修改历史		
版本	日期	说明
1.0.0	2022-10-17	针对 onps 栈初始版本提供的 API 接口说明
1.1.0	2023-07-26	1. 增加了新的底层 API; 2. 公开了几个必要的底层 API; 3. 对部分函数原型进行了调整; 4. 增加了新的工具函数;

Onps 栈 API 接口手册

1. 底层 API

由协议栈底层提供的 api，用于涉及底层操作的一些功能实现，这些 api 接口函数的原型定义分布于不同的文件，它们被统一 include 进了 onps.h 中：

- open_npstack_load：将协议栈载入目标系统，协议栈开始运行
- open_npstack_unload：将协议栈载出目标系统，协议栈结束运行
- route_add：添加一条静态路由
- route_del：删除一条静态路由
- route_del_ext：删除指定网卡在路由表中的所有路由条目，禁止网卡跨网段通讯
- route_get_default：获取缺省路由
- dhcp_req_addr：向 dhcp 服务器请求租用一个动态地址
- ethernet_add：添加 ethernet 网卡
- ethernet_del：删除 ethernet 网卡
- ethernet_put_packet：将收到的 ethernet 报文推送给协议栈
- netif_add：添加新的网络接口
- netif_set_default：设置缺省网络接口
- netif_is_ready：网卡是否已就绪（进入工作状态）
- netif_get_by_name：通过网卡名称查找网卡
- netif_get_next：用于连续获取所有已添加到协议栈的网络接口
- netif_eth_set_ip：改为静态地址模式并设置 ip 地址、子网掩码、网关等（仅限 ethernet 网卡）
- netif_eth_set_ip_by_if_name：同 netif_eth_set_ip，只是其通过名称定位网卡
- netif_eth_set_mac_by_if_name：设置新的 MAC 地址（仅限 ethernet 网卡）
- netif_eth_set_dns_by_if_name：设置主从 DNS 地址（仅限 ethernet 网卡）
- netif_eth_add_ip：添加新的 ip 地址（仅限 ethernet 网卡）
- netif_eth_add_ip_by_if_name：同 netif_eth_add_ip，同样只是通过名称定位网卡，下同
- netif_eth_del_ip：删除 ip 地址（仅限 ethernet 网卡）
- netif_eth_del_ip_by_if_name：同 netif_eth_del_ip
- netif_eth_get_next_ip：用于连续获取 ethernet 网卡所有附加 ip 地址信息（仅限 ethernet 网卡）
- netif_eth_get_next_ipv6：用于连续获取 ethernet 网卡所有 ipv6 地址信息（仅限 ethernet 网卡）
- netif_eth_is_static_addr：检查 ethernet 网卡是否为静态地址模式
- is_local_ip：判断 ip 地址是否为本机 ip 地址
- buddy_alloc：申请一块指定大小的内存
- buddy_free：释放申请的内存
- buddy_usage：返回内存管理单元（MMU）buddy 模块内存使用率

- buddy_usage_details：获取 buddy 模块内存使用情况的详细信息
- buf_list_get_next_node：取出下一个链表节点
- buf_list_get_len：链表所有节点携带的数据长度之和
- buf_list_merge_packet：合并链表节点携带的数据将其放入用户指定的缓冲区
- telnet_srv_entry：telnet 服务器作为线程/任务启动的入口函数
- telnet_srv_end：结束 telnet 服务器的运行
- nvt_cmd_add：添加 NVT 命令
- nvt_cmd_exec_end：通知 NVT 命令已结束运行
- nvt_output：类控制台输出函数，将命令执行结果输出给 telnet 客户端
- nvt_outputf：格式化输出函数，类似 printf，将命令执行结果格式化后输出给 telnet 客户端
- nvt_input：类控制台输入函数，读取 telnet 登录用户的输入
- nvt_close：关闭 NVT，强制用户退出 telnet 登录

open_npstack_load

功能
协议栈的入口函数，目标系统调用该函数启动协议栈。换言之，在使用协议栈之前必须首先成功先调用该函数。

原型
```
BOOL open_npstack_load(EN_ONPSERR *penErr);
```

入口参数
➢ penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值
加载成功，返回 TRUE；反之返回 FALSE，具体的错误信息参见参数 penErr 返回的错误码。

示例
```
EN_ONPSERR enErr;
if(open_npstack_load(&enErr)) //* 加载协议栈
{
    //* 协议栈加载成功，在这里添加你的自定义代码
    ……
}
else
    printf("协议栈加载失败，%s\r\n", onps_error(enErr)); //* 打印错误信息
```

open_npstack_unload

功能
协议栈的退出函数。一旦调用该函数，协议栈会结束运行并释放占用的相关系统资源。

原型
```
void open_npstack_unload(void);
```

入口参数
无

返回值
无

示例
略

route_add

功能
添加一条静态路由到路由表。

原型
```
BOOL route_add(PST_NETIF pstNetif, UINT unDestination, UINT unGateway, UINT unGenmask, EN_ONPSERR *penErr);
```

入口参数
- pstNetif：指向网络接口控制块 ST_NETIF 的指针，该指针唯一的标识一个网络接口（也就是网卡），其用于指定静态路由添加到哪个网络接口
- unDestination：目标网段地址，如果其值为 0 则其为缺省路由
- unGateway：网关地址
- unGenmask：子网掩码
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值
添加成功则返回 TRUE，否则返回 FALSE，具体的错误信息参见参数 penErr 返回的错误码。

示例
```
EN_ONPSERR enErr;
PST_NETIF pstNetif = netif_get_by_name("eth0");
if(pstNetif && route_add(pstNetif, inet_addr_small("47.92.239.0"), inet_addr_small("192.168.0.1"),
inet_addr_small("255.255.255.0"), &enErr))
{
    //* 添加成功，在这里添加你的自定义代码
    ……
}
else
    printf("静态路由添加失败，%s\r\n", onps_error(enErr)); //* 打印错误信息
```

route_del

功能
删除一条静态路由。

原型

```
void route_del(UINT unDestination);
```

入口参数

- unDestination：指定要删除的目标网段地址

返回值

无

示例

略

route_del_ext

功能

删除指定网卡在路由表中的所有路由条目，禁止网卡跨网段通讯。

原型

```
void route_del_ext(PST_NETIF pstNetif);
```

入口参数

- pstNetif：指向网络接口控制块 ST_NETIF 的指针，指定要删除哪个网络接口的所有路由条目

返回值

无

示例

略

route_get_default

功能

获取缺省路由信息。

原型

```
PST_NETIF route_get_default(void);
```

入口参数

无

返回值

返回缺省路由绑定的网卡控制块首地址。

示例

略

dhcp_req_addr

功能

启动 dhcp 客户端，向 dhcp 服务器请求租用一个动态地址。

原型

```
BOOL dhcp_req_addr(PST_NETIF pstNetif, EN_ONPSERR *penErr);
```

入口参数

- pstNetif：指定要进行 dhcp 请求的网卡
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值

请求成功则返回 TRUE，否则返回 FALSE，具体的错误信息参见参数 penErr 返回的错误码。

示例

略

ethernet_add

功能

增加新的 ethernet 网卡到协议栈。

原型

```
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);
```

入口参数

- 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
- 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：指向线程启动函数的指针，协议栈使用该指针指向的函数启动协议栈内部工作线程——ethernet 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：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值
注册成功，返回一个 PST_NETIF 类型的指针，该指针指向新添加的网卡控制块，唯一的标识新添加的这块网卡；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。

示例
参见《onps 栈移植手册》第 4.1 节“ethernet 网卡”关于网卡初始函数的示例代码。

ethernet_del

功能
删除 ethernet 网卡。**如果协议栈开启 IPv6 支持，删除网卡时会回收 ipv6 地址资源，回收工作会阻塞当前程序的执行，调用时请注意这一点。**

原型
```
void ethernet_del(PST_NETIF *ppstNetif);
```

入口参数
- ppstNetif：指向网络接口控制块 PST_NETIF 指针的指针，其指向 ethernet_add()函数返回的 PST_NETIF 指针的首地址

返回值
请求成功则返回 TRUE，否则返回 FALSE，具体的错误信息参见参数 penErr 返回的错误码。

示例
略

ethernet_put_packet

功能
将收到的 ethernet 报文推送给协议栈。

原型
```
void ethernet_put_packet(PST_NETIF pstNetif, PST_SLINKEDLIST_NODE pstNode);
```

入口参数
- pstNetif：指向网络接口控制块 ST_NETIF 的指针
- pstNode：指向协议栈 ethernet 网卡接收链表节点的指针

返回值
无

示例
参见《onps 栈移植手册》第 4.1 节“ethernet 网卡”关于网卡接收函数的示例代码。

netif_add

功能
添加一个新的网络接口。

原型
```
PST_NETIF_NODE netif_add(EN_NETIF enType,
                         const CHAR *pszIfName,
                         PST_IPV4 pstIPv4,
                         PFUN_NETIF_SEND pfunSend,
                         void *pvExtra,
                         EN_ONPSERR *penErr);
```

入口参数
- enType：网络接口类型，协议栈目前仅支持两种类型：NIF_PPP、NIF_ETHERNET
- pszIfName：指向网络接口名称的指针
- pstIPv4:指向 ST_IPV4 结构体的指针,其被用于设置该网络接口的静态 ip 地址,如果为 NULL,则需要在 netif_add()函数返回后调用 dhcp_req_addr()函数申请动态 ip 地址
- pfunSend：指向网络接口提供的发送函数的指针，函数原型参见示例部分
- pvExtra：指向网络接口扩展控制信息的指针，不同类型的接口其扩展信息不同，详情参见示例部分
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值
添加成功则返回一个指向 ST_NETIF_NODE 结构体的指针，这是一个链表节点，其被挂接到了网络接口链表上；失败则返回 NULL。

示例
以添加一个 ethernet 网卡为例：

```
//* 回调函数，由协议栈调用，其完成数据包的实际发送工作
//*        pstIf：指向网络接口的指针
//*   ubProtocol：协议类型，其值参见 EN_NPSPROTOCOL（protocols.h 文件）定义
//* sBufListHead：指向缓存链表的首部节点
//* pvExtraParam：指向用户自定义参数的指针
//*       penErr：指向错误编码的指针
typedef INT(*PFUN_NETIF_SEND)(PST_NETIF pstIf, UCHAR ubProtocol, SHORT sBufListHead, void *pvExtraParam, EN_ONPSERR *penErr);

//* 参见 ethernet.c 中该函数的具体实现
INT ethernet_ii_send(PST_NETIF pstNetif, UCHAR ubProtocol, SHORT sBufListHead, void *pvExtraParam, EN_ONPSERR *penErr)
{
    ......
}

//* 具体详见 ethernet_add()函数（ethernet.c 文件）中关于 netif_add()函数的调用示例
EN_ONPSERR enErr;
```

```
PST_NETIF_NODE pstIfNode = netif_add(NIF_ETHERNET, “eth1”, pstIPv4, ethernet_ii_send, pstExtra, &enErr);
……
```

netif_set_default

功能
设置缺省网络接口。缺省网络接口存在的目的是所有目的路由不明确的数据包都将被通过缺省网络接口送出。换言之，这个函数的功能其实就是切换缺省路由。当然我们也可以通过 route_add()函数设置缺省路由。

原型
```
void netif_set_default(PST_NETIF pstNetif);
```

入口参数
- pstNetif：指向网络接口控制块 ST_NETIF 的指针

返回值
无

示例
略

netif_is_ready

功能
网络接口是否已进入正常工作状态。

原型
```
BOOL netif_is_ready(const CHAR *pszIfName);
```

入口参数
- pszIfName：指向网络接口名称的指针

返回值
网络接口已就绪则返回 TRUE，反之为 FALSE

示例
略

netif_get_by_name

功能
通过名称查找网络接口。

原型

```
PST_NETIF netif_get_by_name(const CHAR *pszIfName);
```

入口参数

- pszIfName：指向网络接口名称的指针

返回值

存在则返回一个指向网络接口控制块 ST_NETIF 的指针；否则返回 NULL。

示例

略

netif_get_next

功能

获取下一个网络接口链表节点。

原型

```
const ST_NETIF *netif_get_next(const ST_NETIF *pstNextNetif);
```

入口参数

- pstNextNetif：指向网络接口控制块 ST_NETIF 的指针，如果其值为 NULL 则获取链表首节点指向的网络接口，将首节点赋值给 pstNextNetif 则函数返回链表第二个节点指向的网络接口，以此类推

返回值

返回链表下一个节点指向的网络接口；到达链表尾部则返回 NULL。

示例

```
const ST_NETIF *pstNetif = NULL; //* 如果要从链表首部开始获取所有网络接口这里就要赋值为 NULL
do {
    //* 获取下一个节点（从第一个开始），如果返回 NULL 则说明已到达链表尾部
    if (NULL != (pstNetif = netif_get_next(pstNetif)))
    {
        ......
    }
} while (pstNetif);
```

netif_eth_set_ip

功能

手动设置 ip 地址、网关、子网掩码。注意，如果这之前该网络接口为动态地址，那么调用这个函数后请务必**重启**目标系统，以释放 dhcp 客户端占用的相关资源。

原型

```
void netif_eth_set_ip(PST_NETIF pstNetif, in_addr_t unIp, in_addr_t unSubnetMask, in_addr_t unGateway);
```

入口参数
- pstNextNetif：指向网络接口控制块 ST_NETIF 的指针
- unIp：ip 地址
- unSubnetMask：子网掩码
- unGateway：网关

返回值
无

示例
略

netif_eth_set_ip_by_if_name

功能
手动设置 ip 地址、网关、子网掩码，同 netif_eth_set_ip()。只不过这个函数是通过接口名称定位网卡。

原型
```
BOOL netif_eth_set_ip_by_if_name(const CHAR *pszIfName,
                                 in_addr_t unIp,
                                 in_addr_t unSubnetMask,
                                 in_addr_t unGateway,
                                 CHAR *pbIsStaticAddr,
                                 EN_ONPSERR *penErr);
```

入口参数
- pszIfName：指向网络接口名称的指针
- unIp：ip 地址
- unSubnetMask：子网掩码
- unGateway：网关
- pbIsStaticAddr：用于获取当前网络接口地址模式的指针（TRUE 为静态地址模式，FALSE 为动态地址模式）
- penErr：指向错误编码的指针，当函数执行失败，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

netif_eth_set_mac_by_if_name

功能

设置网卡 MAC 地址。这个设置只是更新了协议栈记录的 MAC 地址，驱动层未设置，所以成功调用该函数后还需要调用 MAC 驱动设置 ethernet 网卡 MAC 地址。否则驱动层和协议栈之间会因为 MAC 地址不同导致网络通讯故障。

原型

```
BOOL netif_eth_set_mac_by_if_name(const CHAR *pszIfName, const CHAR *pszMac, EN_ONPSERR *penErr);
```

入口参数

- pszIfName：指向网络接口名称的指针
- pszMac：指向类似 4E-65-XX-XX-XX-XX 格式的可读字符串的指针
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值

设置成功返回 TRUE；反之则返回 FALSE，具体错误信息参见 penErr 参数携带的错误码。

示例

```
EN_ONPSERR enErr;
if(netif_eth_set_mac_by_if_name(“eth0”, “4E-65-12-34-56-78”, &enErr))
{
    //* 调用 MAC 驱动设置新的 MAC 地址
    ……
}
else
    printf("%s\r\n", onps_error(enErr)); //*打印错误信息
```

netif_eth_set_dns_by_if_name

功能

设置主从 DNS 地址。

原型

```
BOOL netif_eth_set_dns_by_if_name(const CHAR *pszIfName,
                                  in_addr_t unPrimaryDns,
                                  in_addr_t unSecondaryDns,
                                  EN_ONPSERR *penErr);
```

入口参数

- pszIfName：指向网络接口名称的指针
- unPrimaryDns：主 DNS 地址
- unSecondaryDns：从 DNS 地址，如果不设置其值取 0 即可
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值

成功返回 TRUE；反之则返回 FALSE，具体错误信息参见 penErr 参数携带的错误码。

示例
无

netif_eth_add_ip

功能
为采用静态地址模式的 ethernet 网卡添加一个新的 ip 地址。

原型
```
BOOL netif_eth_add_ip(PST_NETIF pstNetif, in_addr_t unIp, in_addr_t unSubnetMask, EN_ONPSERR *penErr);
```

入口参数
- pstNetif：指向网络接口控制块 ST_NETIF 的指针
- unIp：ip 地址
- unSubnetMask：子网掩码
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值
成功返回 TRUE；反之则返回 FALS，具体错误信息参见 penErr 参数携带的错误码。

示例
无

netif_eth_add_ip_by_if_name

功能
功能同 netif_eth_add_ip()函数，唯一区别是该函数采用名称定位网络接口。

原型
```
BOOL netif_eth_add_ip_by_if_name(const CHAR *pszIfName, in_addr_t unIp, in_addr_t unSubnetMask, EN_ONPSERR *penErr);
```

入口参数
- pszIfName：指向网络接口名称的指针
- unIp：ip 地址
- unSubnetMask：子网掩码
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值
成功返回 TRUE；反之则返回 FALS，具体错误信息参见 penErr 参数携带的错误码。

示例
无

netif_eth_del_ip

功能
删除 ip 地址，注意，必须是采用静态地址模式的网络接口才支持删除操作。删除操作有可能会影响当前网络通讯，所以建议 ip 地址被删除后复位目标系统以修复因地址删除可能出现的故障。

原型
```
void netif_eth_del_ip(PST_NETIF pstNetif, in_addr_t unIp);
```

入口参数
- pstNetif：指向网络接口控制块 ST_NETIF 的指针
- unIp：ip 地址

返回值
无

示例
无

netif_eth_del_ip_by_if_name

功能
功能同 netif_eth_del_ip()函数，唯一区别是该函数采用名称定位网络接口。

原型
```
BOOL netif_eth_del_ip_by_if_name(const CHAR *pszIfName, in_addr_t unIp, EN_ONPSERR *penErr);
```

入口参数
- pszIfName：指向网络接口名称的指针
- unIp：ip 地址
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值
删除成功返回 TRUE；反之返回 FALSE，删除失败有两个原因：其一，未找到网络接口；其二，网络接口为动态地址模式。

示例
无

netif_eth_get_next_ip

功能
获取 ethernet 网卡绑定的下一个附加 ip 地址。

原型

```
UINT netif_eth_get_next_ip(const ST_NETIF *pstNetif, UINT *punSubnetMask, UINT unNextIp);
```

入口参数

- pstNetif：指向网络接口控制块 ST_NETIF 的指针
- punSubnetMask：用于获取子网掩码的指针
- unNextIp：要获取的下一个 ip 地址的上一个地址，为 0 代表要获取第一个附加 ip 地址，将获取到的第一个 ip 地址赋值给 unNextIp，函数返回第二个附加 ip 地址，以此类推

返回值

返回 enternet 网卡绑定的下一个附加 ip 地址；到达尾部则返回 0。

示例

```
UINT unNextIp = 0, unSubnetMask; //* 如果要获取所有附加地址，那 unNextIp 就要赋值为 0，意味着我们要从第一个地址开始读取
do {
    //* 获取下一个附加地址（从第一个开始），如果返回 0 则说明已到达尾部
    if (0 != (unNextIp = netif_eth_get_next_ip(pstNetif, &unSubnetMask, unNextIp)))
    {
        ......
    }
} while (unNextIp);
```

netif_eth_get_next_ipv6

功能

获取 ethernet 网卡绑定的下一个 ipv6 地址。

原型

```
UCHAR *netif_eth_get_next_ipv6(const ST_NETIF *pstNetif,
                               UCHAR ubaNextIpv6[16],
                               EN_IPv6ADDRSTATE *penState,
                               UINT *punValidLifetime);
```

入口参数

- pstNetif：指向网络接口控制块 ST_NETIF 的指针
- ubaNextIpv6：指向 ipv6 地址的指针，这个指针指向要获取的下一个地址的上一个 ipv6 地址，作用同 netif_eth_get_next_ip()的 unNextIp 参数
- penState：用于接收 ipv6 地址当前状态的指针，其取值范围参见 EN_IPv6ADDRSTATE 定义（netif.h）
- punValidLifetime：用于接收 ipv6 地址剩余生存时间的指针

返回值

返回 enternet 网卡绑定的下一个 ipv6 地址，其返回值其实就是参数 ubaNextIpv6 指向的地址；到达尾部则返回 NULL。

示例

```
EN_IPv6ADDRSTATE enState;
```

```
UINT unValidLifetime;
UCHAR ubaNextIpv6[16];

//* 如果要获取所有 ipv6 地址，参数 ubaNextIpv6 数组的第一个单元就要赋值为 0，这意味着我们要从第一个 ipv6 地址开始读取
ubaNextIpv6[0]=0;

//* 获取下一个 ipv6 地址（从第一个开始），如果返回 NULL 则说明已到达尾部
while (netif_eth_get_next_ipv6(pstNetif, ubaNextIpv6, &enState, &unValidLifetime))
{
    //* ubaNextIpv6 保存着当前获取的 ipv6 地址（其返回值如果不为空也是这个地址），16 进制格式
    //* enState 保存着当前地址状态，unValidLifetime 则保存着 ipv6 地址的剩余生存时间
    ……
}
```

netif_eth_is_static_addr

功能

检查 ethernet 网卡是否为静态地址模式。

原型

```
BOOL netif_eth_is_static_addr(const CHAR *pszIfName, EN_ONPSERR *penErr);
```

入口参数

- pszIfName：指向网络接口名称的指针
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值

静态地址模式返回 TRUE；返回 FALSE 且参数 penErr 未捕获到任何错误则当前未动态地址模式，否则 penErr 指向具体的错误信息。

示例

```
EN_ONPSERR enErr = ERRNO;
BOOL blRtnVal = netif_eth_is_static_addr(argv[3], &enErr);
if (ERRNO == enErr)
{
    if (blRtnVal)
    {
        printf("当前为静态地址模式\r\n");
    }
    else
    {
        printf("当前为动态地址模式\r\n");
    }
}
else
    printf("%s\r\n", onps_error(enErr)); //* 打印错误信息
```

is_local_ip

功能
检查 ip 是否为本机 ip。

原型
```
BOOL is_local_ip(in_addr_t unAddr);
```

入口参数
- unAddr：ip 地址

返回值
是则返回 TRUE，否则返回 FALSE。

示例
无

buddy_alloc

功能
向内存管理单元（mmu）申请一块指定大小的内存。

原型
```
void *buddy_alloc(UINT unSize, EN_ONPSERR *penErr);
```

入口参数
- unSize：申请分配的内存大小，单位：字节
- penErr：指向错误编码的指针，当函数执行失败，该参数用于接收实际的错误码

返回值
申请成功返回内存首地址；反之则返回 NULL，具体错误信息参见 penErr 参数携带的错误码。

示例
参见 buf_list_merge_packet 函数示例或《onps 栈移植手册》第 4.1 节“ethernet 网卡”关于网卡接收函数的示例代码。

buddy_free

功能
归还（释放）先前通过 buddy_alloc()函数申请的内存。

原型
```
BOOL buddy_free(void *pvStart);
```

入口参数
- pvStart：指向要归还的内存首地址的指针

返回值
成功归还内存返回 TRUE；当 pvStart 指向的并不是 buddy_alloc()函数返回的内存首地址时则返回 FALSE。

示例
参见 buf_list_merge_packet 函数示例。

buddy_usage

功能
统计并计算 buddy 模块当前内存使用率。

原型
```
FLOAT buddy_usage(void);
```

入口参数
无

返回值
返回内存使用率，单精度浮点型。

示例
无

buddy_usage_details

功能
获取 buddy 模块内存使用详情。

原型
```
FLOAT buddy_usage_details(UINT *punTotalBytes,
                          UINT *punUsedBytes,
                          UINT *punMaxFreedPageSize,
                          UINT *punMinFreedPageSize);
```

入口参数
- punTotalBytes：指向动态内存大小的指针，单位：字节
- punUsedBytes：指向已分配内存大小的指针，单位：字节
- punMaxFreedPageSize：指向最大可用内存页大小的指针，单位：字节
- punMinFreedPageSize：指向最小可用内存页大小的指针，单位：字节

返回值

返回内存使用率，单精度浮点型。

示例
参见源码 net_virtual_terminal.c 文件 mem_usage()函数。

buf_list_get_next_node

功能
按链接顺序从链表首部逐个取出链表节点。

原型
```
void *buf_list_get_next_node(SHORT *psNextNode, USHORT *pusDataLen);
```

入口参数
- psNextNode：指向链表下一个节点的指针
- pusDataLen：指向数据长度的指针，出口参数，其用于接收节点携带的数据长度

返回值
如果尚未到达链表尾部，则返回当前链表节点携带的数据的首地址；反之则返回 NULL。

示例
参见《onps 栈移植手册》第 4.1 节“ethernet 网卡”关于网卡发送函数的示例代码。

buf_list_get_len

功能
统计链表所有节点携带的数据长度之和。

原型
```
UINT buf_list_get_len(SHORT sBufListHead);
```

入口参数
- sBufListHead：链表首地址

返回值
返回链表所有节点携带的数据长度之和。

示例
参见 buf_list_merge_packet 函数示例或《onps 栈移植手册》第 4.1 节“ethernet 网卡”关于网卡发送函数的示例代码。

buf_list_merge_packet

功能

从链表首节点开始顺序取出数据将其放入用户指定的缓冲区，把零散的链表数据合并成一块连续数据。

原型

```
void buf_list_merge_packet(SHORT sBufListHead, UCHAR *pubPacket);
```

入口参数

- sBufListHead：链表首地址
- pubPacket：指向用户缓冲区的指针，其用于接收合并后的数据以返回给用户使用

返回值

无

示例

```
EN_ONPSERR enErr;
UINT unDataLen = buf_list_get_len(sBufListHead /* 链表首地址 */);
UCHAR *pubBuf = (UCHAR *)buddy_alloc(unDataLen, &enErr); //* 申请一块缓冲区用于保存合并后的数据
if(pubBuf)
{
    //* 合并，合并后的数据保存在了 pubBuf 中，数据长度为 unDataLen
    buf_list_merge_packet(sBufListHead, pubBuf);

    //* 在这里增加你自己的代码处理合并后的数据
    ……

    //* 处理完毕，释放刚才申请的内存
    buddy_free(pubBuf);
}
```

2. Berkeley sockets

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。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，指向 ipv4 或 ipv6 地址字符串的指针
- 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 ipv4/ipv6 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 *pvCltIP, USHORT *pusCltPort, INT nWaitSecs, EN_ONPSERR *penErr);
```

入口参数
- socket：socket 句柄
- pvCltIP：指向 tcp 客户端地址（ipv4/ipv6）的指针
- pusCltPort：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 hSockClt = accept(hSockSrv, &unCltIP, &usCltPort, 1, &enErr);
        if(INVALID_SOCKET != hSockClt) //* 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。协议栈利用目标 OS 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表示这个函数最长等待多少秒，等待期间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。

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

```
//* 完成 tcp 服务器的数据读取工作
void tcp_server_recv(void *pvData)
{
    SOCKET hSockClt;
```

```
    EN_ONPSERR enErr;
    INT nRcvBytes;
    UCHAR ubaRcvBuf[100];

    while(TRUE)
    {
        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));
        }
    }
}
```

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_srv_connect

功能

这个函数的存在同样是为了简化用户编程。调用该函数将创建 socket 并连接 tcp 服务器。

原型

```
SOCKET tcp_srv_connect(INT family,
                       void *srv_ip,
```

```
                    USHORT srv_port,
                    INT nRcvTimeout,
                    INT nConnTimeout,
                    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
- srv_ip：目标服务器地址
- srv_port：目标服务器端口
- nRcvTimeout：recv()函数的接收等待时长，单位：秒
- nConnTimeout：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）
- 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

连接成功返回 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

略

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

3. telnet 服务

协议栈为 telnet 提供了一组启停函数用于控制服务的启动与结束：

- telnet_srv_entry：telnet 服务器作为线程/任务启动的入口函数
- telnet_srv_end：结束 telnet 服务器的运行
- thread_nvt_handler：网络虚拟终端作为线程/任务启动的入口函数

telnet_srv_entry

功能

telnet 服务的入口函数。其作为函数指针传递给目标系统的线程/任务启动函数，不能直接调用。

原型

```
void telnet_srv_entry(void *pvParam);
```

入口参数

- pvParam：用户自定义参数，在这里为 NULL 即可

返回值

无

示例

略

telnet_srv_end

功能

结束 telnet 服务。调用该函数协议栈会通知 telnet 服务结束运行，释放占用的相关资源，停止对外提供的 telnet 登录服务。

原型

```
BOOL telnet_srv_end(void);
```

入口参数

无

返回值

成功结束则返回 TRUE；反之则返回 FALSE，此时需要调用目标 OS 提供的线程/任务结束函数强制结束 telnet。

示例

略

thread_nvt_handler

功能

网络虚拟终端 NVT 的入口函数。其作为函数指针传递给目标系统的线程/任务启动函数，不能直接调用。其使用方法参见《onps 栈移植手册》5.2 节。

原型

```
void thread_nvt_handler(void *pvParam);
```

入口参数

- pvParam：用户自定义参数，在这里为 NULL 即可

返回值

无

示例
略

4. NVT 命令接口

NVT 模块提供了一组 API 用于开发用户自己的控制台命令。有关 NVT 部分的详细说明参见《onps 栈移植手册》1.2 节。

- nvt_cmd_add：添加 NVT 命令
- nvt_cmd_exec_end：通知 NVT 命令已结束运行
- nvt_output：类控制台输出函数，将命令执行结果输出给 telnet 客户端
- nvt_outputf：格式化输出函数，类似 printf，将命令执行结果格式化后输出给 telnet 客户端
- nvt_input：类控制台输入函数，读取 telnet 登录用户的输入
- nvt_close：关闭 NVT，强制用户退出 telnet 登录

nvt_cmd_add

功能
添加新的 NVT 命令。

原型
```
void nvt_cmd_add(PST_NVTCMD_NODE pstCmdNode, const ST_NVTCMD *pstCmd);
```

入口参数
- pstCmdNode：指向命令节点的指针
- pstCmd：指向命令定义的指针

返回值
无

示例
```
//* 声明一个自定义命令，注意，一定是静态存储时期，因为在 telnet 服务的整个生命周期内均会随时访问
//* 有关 ST_NVTCMD 及 ST_NVTCMD_NODE 的定义参见 net_virtual_terminal.h 文件
static const ST_NVTCMD l_stNvtCmd = { reset, "reset", "system reset.\r\n" };
static ST_NVTCMD_NODE l_stNvtCmdNode;

//* 将命令加入 NVT 命令链表，链表记录的仅仅是命令的地址，所以必须声明命令节点及命令为静态存储时期的变量
nvt_cmd_add(&l_stNvtCmdNode &l_stNvtCmd);
```

nvt_cmd_exec_end

功能
显式地通知 NVT——命令已结束运行。

原型

```
void nvt_cmd_exec_end(ULONGLONG ullNvtHandle);
```

入口参数

- ullNvtHandle：NVT 访问句柄，唯一地标识一个已启动的 NVT 终端。NVT 执行命令时会传入这个句柄值给命令入口函数。详见 ST_NVTCMD 之命令入口函数 pfun_cmd_entry 的原型声明

返回值

无

示例

略

nvt_output

功能

输出函数，向登录到本地 telnet 服务的远程终端发送数据。

原型

```
void nvt_output(ULONGLONG ullNvtHandle, UCHAR *pubData, INT nDataLen);
```

入口参数

- ullNvtHandle：NVT 访问句柄，唯一地标识一个已启动的 NVT 终端
- pubData：指向用户数据的指针
- nDataLen：用户数据长度

返回值

无

示例

略

nvt_outputf

功能

格式化输出函数，向登录到本地 telnet 服务的远程终端发送格式化后的可读字符串信息。

原型

```
void nvt_outputf(ULONGLONG ullNvtHandle, INT nFormatBufSize, const CHAR *pszInfo, ...);
```

入口参数

- ullNvtHandle：NVT 访问句柄，唯一地标识一个已启动的 NVT 终端
- nFormatBufSize：指定用于格式化操作的缓存大小。取值依据为目标字符串的长度，协议栈会据此从 buddy 模块申请一块内存用于格式化操作。如果分配过小会触发内存访问越界故障，所以取值一定要留出裕量
- pszInfo：指向格式化字符串的指针

返回值
无

示例
略

nvt_input

功能
输入函数，从 telnet 远程终端读取用户输入数据。

原型
```
INT nvt_input(ULONGLONG ullNvtHandle, UCHAR *pubInputBuf, INT nInputBufLen);
```

入口参数
- ullNvtHandle：NVT 访问句柄，唯一地标识一个已启动的 NVT 终端
- pubInputBuf：指向接收缓冲区的指针
- nInputBufLen：接收缓冲区长度

返回值
读取到的用户数据实际长度，为 0 意味着没有读取到任何用户输入。

示例
略

nvt_close

功能
关闭 NVT，强制远程用户退出 telnet 登录。

原型
```
void nvt_close(ULONGLONG ullNvtHandle);
```

入口参数
- ullNvtHandle：NVT 访问句柄，唯一地标识一个已启动的 NVT 终端

返回值
无

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