修改历史		
版本	日期	说明
1.0.0	2022-10-17	针对 onps 栈初始版本的移植说明
1.1.0.230726	2023-07-26	1. os 适配层新增接口函数 os_get_system_msecs()的移植说明； 2. 新增模块 ipv6、tcp sack、telnet 等配置说明； 3. 其它一些因新增模块产生的细节修改说明； 4. 重新划分了章节；

Onps 栈移植手册

onps 栈的移植涉及几个部分：**1）系统配置及裁剪；2）基础数据类型定义；3）RTOS 适配层实现；4）编写网卡驱动并注册网卡**。本文作为 onps 栈移植的指导性文件将给出一般性的移植说明及建议，可参考的移植样例工程请移步码云下载：

- 目标 os 为 rt-thread 的移植样例工程（STM32F407VET6 及 STM32F103RCT6 两个平台）
- 目标 os 为 ucos-ii 的移植样例工程（硬件平台同上）

1. onps 栈的配置及裁剪

协议栈源码（码云/github）port/include/port/sys_config.h 文件是协议栈的配置文件。它提供了一系列的配置宏用于裁剪、配置协议栈。我们可以根据目标系统的具体情况对协议栈进行裁剪，调整配置，以减少或增加对系统资源的占用率。配置文件主要涉及几方面的内容：
1）打开或关闭某个功能模块；
2）指定 mmu（内存管理单元）管理的内存大小；
3）协议层相关配置项，如缺省 ttl 值、路由表大小、arp 缓存表大小等；

1.1 网络接口

onps 栈在数据链路层支持两种类型的网络接口：ethernet 有线以太网络接口；ppp 点对点拨号网络接口。用户必须选择其中至少一个接口：

```
#define SUPPORT_PPP      1 //* 是否支持 ppp：1，支持；0，不支持
#define SUPPORT_ETHERNET 1 //* 是否支持 ethernet：1，支持；0，不支持
```

注意，你的目标系统要么支持 ppp，要么支持 ethernet，要么二者都支持，不能两个都选择不支持，否则协议栈将无法工作。

1.2 网络工具

协议栈还提供了几个常用的网络工具供用户选择使用，用户可以根据具体应用情形选择打开或关闭相关工具：

```
//* 网络工具配置项，0：不支持；1：支持，协议栈将编译连接工具代码到目标系统
```

```
//* ============================================================================================
#define NETTOOLS_PING       1 //* ping 工具，确定目标网络地址是否能到达
#define NETTOOLS_DNS_CLIENT 1 //* dns 查询客户端，通过指定的 dns 服务器查询请求域名对应的 ip 地址
#define NETTOOLS_SNTP       1 //* sntp 客户端，通过指定的 ntp 服务器进行网络校时
#define NETTOOLS_TELNETSRV  1 //* 使能或禁止 telnet 服务端，其值必须为 0 或 1（禁止/使能），因为其还被用于 tcp 服务器资源
                              //* 分配统计（TCPSRV_NUM_MAX + NETTOOLS_TELNETSRV）
//* ============================================================================================
```

协议栈提供一个标准的 telnet 服务器，支持 ANSI、VT100、XTERM 等终端，提供基本的网络操作指令如查看网络接口、增加、删除、修改 ip 地址、路由条目等。支持 ping、dns 查询、网络校时等工具。用户可以根据自己的实际需求使能或禁止这些指令：

```
//* 根据具体需求对 telnet 服务器携带的指令进行裁剪，前提是 NETTOOLS_TELNETSRV 宏必须置位
//* ============================================================================================
#if NETTOOLS_TELNETSRV
  #define NVTCMD_MEMUSAGE_EN 1 //* 使能或禁止 nvt 命令，下同：memusage，查看协议栈内存使用情况
  #define NVTCMD_NETIF_EN    1 //* nvt 命令：netif，查看当前系统已成功注册并启动的网络接口，相当于 ipconfig、ifconfig
  #define NVTCMD_IFIP_EN     1 //* nvt 命令：ifip，增加、删除、修改指定网络接口的 ip、mac、dns 地址等
  #define NVTCMD_ROUTE_EN    1 //* nvt 命令：route，打印路由表，增加、删除路由条目
  #define NVTCMD_TELNET_EN   1 //* nvt 命令：telnet，telnet 客户端,用于登录其它标准 telnet 服务器（如 linux、windows 等）

  #if NETTOOLS_SNTP
    #define NVTCMD_NTP_EN    1 //* nvt 命令：ntp，网络对时，其必须先使能 NETTOOLS_SNTP
  #endif //* #if NETTOOLS_SNTP

  #if NETTOOLS_DNS_CLIENT
    #define NVTCMD_NSLOOKUP_EN 1 //* nvt 命令：nslookup，域名到 ip 地址的解析（ipv4），其必须先使能 NETTOOLS_DNS_CLIENT
  #endif //* #if NETTOOLS_DNS_CLIENT

  #if NETTOOLS_PING
    #define NVTCMD_PING_EN     1 //* nvt 命令：ping，轻型的 ping 探测命令，其必须先使能 NETTOOLS_PING
  #endif //* #if NETTOOLS_PING

  #define NVTCMD_RESET_EN 1 //* nvt 命令：reset，复位目标系统

  #if NVTCMD_TELNET_EN
    #define TELNETCLT_RCVBUF_SIZE 1024 //* telnet 客户端接收缓冲区大小，注意关闭 TCP SACK 选项时，设置的接收缓冲区大小一
                                       //* 旦超过 tcp mtu（一般为 1460 字节），就必须是在用户层限定单个发包的大小不能超过
                                       //* tcp mtu，否则会丢失数据
  #endif //* #if NVTCMD_TELNET_EN

  #define NVTNUM_MAX       2   //* 指定 nvt 并发工作的数量，其实就是指定 telnet 服务器在同一时刻并发连接的数量，超过这个
                               //* 数值服务器拒绝连接
  #define NVTCMDCACHE_EN   1   //* 是否支持命令缓存，也就是通过“↑↓”切换曾经输入的指令
  #define NVTCMDCACHE_SIZE 256 //* 指定指令缓存区的大小

  #if SUPPORT_IPV6
    #define TELNETSRV_SUPPORT_IPv6 1 //* telnet 服务器使能或禁止 ipv6 支持，与 NETTOOLS_TELNETSRV 同，其值必须为 0 或 1
  #endif //* #if SUPPORT_IPV6
```

```
#endif //* #if NETTOOLS_TELNETSRV
```

上面**红色加粗**部分与网络虚拟终端 **NVT**（Network Virtual Terminal）配置相关。NVT 类似传统意义上的控制台终端，如 windows 下的 cmd，linux 下的 console。当用户登录 telnet 服务器后，协议栈会随之启动一个独立的 NVT 为之提供终端操作服务，用户可以通过输入“help”获得详细的 NVT 命令列表，出现在列表中的命令用户可以随时调用，操作方式与传统控制台相同。在上面给出的配置项中，以“NVTCMD_”作为前缀，“_EN”为尾部标识的预定义宏被用于单独**使能**或**禁止**某个 NVT 命令。被禁止的 NVT 命令将不会出现在“help”列表中且不能在 NVT 中调用。考虑到协议栈运行的目标系统资源有限，所以我们在这里单独提供一个宏 **NVTNUM_MAX** 用于设置 telnet 服务器允许的并发登录数量，确保资源不被耗尽。以“NVTCMDCACHE_”作为前缀的两个预定义宏被用于配置 NVT 命令缓存。使能命令缓存需要消耗一定的内存资源用于保存 telnet 用户登录服务器后输入的命令记录。用户可以像操作传统终端一样，使用上下箭头“↑↓”调取、查阅命令记录，简化用户输入操作。**用户设定的缓存大小决定了协议栈能够记录的命令条数。**

NVT 还提供了一个 telnet 命令用于登录其它标准 telnet 服务器，如登录 linux 或 windows 提供的 telnet 服务器。它其实就是一个运行在 NVT 下的轻型 telnet 客户端。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。**TELNETCLT_RCVBUF_SIZE** 宏用于设置这个 telnet 客户端的接收缓冲区大小。当客户端收到 telnet 服务器下发的数据后，这个大小决定了我们进行转发的数据大小。因为作为中继代理，协议栈收到的数据最终是要转发给远程 telnet 客户端的（也就是登录到协议栈提供的 telnet 服务器的远程终端）。最简单的设计就是我们收到多少，就立即转发多少，而不是分多次。但这种设计只在两种情形下可行：

1、 协议栈 SUPPORT_SACK 选项置位；

2、 协议栈 SUPPORT_SACK 选项未置位，但缓冲区大小不超过 tcp 链路协商确定的 MSS 值；

SUPPORT_SACK 选项置位，协议栈底层负责 tcp 报文的发送，其会根据 MSS 值动态调整单个 tcp 报文的大小，用户层只管将数据交给 tcp 链路即可，数据发送的可靠性由协议栈负责。当协议栈 SUPPORT_SACK 选项未置位，意味着我们采用了问答式 tcp 通讯模式。tcp 报文送达对端且收到 tcp ack 后才能发送下一个报文。这种模式下，单个数据报文大小不能超过 tcp 链路协商确定的 MSS 值（一般为 1460 字节），否则 tcp 链路会悄悄丢弃超过的部分，这将导致数据缺失，telnet 通讯不可靠。解决这个问题的唯一方法就是手动调整协议栈 net_tools/telnet_client.c 文件中 telnet_srv_data_handler()函数数据转发部分的代码，控制单次转发的大小不超过 MSS 值。

telnet 服务器能够提供 ipv4 与 ipv6 并发访问支持，前提是 SUPPORT_IPV6 宏、TELNETSRV_SUPPORT_IPv6 宏同时置位。换句话说，我们既可以通过 ipv4 地址也可以通过 ipv6 地址随时登录到服务器进行相关操作。

除协议栈已经提供的这些 NVT 命令外，我们还可以根据自身需求增加更多的命令。有关 telnet 服务器移植及扩展 NVT 命令相关的内容参见“移植 telnet 服务器”一节。

考虑协议栈的目标系统可能无法提供 pc 下常见的文件存储系统，所以协议栈的调试日志等信息是通过标准输出提供的：

```
#define SUPPORT_PRINTF 1 //* 是否支持调用 printf()输出相关调试或系统信息
#if SUPPORT_PRINTF
  #define PRINTF_THREAD_MUTEX 1 //* 是否支持使用 printf 线程互斥锁，确保不同线程的调试输出信息不被互相干扰
  #define DEBUG_LEVEL         1 //* 共 5 个调试级别：
                                //* 0 输出协议栈底层严重错误
                                //* 1 输出所有系统错误（包括 0 级错误）
                                //* 2 输出协议栈重要的配置、运行信息，同时包括 0、1 级信息
                                //* 3 输出网卡的原始通讯通讯报文（ppp 为收发，ethnernet 为发送），以及 0、1、2 级信息
                                //* 4 输出 ethernet 网卡接收的原始通讯报文，被协议栈丢弃的非法（校验和错误、通讯链路
                                //* 不存在等原因）通讯报文，以及 0、1、2、3 级信息（除 ethernet 发送的原始报文）
```

```
#endif
```

基本上所有单片机系统均会提供几个串行口,我们只需选择其中一个将其作为 printf 函数的标准输出口，我们就可以使能协议栈支持日志输出功能，通过 printf()函数输出的日志信息对目标系统进行调试。如果你的目标系统支持某个串口作为 printf()函数的标准输出口，建议将 SUPPORT_PRINTF 宏置 1，打开协议栈的日志输出功能。PRINTF_THREAD_MUTEX 宏用于解决多线程环境下日志输出的冲突问题。如果你的目标系统互斥资源够用，建议打开该功能，否则你在标准输出口看到的日志会出现乱序问题。

1.3 内存管理单元（MMU）

协议栈在很多情形下需要动态申请不同大小的内存以供接下来的逻辑处理过程使用。所以，为了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，我们还单独设计了一个独立的内存管理单元（mmu)。考虑协议栈的目标系统为资源受限的单片机系统，这种系统的内存资源往往都是极度紧张的，因此我们提供了配置宏让用户决定分配多少字节的内存空间给协议栈的 mmu：

```
//* 内存管理单元(mmu)相关配置项，其直接影响协议栈能分配多少个 socket 给用户使用
//* ============================================================================================
#define BUDDY_PAGE_SIZE  32   //* 系统能够分配的最小页面大小，其值必须是 2 的整数次幂
#define BUDDY_ARER_COUNT 9    //* 指定 buddy 算法管理的内存块数组单元数量
#define BUDDY_MEM_SIZE   8192 //* buddy 算法管理的内存总大小，其值由 BUDDY_PAGE_SIZE、BUDDY_ARER_COUNT 两个宏计算得到：
                              //* 32 * (2 ^ (9 - 1))，即 BUDDY_MEM_SIZE = BUDDY_PAGE_SIZE * (2 ^ (BUDDY_ARER_COUNT - 1))
                              //* 之所以在此定义好要管理的内存大小，原因是 buddy 管理的内存其实就是一块提前分配好的静
                              //* 态存储时期的字节型一维数组，以此来确保协议栈不占用宝贵的堆空间
//* ============================================================================================
```

协议栈的内存管理单元采用了 buddy 伙伴算法。以“BUDDY_”作为前缀的三个宏的关系参见 BUDDY_MEM_SIZE 宏的注释。前面说过，mmu 管理的内存用于协议栈的不同业务情形，其中最核心的一种业务情形就是 socket，用户分配的内存大小直接决定了用户编写网络应用时能够申请的 socket 数量。如果你在申请分配一个新的 socket 时报 ERRREQMEMTOOLARGE（The requested memory is too large, please refer to the macro definition BUDDY_MEM_SIZE）或 ERRNOFREEMEM（The mmu has no memory available）错误，则意味着内存已经不够用了,需要你增加内存或者检视你的代码看是否存在未及时释放的 socket 句柄。另外，决定内存利用效率的关键配置项是 BUDDY_PAGE_SIZE 宏，因为 mmu 分配内存的最小单位就是“页”。这个宏设置单个内存页的大小，单位为字节，其值必须是 2 的整数次幂。如果你的通讯报文不大，建议把页面大小调整的小一些，比如 16 字节、32 字节等，以尽量减少单个页面的空余字节数。

1.4 缓存链表（Buffer List）

为了最大限度地减少内存复制带来的性能及资源开销，协议栈采用缓存链表（buffer list）机制来实现写时零复制（zero copy）技术。用户数据在向下层协议传递直至到达数据链路层被发送出去，其间不进行任何复制。每一层封装的报文数据均以缓存（buffer）节点的形式顺序挂接到链表（list）上。数据发送完毕，各节点占用的缓存(buffer)才会被释放。缓存链表机制应用于所有网络通讯，包括 arp、tcp、udp、icmp 等。缓存链表需要事先分配一定的内存资源用于链表操作，我们可以依据应用场景的实际需求及内存使用情况酌情调整缓存链表可供使用的节点总数：

```
#define BUF_LIST_NUM 64   //* 缓存链表的节点数，最大不能超过 2 的 15 次方（32768）
```

调整策略可以采用等幅逐步减少或增加的方式，先找到临界值，然后按照一定比例增加裕量即可确定一

个合适的节点数量。

1.5 协议层

sys_config.h 文件的其余宏均为协议层相关的配置项。这里面最重要的一项就是使能或禁止 ipv6 支持：

```
#define SUPPORT_IPV6 1 //* 使能或禁止 Ipv6：置位使能，复位禁止
```

使能 Ipv6，armcc 编译器下代码尺寸会增加约 26K 字节（优化级别 0），内存消耗增加 272 个字节。除 Ipv6 外，还有几个与底层网络接口相关的配置项需要特别关注：

```
#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 //* 协议栈加载几路 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

#if SUPPORT_ETHERNET
  #define ETHERNET_NUM     1  //* 要添加几个 ethernet 网卡（实际存在几个就添加几个）
  #define ARPENTRY_NUM     32 //* arp 条目缓存表的大小，只要不小于局域网内目标通讯节点的个数即可确保 arp 寻址次数为 1，否则
                              //* 就会出现频繁寻址的可能，当然这也不会妨碍正常通讯逻辑，只不过这会降低通讯效率
  #define SUPPORT_IPV6
    #define IPV6TOMAC_ENTRY_NUM 8 //* Ipv6 到以太网 mac 地址映射缓存表的大小（不要超过 127），同 ARPENTRY_NUM 宏
    #define IPV6_CFG_ADDR_NUM   8 //* 指定所有以太网卡能够自动/手动配置的最大地址数量（不要超过 128）
    #define IPV6_ROUTER_NUM     4 //* 指定所有以太网卡能够添加的路由器最大数量（最多 8 个）
  #endif

  #define ETH_EXTRA_IP_EN 1 //* 是否允许添加多个 ip 地址
  #if ETH_EXTRA_IP_EN
    #define ETH_EXTRA_IP_NUM 2 //* 允许添加的 ip 地址数量
  #endif
#else
  #define ETHERNET_NUM 0
#endif
```

如果目标系统需要用到 ppp 拨号，我们在打开协议栈对 ppp 模块的支持后还需要设置缺省的拨号参数值，比如 apn、拨号账号及密码等。当然你也可以不用设置，后面我们在编写 os 适配层接口的时候也会设置这几项。系统会使用 os 适配层的设置值代替缺省值。另外协议栈在设计之初即考虑支持多路 ppp 同时拨号的情形，目标系统支持几路 ppp，宏 PPP_NETLINK_NUM 值置几即可。SUPPORT_ECHO 宏指定 ppp 链路是否启用 echo 回显探测功能。某些 ppp 接入服务商可能会关闭此项功能，建议缺省情况下关闭此功能。因

为 echo 链路探测功能一旦被启用，协议栈会每隔一小段时间发送探测报文到对端。对端如果不支持此功能会丢弃该探测报文不做任何响应。这将导致协议栈判定 ppp 链路故障，从而主动结束链路、重新拨号。

协议栈同样支持多路 ethernet 网卡，ETHERNET_NUM 宏用于指定目标系统存在几路 ethernet 网卡。这里需要特别注意的是 ARPENTRY_NUM 宏，这个宏用于指定 ethernet 网络环境下进行通讯时 mac 地址缓存表的大小。如果缓存表过小，进行通讯的目标地址并不在缓存表中时，协议栈会先发送 arp 查询报文，得到对端的 mac 地址后才会发送实际的通讯报文。虽然这一切都是协议栈自动进行的，但通讯效率会受到影响。如果目标系统的内存够用，建议放大缓存表的容量，最合理的大小是等于计划通讯的目标地址的数量。

当运行协议栈的终端设备存在多个网段通讯需求时，就需要为网络接口则加多个 ip 地址以对应不同的网段进行通讯。由于 ppp 链路的终端地址是由移动运营商分配的，所以 ppp 链路不支持增加多个 IP 地址。同样，当以太网卡采用 dhcp 动态地址分配策略时，亦不支持增加多个 ip 地址。只有以太网卡采用静态地址分配策略时 ETH_EXTRA_IP_EN 宏才有效。增加 ip 地址会增加更多的内存开销,ETH_EXTRA_IP_NUM 宏的取值以满足需求为准。

上面给出的涉及 ipv6 的配置项均出现在以太网接口下。这是因为 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。所有涉及 ipv6 的配置项均是在以太网络接口使能的情形下才被许可。这些配置项涉及几个方面：ipv6 地址数量、ipv6 路由器数量、ipv6 到 mac 地址缓存条目数量。这几个数量均会影响目标系统的内存开销。一般来说一个以太网接口一般会绑定两到三个地址：一个本地链路地址、一个无状态配置的动态地址，还有一个 DHCPv6 主机分配的动态地址（如果目标网络内存在 DHCPv6 主机的话）。所以 IPV6_CFG_ADDR_NUM 的取值应为：**ETHERNET_NUM * 3（或 2）**。路由器数量 IPV6_ROUTER_NUM 的取值则简单很多，目标网络存在几个支持 ipv6 的路由器就指定几个即可。IPV6TOMAC_ENTRY_NUM 宏的作用及取值策略与 ARPENTRY_NUM 宏完全相同，不再赘述。

其余协议层相关的配置项均属于 ip 及其支持的上层协议：

```
//* ip 支持的上层协议相关配置项
//* ================================================================================
#define SUPPORT_SACK 1 //* 系统是否支持 sack 项，置位：支持；复位：禁止

#define ICMPRCVBUF_SIZE_DEFAULT 128   //* icmp 发送 echo 请求报文时指定的接收缓冲区的缺省大小，注意，如果要发送
                                      //* 较大的 ping 包就必须指定较大的接收缓冲区

#define TCPRCVBUF_SIZE_DEFAULT  2048  //* tcp 层缺省的接收缓冲区大小，大小应是 2^n 次幂才能最大限度不浪费 budyy 模块
                                      //* 分配的内存
#if SUPPORT_SACK
#define TCPSNDBUF_SIZE 4096 //* tcp 层发送缓冲区大小，同接收缓冲区，大小应是 2^n 次幂才能最大限度不浪费 budyy 模块分配的内存
#endif

#define TCPUDP_PORT_START 20000 //* TCP/UDP 协议动态分配的起始端口号

#define TCP_WINDOW_SCALE    0   //* 窗口扩大因子缺省值
#define TCP_CONN_TIMEOUT    30  //* 缺省 TCP 连接超时时间
#define TCP_ACK_TIMEOUT     3   //* 缺省 TCP 应答超时时间

#define TCP_LINK_NUM_MAX    16  //* 系统支持最多建立多少路 TCP 链路（涵盖所有 TCP 客户端 + TCP 服务器的并发连接数），超过这
                                //* 个数量将无法建立新的 tcp 链路，另外这个值最大为 127，超过则系统无法正常运行

#define TCP_ACK_DELAY_MSECS 100 //* 延迟多少毫秒发送 ack 报文，这个值最小 40 毫秒，最大 200 毫秒
```

```
#if SUPPORT_ETHERNET
  #define TCPSRV_BACKLOG_NUM_MAX 10 //* tcp 服务器支持的最大请求队列数量，任意时刻所有已开启的 tcp 服务器的请求连
                                    //* 接队列数量之和应小于该值，否则将会出现拒绝连接的情况

  #define TCPSRV_NUM_MAX         0  //* 系统能够同时建立的 tcp 服务器数量
  #define TCPSRV_RECV_QUEUE_NUM  64 //* tcp 服务器接收队列大小，所有已开启的 tcp 服务器共享该队列资源，如果单位时
                                    //* 间内到达所有已开启 tcp 服务器的报文数量较大，应将该值调大
#endif

#define UDP_LINK_NUM_MAX 4   //* 调用 connect()函数连接对端 udp 服务器的最大数量（一旦调用 connect()函数，收到的非
                             //* 服务器报文将被直接丢弃）
#define SOCKET_NUM_MAX   16 //* 系统支持的最大 SOCKET 数量，如实际应用中超过这个数量则会导致用户层业务逻辑无法全
                             //* 部正常运行（icmp/tcp/udp 业务均受此影响），其值应大于等于 TCP_LINK_NUM_MAX 值
#define IP_TTL_DEFAULT   64 //* 缺省 TTL 值
#define ROUTE_ITEM_NUM   8  //* 系统路由表数量
//* ================================================================================================
```

关于 tcp sack，其为 tcp 选择性确认（Selective Acknowledgment）选项。该选项在 tcp 链路建立时的三次握手阶段协商确认。该选项一旦被链路支持，则允许 tcp 单独确认不连续的包，已到达的包不再被对端重传，这样可以大大提高传输效率。协议栈对 tcp sack 选项的支持是建立在报文缓存的基础上的。每创建一个 tcp 链路，协议栈都会为其建立一个发送缓存，以实现乱序、丢失包的重传。这意味着 tcp sack 选项虽然大大提升了传输效率，但也带来了较大的内存开销。如果我们需要较高的传输速度，就置位 SUPPORT_SACK 宏，同时增加 MMU 管理的内存大小。因为协议栈建立的 tcp 发送缓存需要的内存就是从 MMU 申请得到的。每一个链路建立的缓存大小也就是从 MMU 申请的内存大小由 TCPSNDBUF_SIZE 宏指定。

另外一个宏 TCP_ACK_DELAY_MSECS 被用于指定 tcp 回馈 ack 应答报文的延迟时间，单位：毫秒。当 tcp 收到报文后，并不会立即回馈 ack 给对端以确认收到，而是主动等待一小段时间，其目的是看看等待的这段时间内是否有用户数据要发送。如果有，则 ack 应答合并用户数据一起发送给对端，而不是分为两个包进行发送。这个选项存在的目的就是减少报文数量，提升通讯载荷效费比。TCP_ACK_DELAY_MSECS 宏的取值并没有一个严格的标准，windows、linux 系统一般为 200 毫秒。如果带宽足够，可以减少这个值，反之则增加这个值。

前缀为“TCPSRV_”的三个配置项只有在以太网接口被使能的情形下有效。它们被用于 tcp 服务器的资源配置。tcp 服务器需要固定 ip 地址才能访问，ppp 链路是不具备这个条件的。或者更严谨的说法是建立于 ppp 链路上的 tcp 服务器能否被远程访问是由移动运营商决定的，协议栈对此无能为力。开启 tcp 服务器并不会占用过多内存，每增加一路 tcp 服务器，仅增加二十多字节的内存开销。TCPSRV_NUM_MAX 宏的取值以实际需要为准。TCPSRV_BACKLOG_NUM_MAX 宏的取值应至少等于目标系统调用的所有 listen()函数指定的 backlog 参数值之和。TCPSRV_RECV_QUEUE_NUM 宏的取值没有严格的标准，其用于指定协议栈在同一时刻能够缓存的所有到达 tcp 服务器的数据包数量。如果 tcp 服务器在单位时间内通讯量很大，比如每秒最多能够到达 10 个数据包。则 TCPSRV_RECV_QUEUE_NUM 宏的取值为“**TCPSRV_NUM_MAX * 10 + 一定比例的裕量**”。

ICMPRCVBUF_SIZE_DEFAULT 宏与 ping 工具有关，如果你不想使用 ping 工具可以将这个值设小一些以节省内存。TCP_WINDOW_SCALE 宏建议不要做任何调整，对于内存空间有限的单片机系统 tcp 窗口直接使用指定值即可。TCP_ACK_TIMEOUT 宏用于指定 tcp 报文发送到对端后等待对端回馈 tcp ack 报文的超时时间，单位：秒。UDP_LINK_NUM_MAX 宏决定了目标系统在使用 udp 通讯时，能够建立的 udp 客户端的最大数量。比如目标系统需要建立 5 个 udp 客户端，由于 UDP_LINK_NUM_MAX 值为 4，那么只有 4 个客户端能正常调用 connect()函数，第 5 个客户端在调用 connect()函数时会报 ERRNOUDPLINKNODE(the udp link list is empty) 错误。ROUTE_ITEM_NUM 宏用于指定系统缓存的路由条目数量，你可以根据实际网络情形调整这个值，但不能低于目标系统注册的网卡数量。协议层相关的其它配置项请根据注释自行依据实际情况进行调整即可。

2. 字节对齐及基础数据类型定义

协议栈源码 port/include/port/datatype.h 中根据目标系统架构（16 位 or 32 位）及所使用的编译器定义了基础数据类型及字节对齐方法。这个文件中最重要的移植工作就是依据目标编译器手册定义字节对齐方法。因为网络协议栈最关键的地方就是底层通讯报文结构必须字节对齐，而不是通常情形下的缺省四字节对齐。

```
#define PACKED __attribute__((packed)) //* 缺省提供了 gcc 编译器的字节对齐方法
#define PACKED_FIELD(x) PACKED x
#define PACKED_BEGIN
#define PACKED_END
```

协议栈源码提供了常用的 gcc 编译器的字节对齐方法。PACKED 宏及 PACKED_BEGIN/PACKET_END 组合体宏通常用于结构体字节对齐定义。二者选其一实现即可。PACKED_FIELD 宏用于定义单个变量字节对齐。**注意，字节对齐定义是整个协议栈能否正常运转的关键**。所以，必须确保该定义能正常工作。

协议栈源码提供了 32 位系统下的基础数据类型定义样例，具体移植时可参考该样例进行调整：

```
//* 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
```

其中 in_addr_t 比较特殊，用于 socket 编程，其为 IPv4 地址类型，其必须是无符号 4 字节整型数。

3. OS 适配层

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。

3.1 os_datatype.h

这个文件负责完成与目标操作系统相关的数据类型定义，主要就是互斥锁、信号量、tty 这三种数据类型的定义。互斥锁用于线程同步，信号量用于线程间通讯，tty 则用于 ppp 模块。我们需要在这个文件里定义能够唯一的标识它们的访问句柄供协议栈使用。

```
typedef INT HMUTEX;        //* 线程互斥（同步）锁句柄
#define INVALID_HMUTEX -1 //* 无效的线程互斥（同步）锁句柄

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

typedef INT HSEM;          //* 信号量，适用与不同线程间通讯
#define INVALID_HSEM -1   //* 无效的信号量句柄
```

注意，上面给出的只是一般性定义，使用时请依据目标 os 的实际情形进行调整。另外，如果你的目标系统不需要 ppp 模块，HTTY 及 INVALID_HTTY 无须定义。

源码工程提供的 os_datatype.h 文件为样例文件。基于协议栈的通用性考虑，样例文件提供的与 os 相关的数据类型定义存在冗余。除上述三种数据类型必须定义外，其它预留的类型如目标系统已提供，建议直接使用目标系统的定义，os_datatype.h 文件中的冗余定义直接注释掉即可；如不存在，则直接使用样例文件中的通用定义即可。

3.2 os_adapter.h

协议栈业务逻辑的完成离不开 os 的支持,这个文件的主要作用就是提供与 os 相关的接口函数声明，然后在 os_adapter.c 中实现这些函数。所以，这个文件中要调整的地方并不多，只有两处。一个是协议栈内部工作线程控制块：

```
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); //* 线程入口函数
    void *pvParam;                        //*  传递给入口函数的用户参数
} STCB_PSTACKTHREAD, *PSTCB_PSTACKTHREAD;
```

这个结构体与目标 os 高度相关，其用于保存协议栈内部工作线程列表。协议栈内部设计了一个 one-shot 定时器。该定时器被用于一些需要等待一小段时间才能进行后续处理或定期执行的业务模块。这个定时器是以线程的方式实现的。协议栈的核心业务逻辑均与这个 one-shot 定时器线程有关。协议栈被目标系统加载时该线程将由 os_thread_onpstack_start()函数自动启动。这个函数要启动的线程列表就被保存在 STCB_PSTACKTHREAD 结构体数组中。这个数组是一个静态存储时期的变量，变量名为 lr_stcbaPStackThread，在 os_adapter.c 中定义。STCB_PSTACKTHREAD 结构体需要定义哪些成员变量由目标 os 提供的线程启动函数的入口参数决定。我们会将线程启动用到的入口参数值定义在 lr_stcbaPStackThread 数组中，然后由 os_thread_onpstack_start()将这些参数值传递给线程启动函数启动相应工作线程。

另外一个地方是临界区保护函数：

```
#define os_critical_init()    //* 临界区初始化
#define os_enter_critical()   //* 进入临界区（关中断）
#define os_exit_critical()    //* 退出临界区（开中断）
```

一般的 os 临界区保护函数基本都是进入临界区关中断，离开临界区开中断。代码非常简单，所以这里直接给出了三个函数宏原型，移植时请依据目标系统具体情形添加对应的开、关中断代码即可。

3.3 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。os_adapter.h 中有这些函数的详细功能说明，移植时按照说明实现具体功能即可，不再赘述。

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

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

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

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

//* 线程同步锁初始化，成功返回同步锁句柄，失败则返回 INVALID_HMUTEX
OS_ADAPTER_EXT HMUTEX os_thread_mutex_init(void);

//* 线程同步区加锁
OS_ADAPTER_EXT void os_thread_mutex_lock(HMUTEX hMutex);

//* 线程同步区解锁
OS_ADAPTER_EXT void os_thread_mutex_unlock(HMUTEX hMutex);

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

//* 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
OS_ADAPTER_EXT HSEM os_thread_sem_init(UINT unInitVal, UINT unCount);

//* 投递信号量
OS_ADAPTER_EXT void os_thread_sem_post(HSEM hSem);

//* 等待信号量到达，参数 nWaitSecs 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；
//* 大于 0，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
OS_ADAPTER_EXT INT os_thread_sem_pend(HSEM hSem, INT nWaitSecs);

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

//* 启动协议栈内部工作线程
OS_ADAPTER_EXT void os_thread_onpstack_start(void *pvParam);

#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
```

在这里仅单独给出 os_thread_onpstack_start()函数的伪代码实现：

```
void os_thread_onpstack_start(void *pvParam)
{
    //* 建立工作线程
    INT i;
    for (i = 0; i < sizeof(lr_stcbaPStackThread) / sizeof(STCB_PSTACKTHREAD); i++)
    {
        //* 在这里添加目标系统提供的线程启动函数，启动工作线程
        ……
    }
}
```

4. 添加网卡

移植的最后一步就是编写网卡驱动然后将网卡添加到协议栈。网卡驱动其本质上完成的是数据链路层的工作，在整个通讯链路上处于通讯枢纽位置，通讯报文的发送和接收均由其实际完成。针对网卡部分的移植工作共三步：

1）编写网卡驱动；

2）注册网卡到协议栈；

3）对接网卡数据收发接口；

协议栈目前支持两种网卡类型：ethernet 和 ppp。两种网卡的移植工作虽然步骤一样，但具体移植细节还是有很大区别的，需要分开单独进行。

4.1 ethernet 网卡

从移植的角度看，ethernet 网卡驱动要提供三个接口函数并完成与协议栈的对接：

1）网卡初始化函数，完成网卡初始及启动工作，并将其添加到协议栈；

2）网卡发送函数，发送上层协议传递的通讯报文到对端；

3）网卡接收函数，接收到达的通讯报文并传递给上层协议；

对于网卡初始化函数，其要做的工作用一句话总结就是：参照网卡数据手册对其进行配置，然后将其注册到协议栈：

```
#define DHCP_REQ_ADDR_EN 1 //* dhcp 请求 ip 地址使能宏
static PST_NETIF l_pstNetifEth = NULL; //* 协议栈返回的 netif 结构
int ethernet_init(void)
{
    /* 进行初始配置，比如引脚配置、使能时钟、相关工作参数配置等工作 */
    //* 在这里添加能够完成上述工作的相关代码，请参照目标网卡的技术手册编写
    ……
    ……
    /* 到这里网卡配置工作完成，但还未启动 */

    //* 添加网卡到协议栈，一定要注意启动以太网卡之前一定要先将其添加到协议栈
    EN_ONPSERR enErr;
#if !DHCP_REQ_ADDR_EN
    ST_IPV4 stIPv4;
    //* 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(……);
    if(!l_pstNetifEth)
    {
#if SUPPORT_PRINTF
        printf("ethernet_add() failed, %s\r\n", onps_error(enErr));
#endif
        return -1;
    }

    //* 启动网卡，开始工作,在这里添加与目标网卡启动相关的代码
    ……
```

```
#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 0;
}
```

上面给出的样例代码中，省略的部分是与目标系统相关的网卡初始配置代码，其余则是与协议栈有关的网卡注册代码。这部分代码主要是完成了两块工作：一，注册网卡到协议栈；二，指定或申请一个静态/动态地址。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：

```
//* 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。
//*           pszIfName: 网卡名称
//*          ubaMacAddr: 网卡 mac 地址
//*             pstIPv4: 指向 ST_IPV4 结构体的指针（include/netif/netif.h），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，如果由 DHCP 分配地址，则参数值为 NULL
//*
//*        pfunEmacSend: 函数指针，指向发送函数，函数原型为 INT(* PFUN_EMAC_SEND)(SHORT sBufListHead, UCHAR *pubErr),
//*                      这个指针指向的其实就是网卡发送函数
//*
//* pfunStartTHEmacRecv: 函数指针，协议栈使用该函数启动网卡接收线程，该线程为协议栈内部工作线程，用户移植时只需提供
//*                     启动该线程的接口函数即可
//*
//*           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
//*
//* 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。
PST_NETIF ethernet_add(const CHAR *pszIfName, const UCHAR ubaMacAddr[ETH_MAC_ADDR_LEN], PST_IPV4 pstIPv4
                       , PFUN_EMAC_SEND pfunEmacSend, void (*pfunStartTHEmacRecv)(void *pvParam),
                       PST_NETIF *ppstNetif, EN_ONPSERR *penErr);
```

ethernet_add()函数提供的参数看起来较为复杂，但其实就完成了一件事情：告诉协议栈这个新增加的

网卡的相关身份信息及功能接口，包括名称、地址、数据读写接口等。这个函数有两个地方需要特别说明：一个是样例代码中该函数的返回值保存在了一个静态存储时期的变量 l_pstNetifEth 中；另一个是入口参数 pfunStartTHEmacRecv。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 thread_ethernet_ii_recv()，该线程在协议栈源码 ethernet.c 文件中实现。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，因此这个指针变量在上面的样例代码中被定义成一个静态存储时期的变量，并确保网卡的接收、发送函数均能访问。pfunStartTHEmacRecv 参数指向的函数要实现的功能与前面我们编写的 os 适配层函数 os_thread_onpstack_start()相同，其就是调用 os 提供的线程启动函数启动 thread_ethernet_ii_recv()线程。比如 rt-thread 下：

```
#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);
}
```

其余 os 与之类似。我们启动的这个以太网接收线程完成实际的以太网层的报文接收及处理工作。其轮询等待网卡接收中断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。我们在后面讲述网卡接收函数的移植细节时还会提到这个接收线程。

对于网卡发送函数，有一点需要注意的是——其原型必须符合协议栈的要求，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。前面在介绍 ethernet_add()注册函数时我们已经给出了发送函数的原型定义，也就是 pfunEmacSend 参数指向的函数原型。前面已经说过，协议栈采用了写时零复制（zero copy）技术，网卡发送函数需要结合协议栈的缓存链表机制编写实现代码，其伪代码实现如下：

```
int ethernet_send(SHORT sBufListHead, UCHAR *pubErr)
{
    SHORT sNextNode = sBufListHead;
    UCHAR *pubData;
    USHORT usDataLen;

    //* 调用 buf_list_get_len()函数计算当前要发送的 ethernet 报文长度，其由协议栈提供
    UINT unEthPacketLen = buf_list_get_len(sBufListHead);

    //* 逐个取出 buf list 节点发送出去
__lblGetNextNode:
    //* 获取下一个节点，buf_list_get_next_node()函数由协议栈提供
    pubData = (UCHAR *)buf_list_get_next_node(&sNextNode, &usDataLen);
    if (NULL == pubData) //* 返回空意味着已经到达链表尾部，没有要发送的数据了，直接返回就可以了
        return (int)unEthPacketLen;

    //* 将数据发送出去，pubData 指向要发送的数据，usDataLen 为其长度，这两个值已经通过 buf_list_get_next_node()函数得到
    //* 在这里添加与具体目标网卡相关的数据发送代码
    ……
```

```
    //* 取下一个数据节点
    goto __lblGetNextNode;
}
```

关于缓存链表，这里以 udp 通讯为例再详细地描述一下其实现机制。当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。ip 报文在 ip 层经过路由选择后被送达数据链路层，也就是 ethernet 层。在这一层，协议栈再将 ethernet ii 报文头挂接到 ip 报文头节点的前面。至此，整个报文的封装完成。协议栈此时会根据网卡注册信息调用对应网卡的 ethernet_send()函数将报文发送出去。ethernet_send()函数的核心处理逻辑就是按照上述机制再依序取出链表节点携带的各层报文数据，然后顺序发送出去。

网卡移植拼图的最后一块就是完成网卡接收函数，把网卡收到的数据推送给协议栈。其伪代码实现如下：

```
//* 网卡接收函数，可以是接收中断服务子函数，也可以是普通函数，普通函数必须确保能够在数据到达的第一时间就能
//* 读取并推送给协议栈
void ethernet_recv(void)
{
    EN_ONPSERR enErr;
    unsigned int unPacketLen;
    unsigned char *pubRcvedPacket;

    //* 在这里添加与具体网卡相关的代码，等待接收报文到达，如果数据到达将报文长度赋值 unPacketLen 变量，将报文
    //* 首地址赋值给 pubRcvedPacket
    ……
    ……
    //* 读取到达报文并将其推送给协议栈进行处理，首先利用协议栈 mmu 模块动态申请一块内存用于保存到达的报文
    unsigned char *pubPacket = (UCHAR *)buddy_alloc(sizeof(ST_SLINKEDLIST_NODE) + unPacketLen, &enErr);
    //* 申请成功，根据协议栈要求，刚才申请的内存按照 PST_SLINKEDLIST_NODE 链表节点方式组织并保存刚刚收到的报文
    PST_SLINKEDLIST_NODE pstNode = (PST_SLINKEDLIST_NODE)pubPacket;
    pstNode->uniData.unVal = unPacketLen;
    memcpy(pubPacket + sizeof(ST_SLINKEDLIST_NODE), (UCHAR *)pubRcvedPacket, unPacketLen);

    //* 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_recv()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
    //* 参数 l_pstNetifEth 为前面注册网卡时由协议栈返回的 PST_NETIF 指针值
    ethernet_put_packet(l_pstNetifEth, pstNode);
}
```

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，它在网卡接收函数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”。接收函数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。前文提到的以太网接收线程 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，ethernet 网卡相关的移植工作完成。

4.2 ppp 拨号网卡

在 Linux 系统，2g/4g/5g 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 栈也是围绕着操作一个标准的 tty 设备来实现底层通讯逻辑的。至于如何操作这个 ppp 拨号终端进行实际的数据收发，tty 层并不关心。所以，协议栈完全借鉴了这个成功的设计思想，在底层驱动与拨号终端之间增加了一个 tty 层，将具体的设备操作与上层的业务逻辑进行了剥离。ppp 拨号网卡的的移植工作其实就是完成 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。tty 层要完成的驱动封装工作涉及的函数原型定义依然是在 os_adapter.h 文件中：

```
#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
//* 返回值为实际发送的字节数
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
//* 返回值为实际收到的数据长度，单位：字节
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 2g/4g/5g 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
```

依据上述函数的原型定义及功能说明，我们在 os_adapter.c 文件编码实现它们，相关伪代码实现如下：

```
#if SUPPORT_PPP
HTTY os_open_tty(const CHAR *pszTTYName)
{
    //* 如果你的系统存在多个 ppp 拨号终端，那么 pszTTYName 参数用于区分打开哪个串口
    //* 在这里添加串口打开代码将连接 2g/4g/5g 模块的串口打开
    ......
    ......

    //* 如果目标系统只有一个拨号终端，那么这里返回的 tty 句柄 x 值为 0，如果目标系统存在多个模块，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 x 应从 0 开始自增，步长为 1
    return x;
}

void os_close_tty(HTTY hTTY)
{
    //* 在这里添加串口关闭代码，关闭哪个串口的依据是 tty 设备句柄 hTTY
    ......
}

INT os_tty_send(HTTY hTTY, UCHAR *pubData, INT nDataLen)
{
    //* 在这里添加数据发送代码，其实就是调用对应的串口驱动函数发送数据到拨号终端，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，返回值为实际发送的字节数
    ......
}

INT os_tty_recv(HTTY hTTY, UCHAR *pubRcvBuf, INT nRcvBufLen, INT nWaitSecs)
{
    //* 同上，在这里添加数据读取代码，其实就是调用对应的串口驱动函数从拨号终端读取数据，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读取数据，返回值为实际读取到的字节数
    ......
}

void os_modem_reset(HTTY hTTY)
{
    //* 在这里添加拨号终端的复位代码，如果你的目标板不支持软件复位模块，可以省略这一步
    //* 复位模块的目的是解决绝大部分的因软件问题产生的故障
    ......
}
#endif
```

参照上述伪代码，依据目标系统具体情况编写相应功能代码即可。注意，上述代码能够正常工作的关键是目标系统的串口驱动必须能够正常工作且健壮、可靠。因为 tty 层封装的其实就是操作 ppp 拨号终端的串口驱动代码，tty 只是做了一层简单封装罢了。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：

```
#if SUPPORT_PPP
//* 连接 ppp 拨号终端的串口名称，有几个模块，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_pszaTTY[PPP_NETLINK_NUM] = { …… /* 如"串口 1", "串口 2"等 */ };

//* 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）指定 tty 设备连接的串口名称；

2）指定拨号认证信息：apn、用户和密码；

3）指定 ppp 链路协商初始值；

总之，你的目标系统连接了几个拨号终端，这几件事情就要针对特定的终端分别做一遍，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 blIsProtoComp;             //* 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 blIsAddrCtlComp;           //* 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 blIsNegoValOfProtoComp;    //* 协议域是否压缩的协商结果（远端设置项，代表对端是否支持该项，协商结果影响该字段）
        BOOL blIsNegoValOfAddrCtlComp; //* 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 unLastRcvedSecs; //* 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。

移植工作的最后一步就是把 ppp 网卡的主处理线程 thread_ppp_handler()添加到 os 适配层的工作线程列表中。也就是前面讲解 os 适配层移植工作时提到的 lr_stcbaPStackThread 数组。这个数组保存了协议栈内部工作线程列表，我们先前已经添加了 one-shot 定时器工作线程 thread_one_shot_timer_count()。我们再把 ppp 主处理线程添加到这个数组中即可。伪代码实现如下：

```
//* 协议栈内部工作线程列表
const static STCB_PSTACKTHREAD lr_stcbaPStackThread[] = {
    { thread_one_shot_timer_count, NULL},
#if SUPPORT_PPP
    //* 在此按照顺序建立 ppp 工作线程，入口函数为 thread_ppp_handler()，线程入口参数为 os_open_tty()返回的 tty 句柄值
    //* 其直接强行进行数据类型转换即可，即作为线程入口参数时直接以如下形式传递：
    //* (void *)句柄值
    //* 不要传递参数地址，即(void *)&句柄，这种方式是错误的
#endif
};
```

ppp 主处理线程将在协议栈加载时由 os 适配层函数 os_thread_onpstack_start()启动。在这里只需把其添加到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主处理线程的入口参数为 tty 句柄。其值应直接传递给线程，不能传递句柄地址（参见上面的伪代码注释）。比如实际移植到目标系统时如果系统只存在一路 ppp，os_open_tty()返回的 tty 句柄值为 0，那么添加到工作线程列表中的 ppp 主处理线程入口参数的值应为“(void *)0”。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。对于单路 ppp，由于 tty 句柄值为 0，所以 ppp 链路的名称为“ppp0”。

至此，ppp 网卡相关的移植工作完成。

5. 移植 telnet 服务器

如果 SUPPORT_ETHERNET 与 NETTOOLS_TELNETSRV 宏同时置位，则 telnet 模块将被加入目标系统。此时我们还需要做一些与目标系统相关的移植工作才能正常使用 telnet 服务器。

5.1 启动 telnet 服务

在目标系统启动 telnet 服务很简单，我们只需把 telnet 服务作为目标系统的一个普通工作线程/任务加入到系统即可，与用户添加自己的 tcp 服务器应用没有任何区别。其线程/任务的入口函数为 telnet_srv_entry()，其源码参见 net_tools/telnet_srv.c 文件。

5.2 移植 NVT 网络虚拟终端

前面说过 NVT 类似传统意义上的控制台终端，一个 telnet 登录客户对应一个 NVT。换言之只要 telnet 客户端与服务器建立连接，协议栈就要启动一个对应的 NVT。NVT 与本地系统高度相关，移植的核心工作其实主要就是完成 NVT 命令的本地化接口。本地化接口文件在源码 port 文件夹下，接口函数声明参见 include/telnet/os_nvt.h，具体实现在 telnet/os_nvt.c 中完成。移植工作涉及的接口函数共有几组。第一组与 nvt 启动相关：

```
//* telnet 服务器作为服务启动时会调用这个函数，完成 NVT 的一些初始工作，比如申请资源等，如果不需要则可以不做任何操
//* 作，注意这个函数在 telnet 服务启动时仅调用一次
void os_nvt_init(void);

//* 关闭 telnet 服务器时调用这个函数，完成 NVT 的去初始化工作，比如回收资源等，同样这个函数仅会在服务关闭时调用一次
void os_nvt_uninit(void);

//* telnet 客户端连接服务器成功后调用，其被用于启动 NVT，其实就是调用目标 OS 提供的线程/任务启动函数将 NVT 作为一个启动，其
//* 线程/任务入口函数为 thread_nvt_handler()，源码参见 net_tools/net_virtual_terminal.c，协议栈提供该入口函数
BOOL os_nvt_start(void *pvParam);

//* telnet 客户端退出登录时如果 NVT 并没有正常结束，协议栈会调用这个函数强制结束 NVT 的运行，对于 RTOS 来说其实就是调用
//* 线程/任务删除函数强制结束
void os_nvt_stop(void *pvParam);
```

第二组与 NVT 命令的本地化接口相关。首先是 ifip 命令：

```
//* 将新添加的 ip 地址吸入本地系统的非易失性存储器，参数 pszIfName 为调用 ethernet_add()函数时注册的网络接口名，下同
BOOL os_nvt_add_ip(const CHAR *pszIfName, in_addr_t unIp, in_addr_t unSubnetMask);

//* 从本地系统的非易失性存储器中删除指定的 ip 地址
BOOL os_nvt_del_ip(const CHAR *pszIfName, in_addr_t unIp);

//* 将修改后的静态 ip 地址写入本地系统的非易失性存储器，注意这个地址是调用 ethernet_add()函数设置的地址，其实际为更新操作
BOOL os_nvt_set_ip(const CHAR *pszIfName, in_addr_t unIp, in_addr_t unSubnetMask, in_addr_t unGateway);

//* 修改指定网络接口的 mac 地址，并将新的 mac 地址更新写入本地系统的非易失性存储器。注意这个函数要完成两项工作：
//* 1. 调用底层驱动，重新设置网卡的 mac 地址；
//* 2. 记录修改后的 msc 地址到非易失性存储器，以便本地系统下次启动时能够自动设置为新的 mac 地址；
BOOL os_nvt_set_mac(const CHAR *pszIfName, const CHAR *pszMac);

//* 将新设置的主从 dns 地址写入本地系统的非易失性存储器
BOOL os_nvt_set_dns(const CHAR *pszIfName, in_addr_t unPrimaryDns, in_addr_t unSecondaryDns);
```

```
//* 将本地系统在非易失性存储器中记录的静态地址模式修改为 dhcp 动态地址模式，以便本地系统下次启动时能够自动申请新的
//* 动态 ip 地址
BOOL os_nvt_set_dhcp(const CHAR *pszIfName);

//* 复位本地系统（修改静态 ip 地址或者删除 ip 地址时需要复位系统）
void os_nvt_system_reset(void);
```

路由表操作命令 route 也需要提供一组本地操作接口：

```
//* 将新增加的路由条目写入本地系统的非易失性存储器
BOOL os_nvt_add_route_entry(const CHAR *pszIfName, in_addr_t unDestination, in_addr_t unGenmask, in_addr_t unGateway);

//* 从本地系统的非易失性存储器中删除指定的路由条目
BOOL os_nvt_del_route_entry(in_addr_t unDestination);
```

ping 命令需要提供一个毫秒级的计时函数：

```
//* 获取本地系统当前记录的毫秒数（连续不间断）
UINT os_get_elapsed_millisecs(void);
```

ntpdate 命令需要提供设置本地系统时间的接口函数：

```
//* 设置本地系统时间为新的时间
void os_nvt_set_system_time(time_t tNtpTime);
```

5.3 NVT 命令扩展

telnet 模块允许用户添加自己的 NVT 命令以满足不同应用场景的需求。用户能够添加的命令分为两种类型：非阻塞型及阻塞型。阻塞型命令需要 NVT 以线程/任务的方式启动，然后独立执行；非阻塞型则相对简单，NVT 直接调用执行即可。源码 port/telnet/nvt_cmd.c 文件中提供了这两种类型命令的代码样例，其中 telnet、ping 命令为阻塞型；reset 为非阻塞型。要想添加自己的命令，我们只需将新的命令添加到 nvt_cmd.c 文件头部定义的 l_staNvtCmd 数组中即可：

```
//* NVT 本地命令数组
static const ST_NVTCMD l_staNvtCmd[] = {
#if NVTCMD_TELNET_EN
    { telnet, "telnet", "used to log in to remote telnet host.\r\n" },
#endif //* #if NVTCMD_TELNET_EN

#if NETTOOLS_PING && NVTCMD_PING_EN
    { nvt_cmd_ping, "ping", "A lightweight ping testing tool that supports IPv4 and IPv6 address probing.\r\n" },
#endif //* #if NETTOOLS_PING && NVTCMD_PING_EN

#if NVTCMD_RESET_EN
    { reset, "reset", "system reset.\r\n" },
#endif //* #if NVTCMD_RESET_EN

    //* 你自己的命令
    { 命令入口函数, "命令名称", "命令说明\r\n" },
    ..... .....
```

```
    { NULL, "", "" } //* 注意这个不要删除，当所有 nvt 命令被用户禁止时其被用于避免编译器报错
};
```

命令入口函数的原型参见 ST_NVTCMD 结构体定义（include/net_tools/net_virtual_terminal.h）：

```
typedef struct _ST_NVTCMD_ {
    INT(*pfun_cmd_entry)(CHAR argc, CHAR* argv[], ULONGLONG ullNvtHandle); //* NVT 命令入口函数
    CHAR *pszCmdName;
    CHAR *pszReadme;
} ST_NVTCMD, *PST_NVTCMD;
```

函数原型的最后一个参数为 NVT 句柄，其唯一的标识了哪一个 NVT 在调用这个命令。

用户的自定义命令添加到数组中后，telnet 服务器在启动时会自动将其注册到 NVT。接下来的工作就是参照协议栈提供的命令样例完成自定义命令的编写即可。需要特别说明的是，NVT 命令的编写需要遵守一定的规则。对于阻塞型命令：

- 凡是携带启动参数的命令，其线程/任务的入口函数必须在启动后第一时间建立一份启动参数的副本（确保启动参数在命令运行期间一直有效）并通知命令入口函数副本建立完毕；
- 凡是携带启动参数的命令，命令入口函数必须等待启动参数的副本建立后（收到通知）才能返回 NVT；
- 由 NVT 提供的 nvt_cmd_exec_enable()函数必须作为命令继续执行的判断条件，一旦函数返回假值，命令应立即结束运行；
- 命令结束运行后应在其返回之前调用 nvt_cmd_exec_end()函数通知 NVT——其已结束运行；
- 命令结束运行的最后一项工作就是调用 nvt_cmd_thread_end()函数通知 NVT——线程/任务也已安全退出；

其中 nvt_cmd_exec_end()函数由 NVT 模块提供，无须过多关注；而 nvt_cmd_thread_end()函数因为涉及线程/任务退出操作，与目标系统高度相关，所以需要用户自己编写实现。因此我们还需要提供一组本地接口用于阻塞型命令的启动与结束：

```
//* 以线程/任务方式启动 NVT 命令，入口参数自行增删
void nvt_cmd_thread_start(……);

//* 杀死当前正在执行的命令。当用户退出登录或者长时间没有任何操作需要主动结束 NVT 运行时，此时如果存在正在执行的命令，NVT 会
//* 先主动通知其结束运行，如果规定时间内其依然未结束运行，NVT 会调用这个函数强制其结束运行以释放占用的线程/任务资源
void nvt_cmd_kill(void);

//* 显式地告知其已结束运行
void nvt_cmd_thread_end(void);
```

其中 nvt_cmd_thread_start()由命令入口函数调用，负责启动命令并等待参数副本建立完毕（如果存在启动参数的话）。nvt_cmd_kill()函数则是提供给 NVT 使用的，用于特殊情况下强制结束命令的执行（对于目标系统为 RTOS 的用户来说其实就是强制删除线程/任务）。

之所以阻塞型命令要求显式地告知线程/任务已安全退出，原因是协议栈运行的目标系统属于资源受限系统，如果同一时刻启动的线程/任务过多，极大可能会导致资源耗尽。所以协议栈在设计上要求阻塞型命令在同一时刻只允许运行一个实例。只有当前实例安全、完整的结束运行，协议栈才会允许建立新的阻塞型实例。nvt_cmd_thread_end()函数的作用即在于此。

对于非阻塞型，其编写规则就比较简单了，只有一条：**在命令尾部调用 nvt_cmd_exec_end()函数显式地通知 NVT——其已结束运行后再返回 NVT。**

与传统意义上的控制台终端类似，NVT 同样提供了一组接口函数实现与 telnet 登录用户的交互操作：

- nvt_output，控制台输出函数

- nvt_outputf，控制台格式化输出函数，类似 printf;
- nvt_input，控制台输入函数，读取用户输入;
- nvt_close，强制 telnet 客户退出登录;

这几个函数的具体使用方法不在本文的描述范围内，请直接参考命令样例，不再赘述。