 ppp 链路直至建立成功，然后再启动相关测试线程（任务）。这个处理逻辑与前面以太网场景的处理逻辑完全相同。



如果你的移植步骤没有任何问题，并且 sys\_config.h 文件中 DEBUG\_LEVEL 的值大于 1，那么我们编译整个工程并下载到板子上之后，ppp 栈应能正常工作并在控制台输出建立 ppp 链路的详细过程：

```
COM3 - PuTTY
The open source network protocol stack is loaded successfully.
start Reset Modem...
reset modem succeeded.
AT
OK
ATDT
OK
+SYNTEST:3: 3
OK
+CHRG: 0,1
OK
OK
CONNECT 150000000
modem dial succeeded
sent [Protocol LCP, Id = 00, Code = 'Configure Request', MRU = 1500 Bytes, ACMD = 00000000, Protocol Field Compression, Address/Control Field Compression]
recv [Protocol LCP, Id = 00, Code = 'Configure Request', ACMD = 00000000, Authentication type = 'CHAP', Protocol Field Compression, Address/Control Field Compression]
sent [Protocol LCP, Id = 00, Code = 'Configure Ack']
recv [Protocol LCP, Id = 00, Code = 'Configure Ack']
use CHAP authentication, magic number <363459C0>
recv [Protocol CHAP, Id = 01, Code = 'Discard Request']
recv [Protocol CHAP, Id = 01, Code = 'Challenge', Challenge = <F1311533C174D48888D42F4754600ED5>, name = "UMTS_CHAP_SRVH"]
sent [Protocol CHAP, Id = 01, Code = 'Response', Challenge = <16D252518ABBEA2ACAC8C894A5AE51>, name = "card"]
recv [Protocol CHAP, Id = 01, Code = 'Success']
CHAP authentication succeeded.
sent [Protocol IPCP, Id = 01, Code = 'Configure Request', IP <0.0.0.0>, Primary DNS <0.0.0.0>, Secondary DNS <0.0.0.0>]
recv [Protocol IPCP, Id = 00, Code = 'Configure Request']sent [Protocol IPCP, Id = 00, Code = 'Configure ACK']
recv [Protocol IPCP, Id = 01, Code = 'Configure Nak', IP <10.104.250.14>, Primary DNS <211.137.191.27>, Secondary DNS <218.201.96.131>]
sent [Protocol IPCP, Id = 02, Code = 'Configure Request', IP <10.104.250.14>, Primary DNS <211.137.191.27>, Secondary DNS <218.201.96.131>]
recv [Protocol IPCP, Id = 02, Code = 'Configure Ack']
Local IP Address 10.104.250.14
Primary DNS Server 211.137.191.27
Secondary DNS Server 218.201.96.131
Connect: ppp0 <=> SCP3
<ppp0> added to the protocol stack
[inet 10.104.250.14, netmask 255.255.255.255, Point to Point 10.42.42.42, Primary DNS 211.137.191.27, Secondary DNS 218.201.96.131]
Add/Update network interface <ppp0> to routing table
[destination default, gateway 0.0.0.0, netmask 0.0.0.0]
```

3.2.5 ping 等网络工具测试

除 dhcp 客户端外，ping、dns 客户端、sntp 客户端这三个网络工具均可以正常使用。这三个工具的测试代码与前面以太网场景下的测试代码完全相同。只不过在 ppp 场景下，为了提高代码的可读性，我把它三个挪到了一个测试线程（任务）里（以 rt-thread 为例，ucosii 就不再给出了）：

```
.....

/* 测试代码使能配置宏
//=====
#define PING_TEST_EN      1 /* ping 测试使能
#define DNS_QUERY_TEST_EN 1 /* dns 查询测试使能
#define SNTP_TEST_EN      1 /* sntp 网络校时测试使能
//=====

#if PING_TEST_EN || DNS_QUERY_TEST_EN || SNTP_TEST_EN
#define THNETTOOLSTEST_Prio      31
#define THNETTOOLSTEST_STK_SIZE  256 * 4
#define THNETTOOLSTEST_TIMESLICE 10
#endif
.....

#if PING_TEST_EN || DNS_QUERY_TEST_EN || SNTP_TEST_EN
static void THNetToolsTest(void *pvUserParam)
```



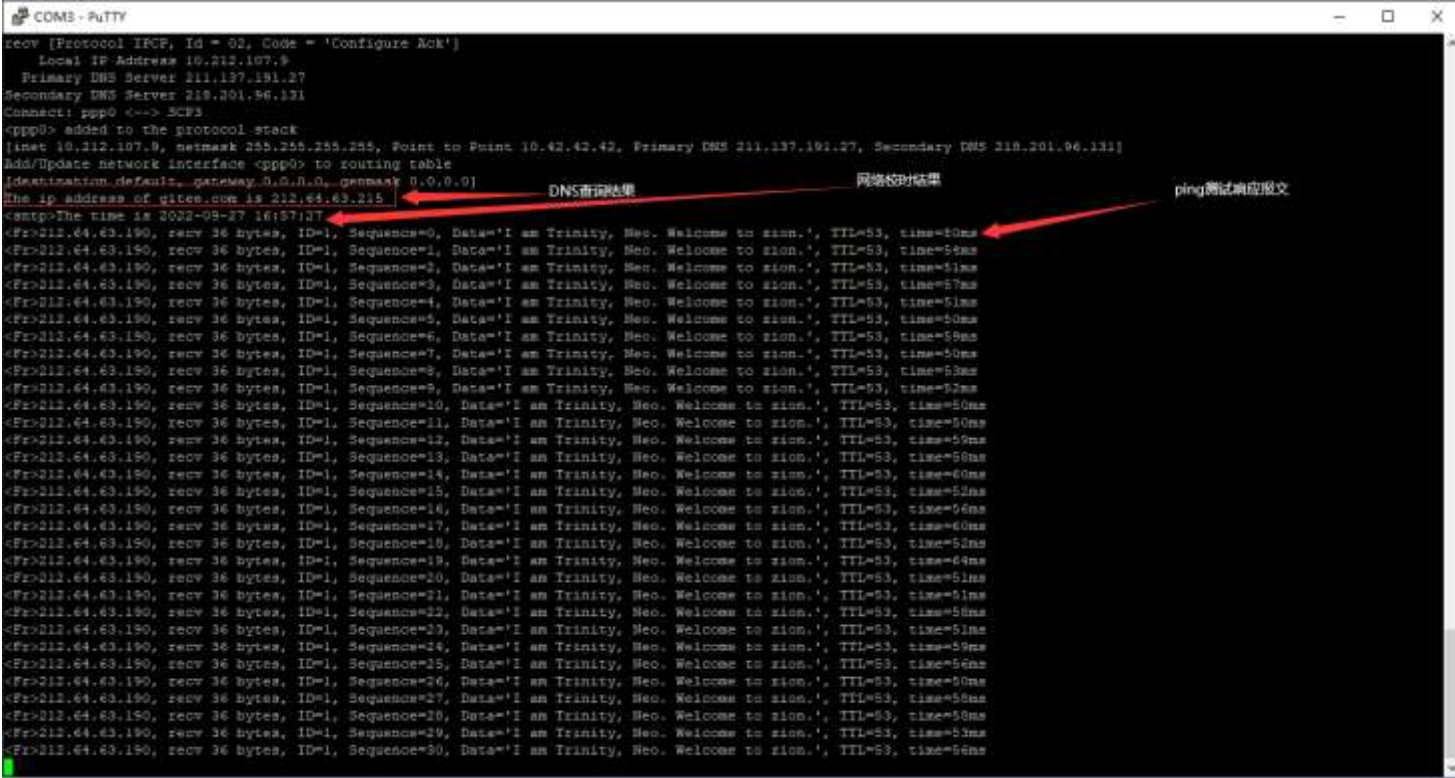
```
{
    EN_ONPSERR enErr;

#if DNS_QUERY_TEST_EN
    ..... /* 参见 3.1.8 节
#endif

#if SNTP_TEST_EN
    ..... /* 参见 3.1.9 节
#endif

#if PING_TEST_EN
    ..... /* 参见 3.1.7 节
#endif
}
```

详细的实现代码请从 [gitee 样例工程](#) 直接获取。这三个测试工具的控制台输出如下：



### 3.2.6 tcp 及 udp 客户端测试

这里的 tcp、udp 客户端测试惟一需要注意的是我们无法把 tcp 测试服务器部署到局域网内，必须部署到有固定 ip 地址的互联网上。除此之外，测试代码及测试方法与以太网场景完全相同，不再赘述。为了测试 ppp 链路的健壮性，样例工程建立了三个 tcp 客户端、一个 udp 客户端，共 4 个测试线程（任务）：

```
/* 以 rt-thread 为例，ucosii 类似
int main(void)
{
```

```
.....

/* 协议栈加载成功且 ppp 链路建立后启动工作线程
tid = rt_thread_create("THTcpCommu1", /* tcp 测试客户端 1
                        THTcpCommu1,
                        RT_NULL,
                        THTCPMMU1_STK_SIZE,
                        THTCPMMU1_PRI0,
                        THTCPMMU1_TIMESLICE);

if(tid != RT_NULL)
    rt_thread_startup(tid);

tid = rt_thread_create("THTcpCommu2", /* tcp 测试客户端 2
                        THTcpCommu2,
                        RT_NULL,
                        THTCPMMU2_STK_SIZE,
                        THTCPMMU2_PRI0,
                        THTCPMMU2_TIMESLICE);

if(tid != RT_NULL)
    rt_thread_startup(tid);

tid = rt_thread_create("THUdpCommu", /* udp 测试客户端
                        THUdpCommu,
                        RT_NULL,
                        THUdPCOMMU_STK_SIZE,
                        THUdPCOMMU_PRI0,
                        THUdPCOMMU_TIMESLICE);

if(tid != RT_NULL)
    rt_thread_startup(tid);
.....
while(TRUE)
{
    ..... /* tcp 测试客户端
}
}
```

详细的实现代码参见 gitee 上的移植[样例工程](#)。样例工程用到的目标 tcp、udp 测试服务器部署在了我自己购买的带宽为 1Mbps 的阿里云服务器上☺。控制台的日志输出如下：

```
COM3 - PuTTY
U#2022-09-29 15:26:01#>recv 20 bytes, Data = <http://www.cmsoft.cn>
1#2022-09-29 15:26:03#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 2, data length is 1128 bytes
U#2022-09-29 15:26:11#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:26:21#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:26:31#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:26:41#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:26:51#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:27:01#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:27:11#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:27:21#>recv 20 bytes, Data = <http://www.cmsoft.cn>
2#2022-09-29 15:27:27#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 1, data length is 1128 bytes
0#2022-09-29 15:27:31#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 0, data length is 1128 bytes
U#2022-09-29 15:27:31#>recv 20 bytes, Data = <http://www.cmsoft.cn>
1#2022-09-29 15:27:38#>recvd the control cmd packet, cmd = 0x01, LinkIdx = 2, data length is 1128 bytes
U#2022-09-29 15:27:41#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:27:51#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:28:01#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:28:11#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:28:21#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:28:31#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:28:41#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:28:51#>recv 20 bytes, Data = <http://www.cmsoft.cn>
U#2022-09-29 15:29:01#>recv 20 bytes, Data = <http://www.cmsoft.cn>
```

与以太网场景的日志输出相同，前缀为“U#”的日志为 udp 客户端收到数据输出，其余为三个 tcp 客户端（前缀为几就代表这是第几个 tcp 客户端）收到的服务器随机下发的控制指令的输出。部署在阿里云上的 tcp 测试服务器的控制台输出如下：

```
1#2022-09-29 15:32:23#>sent control command to peer, cmd = 0x01, ClientID = 2, the data length is 1128 bytes
0#2022-09-29 15:32:22#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6755, the data length is 1100 bytes
1#2022-09-29 15:32:22#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6788, the data length is 900 bytes
2#2022-09-29 15:32:22#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6771, the data length is 900 bytes
1#2022-09-29 15:32:22#>recvd acknowledge packet, AckedLinkIdx = 1, ClientID = 2, AckedTimestamp <2022-09-29 15:32:23>
0#2022-09-29 15:32:23#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6756, the data length is 1100 bytes
2#2022-09-29 15:32:23#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6772, the data length is 900 bytes
1#2022-09-29 15:32:23#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6789, the data length is 900 bytes
0#2022-09-29 15:32:24#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6757, the data length is 1100 bytes
2#2022-09-29 15:32:24#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6773, the data length is 900 bytes
1#2022-09-29 15:32:24#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6790, the data length is 900 bytes
0#2022-09-29 15:32:26#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6758, the data length is 1100 bytes
2#2022-09-29 15:32:26#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6774, the data length is 900 bytes
1#2022-09-29 15:32:26#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6791, the data length is 900 bytes
0#2022-09-29 15:32:28#>sent control command to peer, cmd = 0x01, ClientID = 0, the data length is 1128 bytes
0#2022-09-29 15:32:26#>recvd acknowledge packet, AckedLinkIdx = 0, ClientID = 0, AckedTimestamp <2022-09-29 15:32:28>
0#2022-09-29 15:32:26#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6759, the data length is 1100 bytes
2#2022-09-29 15:32:27#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6775, the data length is 900 bytes
1#2022-09-29 15:32:27#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6792, the data length is 900 bytes
0#2022-09-29 15:32:28#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6760, the data length is 1100 bytes
2#2022-09-29 15:32:28#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6776, the data length is 900 bytes
1#2022-09-29 15:32:28#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6793, the data length is 900 bytes
0#2022-09-29 15:32:29#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6761, the data length is 1100 bytes
2#2022-09-29 15:32:29#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6777, the data length is 900 bytes
1#2022-09-29 15:32:29#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6794, the data length is 900 bytes
0#2022-09-29 15:32:30#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6762, the data length is 1100 bytes
2#2022-09-29 15:32:30#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6778, the data length is 900 bytes
1#2022-09-29 15:32:31#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6795, the data length is 900 bytes
0#2022-09-29 15:32:31#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6763, the data length is 1100 bytes
2#2022-09-29 15:32:32#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6779, the data length is 900 bytes
1#2022-09-29 15:32:32#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6796, the data length is 900 bytes
0#2022-09-29 15:32:33#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6764, the data length is 1100 bytes
2#2022-09-29 15:32:34#>sent control command to peer, cmd = 0x01, ClientID = 1, the data length is 1128 bytes
2#2022-09-29 15:32:33#>recvd the uploaded packet, cmd = 0x00, ClientID = 1, SeqNum = 6780, the data length is 900 bytes
1#2022-09-29 15:32:33#>recvd the uploaded packet, cmd = 0x00, ClientID = 2, SeqNum = 6797, the data length is 900 bytes
2#2022-09-29 15:32:33#>recvd acknowledge packet, AckedLinkIdx = 2, ClientID = 1, AckedTimestamp <2022-09-29 15:32:34>
0#2022-09-29 15:32:34#>recvd the uploaded packet, cmd = 0x00, ClientID = 0, SeqNum = 6765, the data length is 1100 bytes
```

最后多说一句，栈空间的大小是系统稳定的关键因素之一，样例工程各线程（任务）使用的栈空间经过了反复测试调整，目前的大小能够满足当前相对复杂的测试需求。如果你在测试过程中出现莫名其妙的问题，请先确定是否是因栈空间过小导致的问题。

## 4 socket 函数使用说明

如前所述，协议栈提供的伯克利套接字（Berkeley sockets）并不是严格按照传统 socket 标准设计实现的，而是我根据以往 socket 编程经验，以方便用户使用、简化用户编码为设计目标，重新声明并定义的一组常见 socket 接口函数。协议栈简化了传统 BSD socket 编程需要的一些繁琐操作，将一些不必要的操作细节改为底层实现，比如 select/poll 模型、阻塞及非阻塞读写操作等。简化并不意味着推翻，socket 接口函数的基本定义、主要参数、使用方法并没有改变，你完全可以根据以往经验快速上手并熟练使用 onps 栈 sockets。这一点相信你已经从前面的测试代码中得到了佐证。

socket 层所有接口函数的实现源码被封装在了单独的一个文件中，参见 `bsd/socket.c`。其对应的头文件 `socket.h` 有这些接口函数的定义和说明。目前协议栈提供的 socket 接口函数有十几个，能够满足绝大部分应用场景的需求。这些接口函数如下：

- `socket`: 创建一个 socket，目前仅支持 `udp` 和 `tcp` 两种类型
- `close`: 关闭一个 socket，释放当前占用的协议栈资源
- `connect`: 与目标 `tcp` 服务器建立连接（阻塞型）或绑定一个固定的 `udp` 服务器地址
- `connect_ext`: 功能同 `connect()`，只是入口参数的服务器地址部分有所区别
- `connect_nb`: 与目标 `tcp` 服务器建立连接（非阻塞型）
- `connect_nb_ext`: 功能同 `connect_nb()`，同样只是入口参数的服务器地址部分有所区别
- `is_tcp_connected`: 获取当前 `tcp` 链路的连接状态
- `send`: 数据发送函数，`tcp` 链路下为阻塞型
- `send_nb`: 数据发送函数，非阻塞型
- `is_tcp_send_ok`: 数据是否已成功送达 `tcp` 链路的对端（收到 `tcp ack` 报文）
- `sendto`: `udp` 数据发送函数，发送数据到指定目标地址
- `recv`: 数据接收函数，`udp/tcp` 链路通用
- `recvfrom`: 数据接收函数，用于 `udp` 链路，接收数据的同时函数会返回数据源的地址信息
- `socket_set_rcv_timeout`: 设定 `recv()` 函数接收等待的时长，单位：秒
- `bind`: 绑定一个固定端口、地址
- `listen`: `tcp` 服务器进入监听状态
- `accept`: 接受一个到达的 `tcp` 连接请求
- `tcpsrv_recv_poll`: `tcp` 服务器专用函数，等待任意一个或多个 `tcp` 客户端数据到达信号
- `tcpsrv_set_recv_mode`: 设置服务器的接收模式，可以选择采用 `poll` 模型或 `active` 之一
- `tcpsrv_start`: 启动 `tcp` 服务器
- `tcp_srv_connect`: 连接 `tcp` 服务器
- `tcp_send`: `tcp` 数据发送函数
- `socket_get_last_error`: 获取 socket 最近一次发生的错误信息
- `socket_get_last_error_code`: 获取 socket 最近一次发生的错误编码

## socket

### 功能

创建一个 socket，支持 `udp` 和 `tcp` 两种类型，即：`SOCK_DGRAM` 和 `SOCK_STREAM`。注意，不使用时一定要调用 `close()` 函数关闭，以释放其占用的协议栈相关资源。

### 原型

```
SOCKET socket(INT family, INT type, INT protocol, EN_ONPSERR *penErr);
```

## 入口参数

- family: 目前仅支持 IPv4 及 IPv6, 取值为 AF\_INET 或 AF\_INET6, 前者为 IPv4 后者为 IPv6
- type: 指定 socket 类型, 支持 SOCK\_STREAM 和 SOCK\_DGRAM 两种类型, 前者为 tcp, 后者为 udp
- protocol: 未使用, 固定为 0
- penErr: 指向错误编码的指针, 当 socket() 函数执行失败, 该参数用于接收实际的错误码

## 返回值

执行成功返回 socket 句柄, 失败返回 INVALID\_SOCKET, 具体的错误信息参见 penErr 参数返回的错误码。

## 示例

```
EN_ONPSERR enErr;

/* tcp
SOCKET hSocket = socket(AF_INET, SOCK_STREAM, 0, &enErr);
if(INVALID_SOCKET == hSocket) /* 返回一个无效的 socket
    printf("%s\r\n", onps_error(enErr)); /*打印错误信息

/* udp
SOCKET hSocket = socket(AF_INET, SOCK_DGRAM, 0, &enErr);
.....
```

## close

### 功能

关闭 socket, 释放占用的协议栈资源。

### 原型

```
void close(SOCKET socket);
```

## 入口参数

- socket: 要关闭的 socket 句柄

## 返回值

无

## 示例

```
EN_ONPSERR enErr;

SOCKET hSocket = socket(AF_INET, SOCK_STREAM, 0, &enErr);
.....
if(INVALID_SOCKET != hSocket)

    close(hSocket);
```

# connect

## 功能

用于 tcp 类型的 socket 时，其功能为与目标服务器建立 tcp 连接，阻塞型

用于 udp 类型的 socket 时，其功能为绑定一个固定的目标通讯地址，udp 通讯均与这个固定地址进行

## 原型

```
INT connect(SOCKET socket, const CHAR *srv_ip, USHORT srv_port, INT nConnTimeout);
```

## 入口参数

- socket: 要进行 connect 操作的 socket 句柄
- srv\_ip: 目标服务器地址
- srv\_port: 目标服务器端口
- nConnTimeout: 仅用于 tcp 通讯，指定连接超时时间（单位：秒），参数值如小于等于 0 则协议栈会采用缺省值，该值由 TCP\_CONN\_TIMEOUT 宏指定（参见 sys\_config.h）；udp 通讯未使用这个参数，可以指定任意一个值

## 返回值

0: 连接成功

-1: 连接失败，具体的错误信息通过 onps\_get\_last\_error() 获得

## 示例

```
EN_ONPSERR enErr;

SOCKET hSocket = socket(AF_INET, SOCK_STREAM, 0, &enErr);
if(INVALID_SOCKET == hSocket)
{
    printf("%s\r\n", onps\_error(enErr));
    return;
}

if(!connect(hSocket, "47.92.239.107", 6410, 10))
{
    /* 连接成功，在这里添加你的自定义代码
    .....
}
else
{
    /* 连接失败，打印错误信息
    printf("%s\r\n", onps_get_last_error(hSocket, NULL));
}
.....

close(hSocket);
```



## connect\_ext

### 功能

功能同 connect(), 入口参数中服务器地址为 inet\_addr/inet6\_aton 函数转换后的 16 进制地址, 与 connect() 函数直接传入字符串形式的地址有区别。

### 原型

```
INT connect_ext(SOCKET socket, void *srv_ip, USHORT srv_port, INT nConnTimeout);
```

### 入口参数

- socket: 要进行 connect 操作的 socket 句柄
- srv\_ip: 目标服务器地址
- srv\_port: 目标服务器端口
- nConnTimeout: 仅用于 tcp 通讯, 指定连接超时时间 (单位: 秒), 参数值如小于等于 0 则协议栈会采用缺省值, 该值由 TCP\_CONN\_TIMEOUT 宏指定 (参见 sys\_config.h); udp 通讯未使用这个参数, 可以指定任意一个值

### 返回值

- 0: 连接成功
- 1: 连接失败, 具体的错误信息通过 onps\_get\_last\_error() 获得

### 示例

```
.....  
  
/* 连接 ipv4 服务器  
connect_ext(hSocket, inet_addr("47.92.239.107"), 6410, 10);  
  
  
/* 连接 ipv6 服务器  
UCHAR ubaDstAddr[16];  
connect_ext(hSocket, inet6_aton("2408:8214:41a:f29:5e13:5f57:925b:dec", ubaDstAddr), 6410, 10);  
.....
```

## connect\_nb

### 功能

仅用于 tcp 类型的 socket, 非阻塞型, 与目标 tcp 服务器建立连接。

### 原型

```
INT connect_nb(SOCKET socket, const CHAR *srv_ip, USHORT srv_port);
```

### 入口参数

- socket: 要进行 connect 操作的 socket 句柄
- srv\_ip: 目标服务器地址
- srv\_port: 目标服务器端口

## 返回值

- 0: 连接成功
- 1: 连接中
- 1: 连接失败, 具体的错误信息通过 `onps_get_last_error()` 获得

## 示例

```
EN_ONPSERR enErr;

SOCKET hSocket = socket(AF_INET, SOCK_STREAM, 0, &enErr);
if(INVALID_SOCKET == hSocket)
{
    printf("%s\r\n", onps\_error(enErr));
    return;
}

/* 循环等待 tcp 连接成功
while(1)
{
    INT nRtnVal = connect\_nb(hSocket, "47.92.239.107", 6410);
    if(!nRtnVal)
    {
        /* 连接成功, 在这里增加你的自定义代码
        .....

        break; /* 退出循环, 不再轮询检查 tcp 连接进程
    }
    else if(nRtnVal < 0)
    {
        /* 连接失败, 打印错误信息并退出循环不再轮询检查 tcp 连接进程
        printf("%s\r\n", onps\_get\_last\_error(hSocket, NULL));
        break;
    }
    else;

    /* 连接中, tcp 三次握手操作尚未完成, 此时你可以干点别的事情, 或者延时一小段时间后继续检查当前连接状态
    .....

    os\_sleep\_secs(1);
}
.....

close(hSocket);
```

## connect\_nb\_ext

### 功能

功能同 `connect_nb()` 函数, 只不过入口参数中服务器地址为 `inet_addr/inet6_aton` 函数转换后的 16 进制地址。

### 原型

```
INT connect_nb_ext(SOCKET socket, void *srv_ip, USHORT srv_port);
```

## 入口参数

- socket: 要进行 connect 操作的 socket 句柄
- srv\_ip: 目标服务器地址
- srv\_port: 目标服务器端口

## 返回值

- 0: 连接成功
- 1: 连接中
- 1: 连接失败, 具体的错误信息通过 onps\_get\_last\_error() 获得

## 示例

略

# is\_tcp\_connected

## 功能

检查 tcp 链路是否处于连接状态。

## 原型

```
INT is_tcp_connected(SOCKET socket, EN_ONPSERR *penErr);
```

## 入口参数

- socket: socket 句柄
- penErr: 指向错误编码的指针, 函数执行失败时该参数用于接收实际的错误码

## 返回值

- 0: 未连接
- 1: 已连接
- 1: 函数执行失败, 具体的错误信息通过参数 penErr 获得

## 示例

```
EN_ONPSERR enErr;

.....

INT nRtnVal = is_tcp_connected(hSocket, &enErr);
if(nRtnVal > 0)
    printf("已连接\r\n");
else if(!nRtnVal)
    printf("未连接\r\n");
else
    printf("检查失败, %s\r\n", onps_error(enErr));
.....
```

# send

## 功能

发送数据到目标地址。注意 tcp 链路下为阻塞型,直至收到对端的 tcp 层 ack 报文或超时才会返回(详见 3.1.10 节 [send\(\) 函数底层运行机制](#)的详细描述)。udp 链路下为非阻塞型,且只有在调用 connect() 函数后才能使用这个函数。

## 原型

```
INT send(SOCKET socket, UCHAR *pubData, INT nDataLen, INT nWaitAckTimeout);
```

## 入口参数

- socket: socket 句柄
- pubData: 指向要发送的数据的指针
- nDataLen: 要发送的数据的长度,单位: 字节
- nWaitAckTimeout: 仅用于 tcp 链路,指定发送超时时间,单位: 秒,如参数值不大于 0,则协议栈采用系统缺省值,该值由 TCP\_ACK\_TIMEOUT 宏指定(参见 sys\_config.h); udp 链路未使用,可指定任意值

## 返回值

发送成功,则返回值等于 nDataLen; 发送失败,返回值不等于 nDataLen,具体的错误信息通过 onps\_get\_last\_error() 获得。

## 示例

```
/* tcp 链路下 send() 函数使用示例 */

EN_ONPSERR enErr;

.....

UCHAR ubUserData[128];
INT nSndBytes, nSndNum = 0;

__lblSend:
if(nSndNum > 2)
{
    /** 超出重传次数,不再重传,可以关闭当前 tcp 链路重连 tcp 服务器或者你自己的其它处理方式
    .....
    return;
}
nSndBytes = send(hSocket, ubUserData, sizeof(ubUserData), 3);
if(sizeof(ubUserData) == nSndBytes)
{
    /** 发送成功,在这里添加你自己的处理代码
    .....
}
else
{
    const CHAR *pszErr = onps_get_last_error(hSocket, &enErr);
    if(enErr == ERRTCPACKTIMEOUT) /** 等待 tcp 层的 ack 报文超时
```

```
{  
    /* 数据重传，用户层实现 tcp 层重传机制  
    nSndNum++;  
    goto __lblSend;  
}  
else /* 其它错误，意味着底层协议栈捕捉到了内存不够用、网卡故障等类似的严重问题  
{  
    /* 没必要触发重传机制了，根据你自己的具体情形增加容错处理代码并打印错误信息  
    .....  
    printf("发送失败，%s\r\n", pszErr);  
}  
}  
.....
```

## send\_nb

### 功能

发送数据到目标地址，非阻塞型，其它与 send 函数完全相同。

### 原型

```
INT send_nb(SOCKET socket, UCHAR *pubData, INT nDataLen);
```

### 入口参数

- socket: socket 句柄
- pubData: 指向要发送的数据的指针
- nDataLen: 要发送的数据的长度，单位：字节

### 返回值

发送成功，返回值等于 nDataLen；返回值为 0，上一组数正处于发送中（尚未收到对端的 tcp ack 报文），需要等待其发送成功后再发送当前数据；发送失败，返回值小于等于 0，具体的错误信息通过 onps\_get\_last\_error() 获得。

### 示例

```
/* tcp 链路下 send_nb() 函数使用示例 */  
  
EN_ONPSERR enErr;  
  
.....  
  
UCHAR ubUserData[128];  
INT nRtnVal;  
  
__lblSend:  
nRtnVal = send_nb(hSocket, ubUserData, sizeof(ubUserData));  
if(sizeof(ubUserData) == nRtnVal)  
{  
    /* 调用 is\_tcp\_send\_ok\(\) 函数等待是否已成功送达对端，或者（同时）做点别的事情
```

```
.....
}
else if(0 == nRtnVal)
{
    /* 上一组数据尚未发送完毕，需要等待发送完毕后再发送当前数据，等待期间你可以在这里做点别的事情
    .....
    goto __lblSend;
}
else
{
    /* 发送失败，协议栈底层捕捉到了一个严重的系统错误，这里增加你的容错代码并打印错误信息，不再继续发送
    .....
    printf("发送失败，%s\r\n", onps_get_last_error(hSocket, NULL));
}
.....
```

## is\_tcp\_send\_ok

### 功能

非阻塞型，数据是否已成功送达 tcp 链路的对端（已收到对端回馈的 tcp ack 报文）。

### 原型

```
INT is_tcp_send_ok(SOCKET socket);
```

### 入口参数

➤ socket: socket 句柄

### 返回值

0: 发送中

1: 发送成功

-1: 发送失败，具体错误信息通过 onps\_get\_last\_error() 函数获得

### 示例

```
EN_ONPSERR enErr;

.....

UCHAR ubUserData[128];
INT nRtnVal;

__lblSend:
nRtnVal = send_nb(hSocket, ubUserData, sizeof(ubUserData));
if(sizeof(ubUserData) == nRtnVal)
{
    /* 数据已通过网卡成功送出，接下来轮询等待对端回馈的 tcp ack 报文，确保数据成功送达对端
    while(1)
```



```
{
    INT nResult = is_tcp_send_ok(hSocket);
    if(nResult == 1)
    {
        /* 发送成功了，退出轮询等待
        break;
    }
    else if(nResult < 0)
    {
        /* 协议栈底层捕捉到了一个严重的系统错误，不再轮询等待，并打印错误信息
        .....
        printf("%s\r\n", onps_get_last_error(hSocket, NULL));
        break;
    }

    /* 发送中，在这里你可以做点别的事情
    .....
}

.....
}
else if(0 == nRtnVal)
{
    /* 上一组数据尚未发送完毕，需要等待发送完后再发送当前数据，等待期间你可以在这里做点别的事情
    .....
    goto __lblSend;
}
else
{
    /* 发送失败，协议栈底层捕捉到了一个严重的系统错误，这里打印错误信息，不再继续发送
    printf("发送失败, %s\r\n", onps_get_last_error(hSocket, NULL));
}
}
.....
```

## sendto

### 功能

非阻塞型，仅用于 udp 通讯，发送数据到指定的目标地址。

### 原型

```
INT sendto(SOCKET socket, const CHAR *dest_ip, USHORT dest_port, UCHAR *pubData, INT nDataLen);
```

### 入口参数

- socket: socket 句柄
- dest\_ip: 目标地址

- dest\_port: 目标端口
- pubData: 指向要发送的数据的指针
- nDataLen: 要发送的数据的长度, 单位: 字节

## 返回值

发送成功, 返回值等于 nDataLen, 反之则发送失败, 具体的错误信息通过 onps\_get\_last\_error() 获得。

## 示例

```
EN_ONPSERR enErr;

.....

UCHAR ubUserData[128];
INT nSndBytes = sendto(hSocket, "47.92.239.107", 6411, ubUserData, sizeof(ubUserData));
if(sizeof(ubUserData) == nSndBytes)
{
    /* 发送成功, 在这里增加你自己的业务代码
    .....
}
else
{
    /* 发送失败, 在这里增加你自己的容错代码并打印错误信息
    .....

    printf("发送失败, %s\r\n", onps_get_last_error(hSocket, NULL));
}

.....
```

## recv

### 功能

读取链路对端发送的数据。其阻塞类型取决于 socket\_set\_rcv\_timeout() 函数设定的接收等待时长。缺省为阻塞型, 一直等待直至收到数据或报错。

### 原型

```
INT recv(SOCKET socket, UCHAR *pubDataBuf, INT nDataBufSize);
```

### 入口参数

- socket: socket 句柄
- pubDataBuf: 指向数据接收缓冲区的指针
- nDataBufSize: 数据接收缓冲区的大小, 单位: 字节

### 返回值

大于等于 0 为实际到达的数据长度, 单位: 字节; 小于 0, 接收失败, 具体的错误信息通过 onps\_get\_last\_error() 获得。

### 示例

```
EN_ONPSERR enErr;

.....

UCHAR ubRcvBuf[1500];
INT nRcvBytes = recv(hSocket, ubRcvBuf, sizeof(ubRcvBuf));
if(nRcvBytes > 0) /* 收到数据
{
    .....
}
else
{
    if(nRcvBytes < 0) /* 协议栈底层捕捉到了一个严重错误，在这里增加你的容错代码并打印错误信息
    {
        .....
        printf("%s\r\n", onps_get_last_error(hSocket, NULL));
    }
}
.....
```

## socket\_set\_rcv\_timeout

### 功能

设定 `recv()` 函数接收等待的时长。其设定的接收等待时长决定了 `recv()` 函数的阻塞类型。

等于 0：非阻塞，`recv()` 不等待立即返回

大于 0：阻塞，`recv()` 等待指定时长直至数据到达或超时

小于 0：阻塞，`recv()` 一直等待直至数据到达或出错

### 原型

```
BOOL socket_set_rcv_timeout(SOCKET socket, CHAR bRcvTimeout, EN_ONPSERR *penErr);
```

### 入口参数

- `socket`: `socket` 句柄
- `bRcvTimeout`: `recv()` 函数的接收等待时长，单位：秒
- `penErr`: 指向错误编码的指针，函数执行失败时该参数用于接收实际的错误码

### 返回值

设置成功返回 TRUE，否则返回 FALSE，具体的错误信息通过参数 `penErr` 获得

### 示例

```
EN_ONPSERR enErr;

.....

if(!socket_set_rcv_timeout(hSocket, 1, &enErr))
{
```

```
/* 设置失败，打印错误信息，此时系统采用缺省值，即 recv() 函数一直等待直至收到数据或协议栈报错
printf("%s\r\n", onps\_error(enErr));
}
.....
```

## recvfrom

### 功能

接收数据并返回数据源的地址信息，仅用于 udp 通讯。

### 原型

```
INT recvfrom(SOCKET socket, UCHAR *pubDataBuf, INT nDataBufSize, void *pvFromIP, USHORT *pusFromPort);
```

### 入口参数

- socket: socket 句柄
- pubDataBuf: 指向数据接收缓冲区的指针
- nDataBufSize: 数据接收缓冲区的大小，单位：字节
- pvFromIP: 指向数据源地址（ipv4/ipv6）的指针
- pusFromPort: 指向数据源端口的指针

### 返回值

实际收到的数据的长度，单位：字节；小于 0 则接收失败，具体的错误信息通过 `onps_get_last_error()` 获得。

### 示例

```
EN_ONPSERR enErr;

.....

UCHAR ubRcvBuf[512];
in_addr_t unFromIp;
USHORT usFromPort;
INT nRcvBytes = recvfrom(hSocket, ubRcvBuf, sizeof(ubRcvBuf), &unFromIp, &usFromPort);
if(nRcvBytes > 0) /* 收到数据
{
    CHAR szAddr[20];
    printf("收到来自地址%s:%d 的%d 字节的数据\r\n", inet\_ntoa\_safe\_ext(unFromIp, szAddr), usFromPort, nRcvBytes);
    .....
}
else
{
    if(nRcvBytes < 0) /* 协议栈底层捕捉到了一个严重错误，在这里增加你的容错代码并打印错误信息
    {
        .....

        printf("发送失败，%s\r\n", onps\_get\_last\_error(hSocket, NULL));
    }
}
```

.....

## bind

### 功能

绑定一个 ip 地址和端口。

### 原型

```
INT bind(SOCKET socket, const CHAR *pszNetifIp, USHORT usPort);
```

### 入口参数

- socket: socket 句柄
- pszNetifIp: 指向要绑定的 ip 地址的指针, 为 NULL 绑定任意网络接口
- usPort: 要绑定的端口

### 返回值

- 0: 成功
- 1: 失败, 具体的错误信息通过 onps\_get\_last\_error() 获得

### 示例

```
EN_ONPSERR enErr;

.....

SOCKET hSockSrv = socket(AF_INET, SOCK_STREAM, 0, &enErr);
if(INVALID_SOCKET != hSockSrv)
{
    if(!bind(hSockSrv, NULL, 6411)) /* 绑定成功
        .....
    else /* 绑定失败
        printf("%s\r\n", onps_get_last_error(hSocket, NULL)); /* 打印错误信息
}
else
    printf("%s\r\n", onps_error(enErr)); /*打印错误信息
.....
```

## listen

### 功能

tcp 服务器进入监听状态, 等待 tcp 客户端连接请求的到达。

### 原型

```
INT listen(SOCKET socket, USHORT backlog);
```

## 入口参数

- socket: socket 句柄
- backlog: 等待用户层接受 (accept) 连接请求的 tcp 客户端数量

## 返回值

- 0: 成功
- 1: 失败, 具体的错误信息通过 onps\_get\_last\_error() 获得

## 示例

```
EN_ONPSERR enErr;

.....

SOCKET hSockSrv = socket(AF_INET, SOCK_STREAM, 0, &enErr);
if(INVALID_SOCKET != hSockSrv)
{
    if(!bind(hSockSrv, NULL, 6411)) /* 绑定成功
    {
        if(!listen(hSockSrv, usBacklog)) /* 进入监听状态
        {
            .....
        }
    }
    else /* 失败
        printf("%s\r\n", onps_get_last_error(hSocket, NULL)); /* 打印错误信息
}
else /* 绑定失败
    printf("%s\r\n", onps_get_last_error(hSocket, NULL)); /* 打印错误信息
}
else
    printf("%s\r\n", onps_error(enErr)); /*打印错误信息
.....
```

## accept

### 功能

阻塞/非阻塞型, 接受一个到达的 tcp 连接请求。

### 原型

```
SOCKET accept(SOCKET socket, void *pvClntIP, USHORT *pusClntPort, INT nWaitSecs, EN_ONPSERR *penErr);
```

## 入口参数

- socket: socket 句柄
- pvClntIP: 指向 tcp 客户端地址 (ipv4/ipv6) 的指针
- pusClntPort: 指向 tcp 客户端端口的指针
- nWaitSecs: 指定等待时长, 单位: 秒。0, 不等待, 立即返回; 大于 0, 等待指定时间直至收到一个客户端连接请求或超时; 小于 0, 一直等待, 直至收到一个客户端连接请求或协议栈报错



➤ penErr: 指向错误编码的指针, 函数执行失败时该参数用于接收实际的错误码

## 返回值

返回请求连接的 tcp 客户端的 socket 句柄; 当没有新的客户端连接请求到达或协议栈报错时返回 INVALID\_SOCKET, 具体的错误码通过 penErr 参数获得。

## 示例

```
EN_ONPSERR enErr;

.....

SOCKET hSockSrv = socket(AF_INET, SOCK_STREAM, 0, &enErr);
if(INVALID_SOCKET != hSockSrv)
{
    if(!bind(hSockSrv, NULL, 6411)) /* 绑定成功
    {
        if(!listen(hSockSrv, usBacklog)) /* 进入监听状态
        {
            /* 循环等待并处理到达的 tcp 连接请求
            while(1)
            {
                in_addr_t unCltIP;
                USHORT usCltPort;
                SOCKET hSockClnt = accept(hSockSrv, &unCltIP, &usCltPort, 1, &enErr);
                if(INVALID_SOCKET != hSockClnt) /* 返回了一个有效的客户端 socket 句柄
                {
                    /* 新的客户端到达, 在这里增加你的自定义代码
                    .....
                }
            }
            else
            {
                /* 错误码为 ERRNO 代表无错误发生, 意味着没有新的客户端连接请求到达, 回到循环开始处继续等待即可
                if(ERRNO == enErr)
                {
                    continue;
                }
                else /* 不等于 ERRNO 意味着协议栈报错, 需要处理
                {
                    .....
                    printf("%s\r\n", onps_error(enErr)); /* 打印错误信息
                }
            }
        }
    }
}

else /* 失败
{
    printf("%s\r\n", onps_get_last_error(hSocket, NULL)); /* 打印错误信息
}

else /* 绑定失败
{
    printf("%s\r\n", onps_get_last_error(hSocket, NULL)); /* 打印错误信息
}
}
```

```
else
    printf("%s\r\n", onps\_error(enErr)); /*打印错误信息
.....
```

## tcpsrv\_recv\_poll

### 功能

阻塞/非阻塞型，tcp 服务器数据接收专用函数，等待任意一个或多个 tcp 客户端数据到达信号。

### 原型

```
SOCKET tcpsrv_recv_poll(SOCKET hSocketSrv, INT nWaitSecs, EN_ONPSERR *penErr);
```

### 入口参数

- hSocketSrv: tcp 服务器的 socket 句柄
- nWaitSecs: 等待时长，单位：秒。0，不等待，立即返回；大于 0，等待指定时间直至收到一个/多个客户端数据到达信号或超时；小于 0，一直等待，直至收到一个/多个客户端数据到达信号或协议栈报错
- penErr: 指向错误编码的指针，函数执行失败时该参数用于接收实际的错误码

### 返回值

返回已经收到数据的 tcp 客户端的 socket 句柄；当没有任何 tcp 客户端收到数据或协议栈报错时返回 INVALID\_SOCKET，具体的错误码通过 penErr 参数获得。

### 示例

参见 3.1.11 节 [THTcpSrv 线程（任务）](#) 的实现代码。

## tcpsrv\_set\_recv\_mode

### 功能

设置服务器的接收模式。有两种模式可以选择：

- TCPSRVRCVMODE\_ACTIVE 主动模式，通过 recv() 函数遍历读取每个已连接的客户端到达的数据
- TCPSRVRCVMODE\_POLL poll 模式，通过 tcpsrv\_recv\_poll() 函数等待用户数据到达

### 原型

```
BOOL tcpsrv_set_recv_mode(SOCKET hSocketSrv, CHAR bRcvMode, EN_ONPSERR *penErr);
```

### 入口参数

- hSocketSrv: tcp 服务器的 socket 句柄
- bRcvMode: 指定数据接收模式
- penErr: 指向错误编码的指针，该参数用于接收实际的错误码

### 返回值

设置成功返回 TRUE；否则返回 FALSE。

## 示例

略

# tcpsrv\_start

## 功能

这个函数的存在纯粹是为了简化用户编程。调用该函数就不需要再依次调用 `socket()`、`bind()`、`listen()` 等函数来启动 tcp 服务了，其自动完成 tcp 服务的创建工作，绑定任意 ip 地址。

## 原型

```
SOCKET tcpsrv_start(INT family, USHORT usSrvPort, USHORT usBacklog, CHAR bRcvMode, EN_ONPSERR *penErr);
```

## 入口参数

- family: 地址族类型，取值为 AF\_INET 或 AF\_INET6，前者为 IPv4 后者为 IPv6
- usSrvPort: 指定服务端口
- usBacklog: 等待用户层接受 (accept) 连接请求的 tcp 客户端数量
- bRcvMode: 指定数据接收模式
- penErr: 指向错误编码的指针，该参数用于接收实际的错误码

## 返回值

创建成功返回 socket；否则返回 INVALID\_SOCKET。

## 示例

略

# tcp\_send

## 功能

同样也是为了简化用户编程同时增加了容错处理逻辑：

- 1) 如果开启 tcp sack 支持，函数会确保所有数据送达底层链路的发送缓存后再返回，如果底层链路报错则返回 FALSE；
- 2) 如果未开启 tcp sack 支持，函数会在单次发送失败后重试，直至重试三次依然失败再返回 FALSE；

## 原型

```
BOOL tcp_send(SOCKET hSocket, UCHAR *pubData, INT nDataLen);
```

## 入口参数

- hSocket: socket 句柄
- pubData: 指向用户数据的指针
- nDataLen: 用户数据长度

## 返回值

发送成功返回 TRUE；否则返回 FALSE。

## 示例

略

## socket\_get\_last\_error/onps\_get\_last\_error

### 功能

获取 socket 最近一次发生的错误，包括描述信息及错误编码。该函数其实是前面示例代码中出现的 onps\_get\_last\_error() 函数的二次封装，功能及使用方式与之完全相同。

### 原型

```
const CHAR *socket_get_last_error(SOCKET socket, EN_ONPSERR *penErr);
```

### 入口参数

- socket: socket 句柄
- penErr: 指向错误编码的指针，该参数用于接收实际的错误码

### 返回值

返回值为字符串指针，指向 socket 最近一次发生的错误描述字符串。

### 示例

参见示例代码 onps\_get\_last\_error() 函数的使用。

## socket\_get\_last\_error\_code

### 功能

获取 socket 最近一次发生的错误编码。

### 原型

```
EN_ONPSERR socket_get_last_error_code(SOCKET socket);
```

### 入口参数

- socket: socket 句柄

### 返回值

返回值为 socket 最近一次发生的错误编码。

### 示例

略

## 5 常用工具函数

协议栈还提供了一组网络编程常见的工具函数以供用户使用，同时还提供了一些常用的比如字符串操作、16 进制格式化转换输出等函数：

- htonXX 系列：网络字节序转换函数

- inet\_XX 系列：网络地址转换函数
- inet6\_XX 系列：ipv6 地址转换函数
- ip\_addressing：检查 ip 地址是否在同一网段
- strtok\_safe：线程安全的 strtok 函数
- snprintf\_hex：将 16 进制数据格式化转换成字符串
- printf\_hex：将 16 进制数据格式化转换成字符串后输出到控制台
- onps\_error：将协议栈返回的错误码转换成具体的描述字符串

## htonll

### 功能

实现 64 位长整型数的网络字节序转换。

### 原型

```
LONGLONG htonll(LONGLONG llVal);
```

### 入口参数

➤ llVal：64 位长整型数

### 返回值

返回值为字节序转换后的 64 位长整型数。

### 示例

略

## htonl

### 功能

实现 32 位整型数的网络字节序转换。

### 原型

```
LONG htonl(LONG lVal);
```

### 入口参数

➤ lVal：32 整型数

### 返回值

返回值为字节序转换后的 32 位整型数。

### 示例

略

## htons

### 功能

实现 16 位整型数的网络字节序转换。

### 原型

```
SHORT htonl(SHORT sVal);
```

### 入口参数

➤ sVal: 16 位整型数

### 返回值

返回值为字节序转换后的 16 位整型数。

### 示例

略

## inet\_addr

### 功能

实现点分十进制 IPv4 地址到 4 字节无符号整型地址的转换，即 10.0.1.2 转换为 0x0A000102。

### 原型

```
in_addr_t inet_addr(const char *pszIP);
```

### 入口参数

➤ pszIP: 指向点分十进制 IPv4 地址字符串的指针

### 返回值

返回值为无符号 32 位整型地址。

### 示例

略

## inet\_addr\_small

### 功能

实现点分十进制 IPv4 地址到 4 字节无符号整型地址的转换，即 10.0.1.2 转换为 0x0201000A。

### 原型

```
in_addr_t inet_addr_small(const char *pszIP);
```

### 入口参数



➤ pszIP: 指向点分十进制 IPv4 地址字符串的指针

## 返回值

返回值为无符号 32 位整型地址。

## 示例

略

# inet\_ntoa

## 功能

注意，这是一个线程不安全的函数，实现 in\_addr 类型的地址到点分十进制 IPv4 地址的转换。

## 原型

```
char *inet_ntoa(struct in_addr stInAddr);
```

## 入口参数

➤ stInAddr: 指向 in\_addr 类型的 IPv4 地址的指针

## 返回值

返回字符串指针，指向转换后的点分十进制格式的 IPv4 地址字符串。

## 示例

```
struct in_addr stAddr;  
  
stSrcAddr.s_addr = inet_addr_small("192.168.0.9");  
  
printf("%s\r\n", inet_ntoa(stAddr));
```

# inet\_ntoa\_ext

## 功能

注意，这是一个线程不安全的函数，实现 4 字节无符号整型地址到点分十进制 IPv4 地址的转换。

## 原型

```
char *inet_ntoa_ext(in_addr_t unAddr);
```

## 入口参数

➤ unAddr: 要转换的 IPv4 地址，4 字节无符号整型格式

## 返回值

返回字符串指针，指向转换后的点分十进制格式的 IPv4 地址字符串。

## 示例

```
in_addr_t unAddr = inet_addr_small("192.168.0.9");
```

```
printf("%s\r\n", inet_ntoa_ext(unAddr));
```

## inet\_ntoa\_safe

### 功能

注意，这是一个线程安全的函数，实现 in\_addr 类型的地址到点分十进制 IPv4 地址的转换。

### 原型

```
char *inet_ntoa_safe(struct in_addr stInAddr, char *pszAddr);
```

### 入口参数

- stInAddr: 指向 in\_addr 类型的 IPv4 地址的指针
- pszAddr: 指向转换后的点分十进制 IPv4 地址字符串的指针

### 返回值

返回字符串指针，指向转换后的点分十进制格式的 IPv4 地址字符串，其地址其实就是参数 pszAddr 指向的地址。

### 示例

```
CHAR szAddr[20];

struct in_addr stAddr;

stSrcAddr.s_addr = inet_addr_small("192.168.0.9");

printf("%s\r\n", inet_ntoa_safe(stAddr, szAddr));
```

## inet\_ntoa\_safe\_ext

### 功能

注意，这是一个线程安全的函数，实现 4 字节无符号整型地址到点分十进制 IPv4 地址的转换。

### 原型

```
char *inet_ntoa_safe_ext(in_addr_t unAddr, char *pszAddr);
```

### 入口参数

- unAddr: 要转换的 IPv4 地址，4 字节无符号整型格式
- pszAddr: 指向转换后的点分十进制 IPv4 地址字符串的指针

### 返回值

返回值为字符串指针，指向转换后的点分十进制格式的 IPv4 地址字符串，其地址其实就是参数 pszAddr 指向的地址。

### 示例

```
CHAR szAddr[20];

in_addr_t unAddr = inet_addr_small("192.168.0.9");
```

```
printf("%s\r\n", inet_ntoa_safe_ext(unAddr, szAddr));
```

## inet6\_ntoa

### 功能

实现 16 进制 ipv6 地址到冒号分割地址串的转换。

### 原型

```
const CHAR *inet6_ntoa(const UCHAR ubaIpv6[16], CHAR szIpv6[40]);
```

### 入口参数

- ubaIpv6: 要转换的 IPv6 地址，16 进制
- szIpv6: 指向转换后的冒号分割地址串的指针

### 返回值

返回值为字符串指针，指向转换后的冒号分割地址串，其地址其实就是参数 szIpv6 指向的地址。

### 示例

略

## inet6\_aton

### 功能

实现冒号分割地址串到 16 进制 ipv6 地址的转换。

### 原型

```
const UCHAR *inet6_aton(const CHAR *pszIpv6, UCHAR ubaIpv6[16]);
```

### 入口参数

- pszIpv6: 指向要进行转换的冒号分割地址串的指针
- ubaIpv6: 指向转换后的 16 进制 ipv6 地址的指针

### 返回值

返回一个指向转换后的 16 进制 ipv6 地址的指针，其指向的地址其实就是参数 ubaIpv6 指向的地址。

### 示例

略

## ip\_addressing

### 功能

比较两个 IPv4 地址是否属于同一网段。

## 原型

```
BOOL ip_addressing(in_addr_t un1stIp, in_addr_t un2ndIp, in_addr_t unGenmask);
```

## 入口参数

- un1stIp: 第一个被比较的 IPv4 地址
- un2ndIp: 第二个被比较的 IPv4 地址
- unGenmask: 子网掩码

## 返回值

返回 TRUE 表示在同一网段, FALSE 则不属于同一网段。

## 示例

略

# strtok\_safe

## 功能

线程安全的 strtok 函数。

## 原型

```
CHAR *strtok_safe(CHAR **ppszStart, const CHAR *pszSplitStr);
```

## 入口参数

- ppszStart: 指向下一个要被截取的字符串片段的指针的指针
- pszSplitStr: 指向分隔符的指针

## 返回值

返回字符串指针, 指向下一个分隔符之前的字符串; 返回值为 NULL 则截取完毕。

## 示例

```
CHAR szTestStr[64];

sprintf(szTestStr, "123;456;789,ABC;,EFG");

CHAR *pszStart = szTestStr;

CHAR *pszItem = strtok_safe(&pszStart, ";");

while(NULL != pszItem)
{
    printf("%s\r\n", pszItem);
    pszItem = strtok_safe(&pszStart, ";");
}
```

# snprintf\_hex

## 功能

将 16 进制数据格式化为字符串。

## 原型

```
void snprintf_hex(const UCHAR *pubHexData, USHORT usHexDataLen, CHAR *pszDstBuf, UINT unDstBufSize, BOOL blIsSeparate);
```

## 入口参数

- pubHexData: 指向 16 进制数据的指针
- usHexDataLen: 16 进制数据的长度
- pszDstBuf: 指向接收缓冲区的指针，其用于接收格式化后的字符串
- unDstBufSize: 接收缓冲区的长度，单位：字节
- blIsSeparate: 格式化后的字符串在两个 16 进制数据之间是否增加空格，即 2A 1F 还是 2A1F

## 返回值

无

## 示例

```
UCHAR ubHexData[16] = "\xAB\xCD\x2A\x1F\x3C\x4D";  
CHAR szHexDataStr[64];  
snprintf_hex(ubHexData, 6, szHexDataStr, 64, TRUE);  
printf("%s\r\n", szHexDataStr);
```

# printf\_hex

## 功能

将 16 进制数据格式化为字符串输出到控制台。

## 原型

```
void printf_hex(const UCHAR *pubHexData, USHORT usHexDataLen, UCHAR ubBytesPerLine);
```

## 入口参数

- pubHexData: 指向 16 进制数据的指针
- usHexDataLen: 16 进制数据的长度
- ubBytesPerLine: 每行固定输出多少 字节的数据，比如 16 字节一行

## 返回值

无

## 示例

```
UCHAR ubHexData[16] = "\xAB\xCD\x2A\x1F\x3C\x4D\xAA\x4E\xFE\x45\x6B\x9A\x05\x71\x8E\x1B\x52\x78";  
printf_hex(ubHexData, 18, 16);
```

# onps\_error

## 功能

将协议栈返回的错误码转换成具体的描述字符串。

## 原型

```
const CHAR *onps_error(EN_ONPSERR enErr);
```

## 入口参数

- enErr: 错误编码

## 返回值

返回值为字符串指针，指向具体的错误描述字符串的指针

## 示例

略

# 6 获取源码

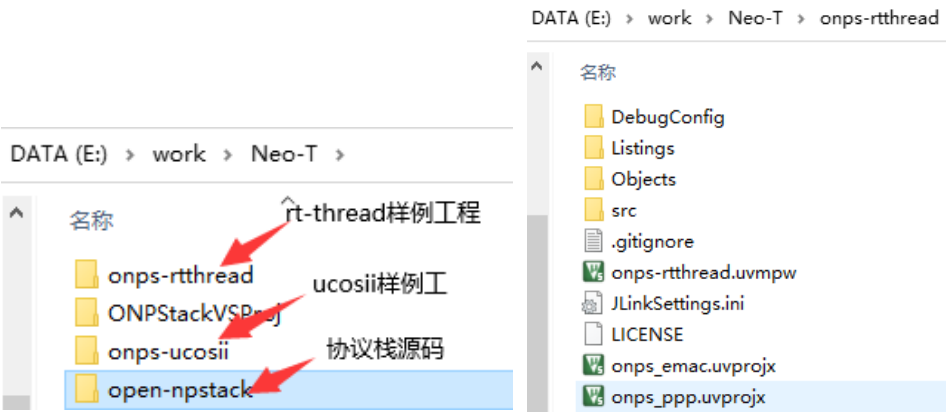
协议栈源码从码云或 github 上获取。

- https://gitee.com/Neo-T/open-npstack/releases/tag/v1.1.0.230726
- https://github.com/Neo-T/OpenNPStack/releases/tag/v1.1.0.230726

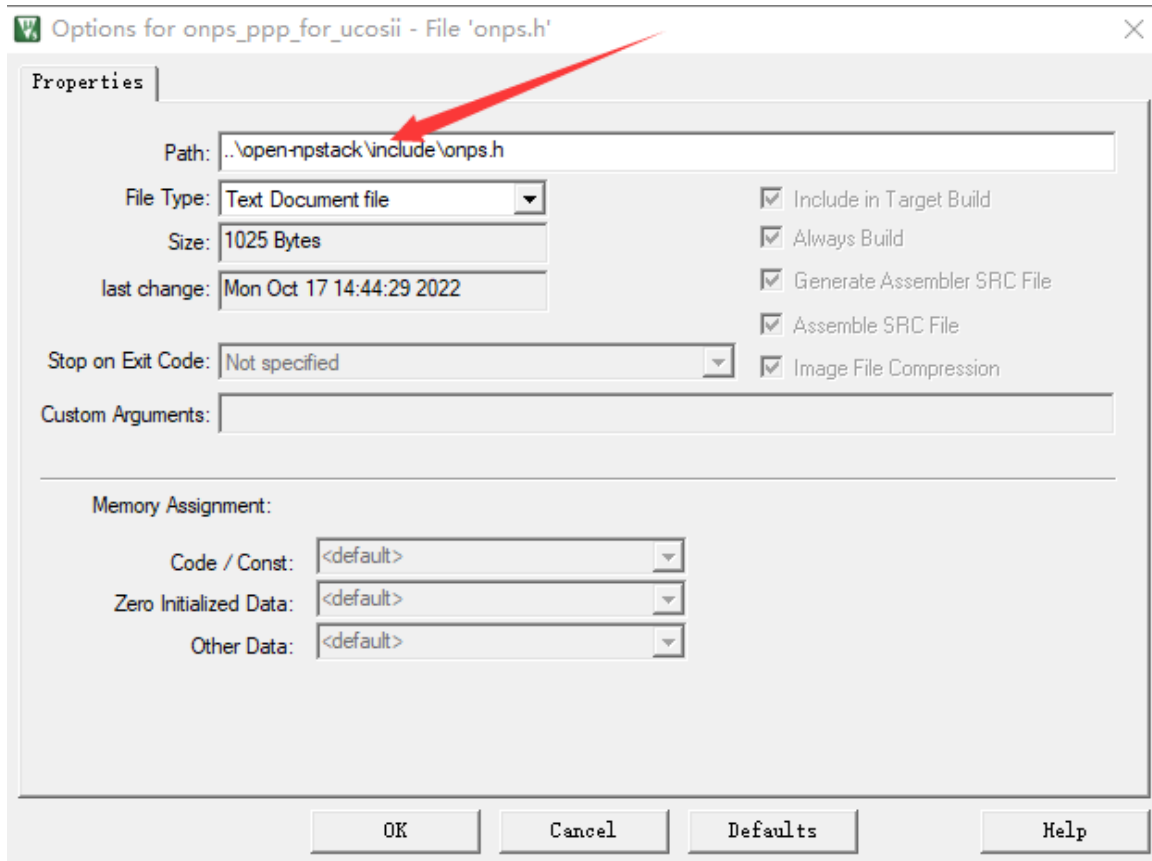
移植样例工程的源码从码云上获取。

- https://gitee.com/Neo-T/onps-rtthread/releases/tag/v1.1.0
- https://gitee.com/Neo-T/onps-ucosii/releases/tag/v1.1.0

需要注意的是，移植样例工程一定要与协议栈源码放到到同一个文件夹下，否则用 keil 打开时会找不到除移植文件外的其它协议栈源码文件。文件夹上层组织如下：



样例工程中协议栈源码在 IDE 下的文件位置定义如下：



## 7 后续工作计划

- ✧ 更多目标平台的适配工作，提供相应移植样例；
- ✧ 重构部分代码，进一步降低代码尺寸及内存消耗；
- ✧ 支持 ftp 客户端/服务器；
- ✧ 支持 http 客户端/服务器；