WS63V100 lwIP

开发指南

文档版本　02

发布日期　2024-06-27

前言

概述

本文档介绍了 lwIP（A Lightweight TCP/IP stack）的相关内容，包括 lwIP 简介、应用开发、网络安全和常见问题。本文档主要用于适配 lwIP 和 Huawei LiteOS，同时也用于实现如 DNS 客户端、DHCP 服务器等附加特性。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
WS63	V100

读者对象

本文档主要适用于以下对象：

- 测试工程师
- 软件开发工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明

符号	说明
危险	表示如不避免则将会导致死亡或严重伤害的具有高等级风险的危害。
警告	表示如不避免则可能导致死亡或严重伤害的具有中等级风险的危害。
注意	表示如不避免则可能导致轻微或中度伤害的具有低等级风险的危害。
须知	用于传递设备或环境安全警示信息。如不避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修改记录

文档版本	发布日期	修改说明
02	2024-06-27	更新“3.7 限制条件”章节内容。
01	2024-04-10	第一次正式版本发布。
00B01	2024-03-15	第一次临时版本发布。

目 录

前言 i

1 概述 1

1.1 背景介绍 1

1.2 第三方参考 1

1.3 RFC（Request For Comments）遵从 2

2 功能特性 3

2.1 支持的特性 3

2.2 不支持的特性 6

3 开发指引 9

3.1 前提条件 9

3.2 依赖关系 9

3.3 lwIP 的使用 10

3.4 移植方法 10

3.4.1 lwIP 初始化 10

3.4.2 添加 netif 接口及驱动功能 10

3.5 优化 lwIP 13

3.5.1 优化吞吐量 13

3.5.2 优化内存 14

3.5.3 定制化 14

3.5.4 lwIP 宏 14

3.6 示例代码 20

3.6.1 应用示例代码 20

3.6.1.1 UDP 示例代码 20

3.6.1.2 TCP 客户端示例代码 23
3.6.1.3 TCP 服务端示例代码 25
3.6.1.4 DNS 示例代码 28
3.6.2 驱动相关示例代码 32
3.6.3 服务示例代码 34
3.6.3.1 SNTP 示例代码 34
3.6.3.2 DHCP 客户端示例代码 35
3.6.3.3 DHCP 服务端示例代码 37
3.7 限制条件 39
4 网络安全 45
4.1 网络安全声明 45
5 常见问题 46
A 术语 52

1 概述

1.1 背景介绍

1.2 第三方参考

1.3 RFC（Request For Comments）遵从

1.1 背景介绍

为了实现将智能家居芯片迁移到轻量级操作系统和 TCP/IP 协议栈上，开发了 lwIP，其特点如下：

- Lite Operating System 是 RTOS，意为“轻量级的实时操作系统”；专门针对 Cortex-M series、Cortex-R series、Cortex-A series 芯片架构而设计的 RTOS，该系统主要针对智能终端/穿戴式设备定制，目前已有成熟商业应用。
- 在开源 lwIP 基础上，实现了 DHCP 服务器等附加特性。

1.2 第三方参考

本文档提供客户第三方应用参考如下：

- SNTP 客户端：此 lwIP 没有重新实现 SNTP（Simple Network Time Protocol）客户端，但 lwIP contrib 开源中已经有实现了的 SNTP 客户端，可供用户使用。
- 操作系统适配层代码：lwIP contrib 中有适配到 Linux 平台的代码，LiteOS 支持大部分的 POSIX（Portable Operating System Interface of UNIX）接口，在集成时使用到此代码。

1.3 RFC（Request For Comments）遵从

说明

按照业界协议的标准实现，和其他芯片交互可以正常使用，无私有化的协议，这些协议仅少部分未完全遵从。

lwIP 遵从如下 RFC 协议标准：

- RFC 791 (IPv4 Standard)
- RFC 2460 (IPv6 Standard)
- RFC 768 (UDP) User Datagram Protocol
- RFC 793 (TCP) Transmission Control Protocol
- RFC 792 (ICMP) Internet Control Message Protocol
- RFC 826 (ARP) Address Resolution Protocol
- RFC 1035（DNS）DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION
- RFC 2030（SNTP）Simple Network Timer Protocal（SNTP） Version 4 for IPv4，IPv6 and OSI
- RFC 2131 (DHCP) Dynamic Host Configuration Protocol
- RFC 2018 (SACK) TCP Selective Acknowledgment Options
- RFC 7323 (Window Scaling)
- RFC 6675 (SACK for TCP) RFC 3927 (Autoip) Dynamic Configuration of IPv4 Link-Local Addresses
- RFC 2236 (IGMP) Internet Group Management Protocol, Version 2
- RFC4861 (ND for IPv6) Neighbor Discovery for IP version 6 (IPv6)
- RFC4443 Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification
- RFC4862 IPv6 Stateless Address Autoconfiguration
- RFC2710 Multicast Listener Discovery (MLD) for IPv6

2 功能特性

2.1 支持的特性

2.2 不支持的特性

2.1 支持的特性

lwIP 支持的特性如下：

- 网际协议版本 4（IPv4：Internet Protocol version 4）

  IPv4 是一种无连接的协议，在分组交换网络中使用。此协议会尽最大努力交付数据包，意即它不保证无丢包，也不保证所有数据包均按照正确的顺序无重复地到达。

- 互联网控制消息协议（ICMP：Internet Control Message Protocol）

  互联网协议族的核心协议之一。该协议用于发送“主机不可达”等通知消息，也可以发送“回显请求”和“回显回复”等 Ping 消息。

- 用户数据报协议（UDP：User Datagram Protocol）

  互联网协议族的核心协议之一。该协议使用简单的无连接的传输模型和最小协议机制，这种协议提供无连接的数据包服务，低延迟，可靠性稍弱。因此，该协议会将底层网络协议的不可靠性暴露给用户程序，不能保证无丢包、有序及无重复。UDP 提供校验和用于数据完整性，以及端口号用于寻址数据包的源和目的地等不同功能。

- 传输控制协议（TCP：Transmission Control Protocol）

  互联网协议族的核心协议之一。该协议起源于最初的网络实现，它补充了 IP。因此，整个协议族通常称为 TCP/IP。TCP 为 IP 网络上主机的应用程序提供可靠、有序和纠错的传输。不需要可靠数据流服务的应用程序可以使用 UDP。

- 域名解析器（DNS：Domain Name System）客户端

  DNS 是一个分层的分布式命名系统，用于计算机、服务或连接到互联网或私有网络的资源。lwIP 支持基于 RFC 1035 协议的，特性少的 DNS 客户端（支持域名解析）。

  在 lwip 中，DNS 在 A 记录和 AAAA 记录之间采用了可配置的回退机制。

  回退机制意味着，如果主机名解析到“A”记录失败，那么 DNS 会自动尝试将主机名解析为“AAAA”记录，或反之亦然，而不通知应用程序主机名解析到“A”记录失败。

  DNS 响应依赖于配置参数 LWIP_DNS_ADDRTYPE_DEFAULT。该参数取值如下：

  - LWIP_DNS_ADDRTYPE_IPV4
    DNS 只响应客户端请求解析的主机名的 A 记录。
  - LWIP_DNS_ADDRTYPE_IPV6
    DNS 只响应客户端请求解析的主机名的 AAAA 记录。
  - LWIP_DNS_ADDRTYPE_IPV4_IPV6
    优先解析 A 记录，如果失败，则回落到 AAAA 记录。
  - LWIP_DNS_ADDRTYPE_IPV6_IPV4
    优先解析 AAAA 记录，如果失败，则回落到 A 记录。

  lwip 支持为单个主机名获取多个已解析的 IP 地址。

  - “count”参数中应提供客户端所需解析的 IP 地址数量。
  - “count”的最小值应为 1，如果“count”小于 1，则 DNS 解析失败，返回 ERR _ ARG。
  - “count”的最大值等于可配置的 DNS_MAX_IPADDR 参数。
    如果给定的“count”值大于 DNS_MAX_IPADDR，那么主机名 DNS_MAX_IPADDR 最大解析 IP 地址只会返回给应用程序。

  DNS 接口可以以阻塞或非阻塞的方式调用。

  说明

  lwIP 2.1.3 上的 DNS 不会验证或更改 DNS 服务器提供的已解析的 IP 地址。DNS 只会检查解析后的 IP 地址是否正确。因此，被解析的 IP 地址将直接返回给应用程序。

- 动态主机配置协议（DHCP）客户端和服务器

  一种在 IP 网络上使用的标准化网络协议，用于动态分配网络配置参数，如接口 IP 地址、业务 IP 地址等。通过 DHCP，计算机自动从 DHCP 服务器请求 IP 地址和网络参数，减少了网络管理员或用户手动配置的工作量。lwIP 支持基于 RFC 2131 协议的 DHCP 协议的客户端和服务器。

- 如果 DHCPv4 范围（地址池）使用缺省配置，则 start_ip 和 ip_num 必须为 NULL。
- 如果需要手动配置 DHCPv4 范围（地址池），则 start_ip 和 ip_num 不能为 NULL。
- 无论使用缺省地址范围还是手动配置地址范围，可分配 IP 是否少一个并不决定于是否使用缺省地址范围，而是取决于服务器的 IP 地址是否在地址池的范围内： 如果在地址池的范围内，则少一个；如果不在地址池的范围内，则不会少一个。

  注意：现在缺省地址池的范围是从 2 开始。
- 配置 DHCPv4 范围（地址池）时，如果 DHCPv4 服务器的 IP 地址在（ip_start，ip_start + ip_num）范围内，则客户端池中可用的地址总数将小于 ip_num，因为池中的一个地址将被 DHCPv4 服务器占用。
- DHCPv4 范围内的地址总数将是 ip_num 和 LWIP_DHCPS_MAX_LEASE 的最小值。
- 手动配置 DHCPv4 范围时，如果 ip_start + ip_num 超过 x.y.z.254，则池中的地址总数只能从（ip_start，x.y.z.254）开始， 如果子网掩码是 255.255.0.0 或 255.255.255.128，则上述条件不适用。
- DNS SERVER OPTION 是由 DHCP 服务器返回给客户端的，DISCOVER 和 REQUEST 报文不会携带该 OPTION，在 lwIP 2.1.2 中，该 DNS 服务器地址有以下两种情况：
  - 如果配置了 IPv4 的 DNS 服务器，将会返回配置的 DNS 地址（可能有 2 个）。
  - 如果没有配置 DNS 地址，将会返回 DHCP 服务器的接口 IP 地址。
- DHCPv4 服务器只响应主 DNS 服务器地址，不提供备 DNS 服务器地址；如果系统配置了两个 DNS 地址，则会将 2 个 DNS 服务器地址都返回。

- RFC 2131 偏差标准

  根据 RFC 2131 协议正文的 4.3.2 节，DHCPv4 服务器在 DHCP REQUEST 报文的 Init-Reboot、Renew 或 Rebinding 状态中检查客户端是否填充了 server_id，如果客户端在这些状态下填充了 server_id，则服务器不会接受这些报文。但在 lwIP 2.1.2 中，DHCPv4 服务器没有考虑 server_id 报文的 Init-Reboot、Renew 或 Rebinding 状态。
- 以太网地址解析协议（ARP）

ARP 是一种用于将网络层地址解析为链路层地址的电信协议，是多址网络中的一个重要功能。基于 RFC 826 协议的 Huawei LiteOS lwIP2.1.2 支持 ARP 协议和可配置的 ARP 表。

- 因特网组管理协议（IGMP）

  IGMP 协议用于在主机和本地路由器之间传递组播组成员的信息。lwIP 目前只支持基于 RFC 2236 协议的 IGMP v2 协议。

- socket 接口类型

  lwIP 提供了以下几种类型的 socket 接口：

  - 非线程安全的低级接口
  - 线程安全的 Netconn 接口

    内部调用 Netconn 接口的 BSD（Berkeley Software Distribution）式接口。提供 Berkeley Software Distribution 式接口，帮助应用从 Linux TCP/IP 堆栈平滑迁移到 lwIP TCP/IP 堆栈。

- TCP 选择性确认选项

  此选项用于确认堆栈接收到的失序列段，以便在丢失恢复期间可以跳过这些选择性确认的列段进行重传。

须知

LWIP 相对开源代码优化如下：

- 支持丰富的 AT 调测命令。
- 多项 RFC 遵从性整改，有助于提高对商用路由的兼容性 。
- 提供标准的 tcp sack 流控算法（RFC2018），提高丢包场景下的传输性能 。
- 完善 ipv6 协议支持，如扩展头域处理、Socket Interface Extensions。
- 新增 RDNS 协议和 RPL 协议。

2.2 不支持的特性

lwIP 不支持的特性或协议如下：

- PPPoS/PPPOE
- SNMP 代理（目前，lwIP 仅支持私有 MIB）
- 通过多个网络接口实现 IP 转发

- 路由功能（仅支持终端设备功能）

lwIP 小型化的特性如下：

- 简单网络时间协议（Simple Network Time Protocol）

  SNTP（Simple Network Time Protocol）是一种基于包交换、可变延迟数据网络的计算机系统之间的时钟同步网络协议。lwIP 支持基于 RFC 2030 协议的 SNTP 版本 4。

- lwip_get_conn_info()接口有如下限制：
  - 调用该函数获取 TCP 或 UDP 连接信息。
  - 空指针是 tcpip_conn 类型。
  - 在 LISTEN 状态下，此接口不支持获取 TCP 套接字连接信息。
- TCP 窗口扩大选项

  lwIP 支持基于 RFC 7323 协议的 TCP 窗口扩大选项。
  - TCP 中的窗口扩大选项允许在 TCP 连接中使用 30 位的窗口大小，而不是 16 位。
  - 窗口扩大功能将 TCP 窗口的像素扩展到 30 位，然后使用隐式缩放因子在 TCP 报头的 16 位窗口字段中携带这个 30 位的值。
  - 缩放因子的指数携带在一个名为窗口扩大的 TCP 选项中。
- lwIP 支持 AutoIP 模块。
- lwIP 提供设置 Wi-Fi 驱动状态的接口。
- lwIP 提供设置 Wi-Fi 驱动状态为 lwIP 栈的接口。
  - 当 Wi-Fi 驱动忙时，lwIP 栈停止发送消息；
  - 当 Wi-Fi 驱动 ready 时，lwIP 栈继续发送消息。

  如果在 DRIVER_WAKEUP_INTERVAL 超时之前，驱动程序没有从繁忙状态唤醒，那么所有使用这个 Netif 驱动的 TCP 连接都会被删除。因此，阻塞连接调用将等待 Netif 驱动程序唤醒，SYN（chronize Sequence NumbersSyn）重传，或等待 TCP 连接超时被清除。
- lwIP 在 SOCK_RAW 上提供 PF_PACKET 选项。

  lwIP 支持 PF_PACKET 族的 SOCK_RAW。应用程序可以使用这个特性创建链路层套接字。因此，lwIP 堆栈发送报文时期望应用程序中包含链路层标头。SOCK_RAW 报文在设备驱动之间传递，报文数据无变化。

  默认情况下，所有指定协议类型的报文都被传递到一个报文套接字。若要只从特定接口获取报文，请将该套接字绑定到指定接口。sll _ protocol 和 sll_ifindex 地址

字段用于绑定。当应用程序发送报文时，只需指定 sll_family、sll_addr、sll_halen、sll_ifindex。其他字段应该为 0，收到的报文设置为 sll_hatype 和 sll_pkttype。

3 开发指引

lwIP 支持以太网或 Wi-Fi，应用须根据实际需求配置 lwIP 和适配驱动。

3.1 前提条件

3.2 依赖关系

3.3 lwIP 的使用

3.4 移植方法

3.5 优化 lwIP

3.6 示例代码

3.7 限制条件

3.1 前提条件

lwIP 需满足下列前提条件：

- lwIP 需要优化后方能支持 Huawei LiteOS；目前 lwIP 暂不支持其他操作系统，具体请参见“3.5 优化 lwIP”。
- 使用 lwIP 前，基于 lwIP 相关配置更新驱动代码。详情请参见“3.4 移植方法”中的伪代码。

3.2 依赖关系

lwIP 的依赖关系如下：

- 依赖于 Huawei LiteOS 调用。
- 需要通过以太网或 Wi-Fi 驱动模块完成物理层数据的发送和接收。

3.3 lwIP 的使用

lwIP 提供了 BSD TCP/IP 套接字接口，应用可通过该接口进行连接。lwIP 的优势在于：

- 运行在 BSD TCP/IP 协议栈上的旧应用代码可直接被移植到 lwIP 中。
- lwIP 支持 DHCP 客户端特性，用于配置动态 IP 地址。
- lwIP 支持 DNS 客户端特性，应用可通过该特性解析域名。
- lwIP 占用资源少，可满足协议栈功能。

须知

lwIP 2.1.3 使用 socket 对外 API，close()使用 closesocket()接口替换，ioctl()接口使用 lwip_ioctl()接口替换，避免和 LiteOS 的原生接口产生冲突。

3.4 移植方法

3.4.1 lwIP 初始化

lwIP 的初始化由 tcpip_init()接口完成。该接口接收可选的回调函数及其参数（参数值也可以设为 NULL）。一旦初始化完成，回调函数会被调用，并传入对应的参数。

示例代码如下：

```
void tcpip_init_func(void *arg) {
printf("lwIP initialization successfully done");
}
void lnit_lwIP() {
tcpip_init(tcpip_init_func, NULL);
}
```

3.4.2 添加 netif 接口及驱动功能

初始化完成后，应用必须至少添加一个 Netif 接口用于通信。

应用需要根据驱动类型实现相关回调函数。相关链路层类型、MAC 地址和发送回调函数使用前必须完成注册。如果需要支持原始套接字接口的混杂模式，则还需要在驱动中注册实现该功能的回调函数。

以太网的示例代码如下：

```
struct netif g_netif;
/* user_driver_send mentioned below is the pseudocode for the */
/* driver send function. It explains the prototype for the driver send function. */
/* User should implement this function based on their driver */
void user_driver_send(struct netif *netif, struct pbuf *p) {
/* This will be the send function of the driver */
/* It should send the data in pbuf p->payload of size p->tot_len */
}
/* user_driver_recv mentioned below is the pseudocode for the */
/* driver receive function. It explains how it should create pbuf */
/* and copy the incoming packets. User should implement this function */
/* based on their driver */
void user_driver_recv(char *data, int len) {
/* This should be the receive function of the user driver */
/* Once it receives the data it should do the following*/
  struct pbuf *p = NULL;
  struct pbuf *q = NULL;
p = pbuf_alloc(PBUF_RAW, (len + ETH_PAD_SIZE), PBUF_RAM);
if (p == NULL) {
printf("user_driver_recv : pbuf_alloc failed\n");
return;
}
#if ETH_PAD_SIZE
pbuf_header(p, -ETH_PAD_SIZE); /* drop the padding word */
#endif
memcpy(p->payload, data, len);
#if ETH_PAD_SIZE
pbuf_header(p, ETH_PAD_SIZE); /* reclaim the padding word */
#endif
```

```
driverif_input(&gnetif, p);
}
void eth_drv_config(struct netif *netif, u32_t config_flags,u8_t setBit) {
/* Enable/Disable promiscuous mode in driver code. */
}
/* user_driver_init_func mentioned below is the pseudocode for the */
/* driver initialization function. It explains the lwIP configuration which needs to be */
/* done along with driver initialization. User should implement this function based on
their driver */
void user_driver_init_func() {
ip4_addr_t ipaddr, netmask, gw;
/* After performing user driver initialization operation here */
/* lwIP driver configuration needs to be done*/
#ifndef lwIP_WITH_DHCP_CLIENT
IP4_ADDR(gw, 192, 168, 2, 1);
IP4_ADDR(ipaddr, 192, 168, 2, 5);
IP4_ADDR(netmask, 255, 255, 255, 0);
#endif
g_netif.link_layer_type = ETHERNET_DRIVER_IF;
g_netif.hwaddr_len = ETHARP_HWADDR_LEN;
g_netif.drv_send = user_driver_send;
memcpy(g_netif.hwaddr, driver_mac_address, ETHER_ADDR_LEN);
#if LWIP_NETIF_PROMISC
g_netif.drv_config = eth_drv_config;
#endif
#ifndef lwIP_WITH_DHCP_CLIENT
netifapi_netif_add(&g_netif, ipaddr, netmask, gw);
#else
netifapi_netif_add(&g_netif, 0, 0, 0);
#endif
/* lwIP configuration ends */
}
void Init_Configure_lwIP() {
Init_lwIP();
```

```
    user_driver_init_func();
  netifapi_netif_set_up(&g_netif);
#ifndef lwIP_WITH_DHCP_CLIENT
netifapi_dhcp_start(&g_netif);
do {
msleep(20);
} while (netifapi_dhcp_is_bound(&g_netif) != ERR_OK);
#endif
printf("Network is up !!!\n");
}
```

3.5 优化 lwIP

3.5.1 优化吞吐量

不同的应用场景，为提升 lwIP 吞吐量，建议采用下列优化方案：

- 设置宏 MEMP_NUM_UDP_PCB 的值，即所需 UDP 连接数。

  DHCP 也创建了一个 UDP 连接，所以在设置该宏的值时也要予以考虑。
- 设置宏 MEMP_NUM_TCP_PCB 的值，即所需 TCP 连接数。
- 设置 MEMP_NUM_RAW_PCB 的值，即所需 RAW 连接数。

  LWIP_ENABLE_LOS_SHELL_CMD 模块使用一个 RAW 连接来执行 ping 命令。
- 设置 MEMP_NUM_NETCONN 的值，即所需 UDP、TCP 和 RAW 连接的总数。
  - 如果 LWIP_ENABLE_LOS_SHELL_CMD 和 RAW 连接均未被使用，则禁用 LWIP_RAW。
  - 如果 UDP 未被使用，那么禁用 LWIP_UDP。如果 TCP 未被使用，则禁用 LWIP_TCP。
  - 如果调用驱动接收函数中的 pbuf_alloc()申请 PBUF_RAM 类型内存失败，则增加 MEM_SIZE。
  - 如果 TCPIP mbox 内收入报文空间不可用而导致 driverif_input()调用失败，则增加 TCPIP_MBOX_SIZE 和 MEMP_NUM_TCPIP_MSG_INPKT 的值。如果驱动模块接收报文太快，则也会发生报文空间不可用导致 driverif_input 调用失败。
- 禁用所有调试选项，并且不定义 LWIP_DEBUG。

- 如果架构字长为 4，则设置 ETH_PAD_SIZE 为 2。

3.5.2 优化内存

为节省 lwIP 内存空间，建议采用下列优化方案：

- 调用驱动接收函数中 pbuf_alloc()申请 PBUF_POOL 类型内存，设置 PBUF_POOL_SIZE 的值（即所需 pbuf 数量），并根据 MTU 设置 PBUF_POOL_BUFSIZE 的值。
- 根据所需 TCP、UDP 和 RAW 以及总的连接数，设置 MEMP_NUM_TCP_PCB、MEMP_NUM_UDP_PCB、MEMP_NUM_RAW_PCB 和 MEMP_NUM_NETCONN 的值。
- 根据对端的实际发送数据量，可降低 TCPIP_MBOX_SIZE 和 MEMP_NUM_TCPIP_MSG_INPKT 的值。
- 禁用所有调试选项，并且不定义 LWIP_DEBUG。

  在 lwipopts.h 里面将改为#define LWIP_DBG_TYPES_ON LWIP_DBG_OFF。
- 禁用 ETHARP_TRUST_IP_MAC。
  - 如果启用此宏，则所有接收到的报文都会被用于更新 ARP 表。
  - 如果禁用此宏，则只有进行 ARP 查询才会更新 ARP 表中项目内容。

3.5.3 定制化

在实际应用中，可能会需要定制一些功能，lwIP 中新定义的宏的使用说明如下：

- LWIP_DHCP

  当用户启用 DHCP 服务器时，需要启用 LWIP_DHCPS 宏。如果 LWIP_DHCPS 宏被启用，则也必须启用 LWIP_DHCP。
- LWIP_DHCPS_DISCOVER_BROADCAST

  通常 DHCP 服务器会根据 Discover 消息中由客户端设置的标志，广播或单播 Offer 报文。但如果用户希望一直广播 Offer 消息，则可以启用 LWIP_DHCPS_DISCOVER_BROADCAST 宏。

3.5.4 lwIP 宏

说明

lwIP 2.1.3 的配置宏由于开源很多，未全部描述，请参考开源链接说明：https://www.nongnu.org/lwip/2_1_x/index.html，如需修改，请联系相应接口人。

本节列出了 lwIP 宏，这些宏的描述如下（如需更改，请联系相应接口人）：

- lwIP 宏的默认值在 opt.h 和 lwipopts_default.h 头文件中有定义。
- lwipopts_default.h 头文件中的宏定义会覆盖 opt.h 头文件中的宏定义。

表3-1 宏列表

宏	描述
LWIP_AUTOIP	该宏用于启用或禁用 AUTOIP 模块。
MEM_SIZE	lwIP 可维护堆内存（包括 mem_malloc 和 mem_free）管理模块，该模块用于动态内存分配。该宏用于定义 lwIP 中堆内存管理模块的大小。
MEM_LIBC_MALLOC	如果该宏被启用，那么系统将调用 malloc()和 free()进行所有动态内存分配，而且 lwIP 中的堆内存管理模块代码将会被禁用。
MEMP_MEM_MALLOC	lwIP 为频繁使用的结构提供池内存（包括 memp_malloc 和 memp_free）管理模块。
MEM_ALIGNMENT	该宏的值需要基于架构进行设置。例如，如果是 32 位架构，那么需要设置该宏的值为 4。
MEMP_NUM_TCP_PCB	该宏用于设置同一时间内所需的 TCP 连接数。
MEMP_NUM_UDP_PCB	该宏用于设置同一时间内所需 UDP 连接数。设置该宏的值时，用户须考虑 lwIP 内部模块如 DNS 模块和 DHCP 模块。DHCP 模块会创建 UDP 连接用于自身通信。
MEMP_NUM_RAW_PCB	该宏用于设置同一时间内所需 RAW 连接数。
MEMP_NUM_NETCONN	该宏用于设置 TCP、UDP 和 RAW 连接总数。该宏的值必须是 MEMP_NUM_TCP_PCB、MEMP_NUM_UDP_PCB 和 MEMP_NUM_RAW_PCB 宏的值之和。
MEMP_NUM_TCP_PCB_LISTEN	该宏用于设置同时监听所需 TCP 连接数。
MEMP_NUM_REASSDATA	该宏用于设置同时排队等待重组的 IP 报文数（完整报文，而不是分片报文）。

宏	描述
MEMP_NUM_FRAG_PBUF	该宏用于设置同时发出的 IP 报文数（分片报文，而不是完整报文）。
ARP_TABLE_SIZE	该宏用于设置 ARP 缓存表的大小。
ARP_QUEUEING	如果该宏被启用，那么在 ARP 解析过程中会有多个出报文在排队。如果该宏被禁用，那么只能为每个目的地址保留最近由上层发送的报文。
MEMP_NUM_ARP_QUEUE	该宏用于设置同时排队的出报文（pbuf）的数量。这些出报文正在等待 ARP 响应以解析目的地址。该宏仅在 ARP_QUEUEING 被启用时适用。
ETHARP_TRUST_IP_MAC	如果该宏被启用，那么 ARP 缓存表中源 IP 地址和源 MAC 地址都会被所有的入报文更新。如果该宏被禁用，那么只能通过 ARP 查询来更新 ARP 缓存表中项目内容。
ETH_PAD_SIZE	该宏用于设置在以太网报头之前添加的字节数，以确保在报头之后的荷载对齐。由于报头长度为 14 字节，如果不填充，那么 IP 报头中的地址就不会被对齐。例如，在 32 位架构中，将该宏的值设置为 2 可以使 IP 报头长度对齐 4 字节，也可以加速报文传递。
ETHARP_SUPPORT_STATIC_ENTRIES	该宏支持通过应用的 etharp_add_static_entry()和 etharp_remove_static_entry()接口静态更新 ARP 缓存表。
IP_REASSEMBLY	该宏支持重组 IP 分片报文。
IP_FRAG	该宏支持对所有出报文实施 IP 层分片。
IP_REASS_MAXAGE	该宏用于设置 IP 入报文分割的超时时间。lwIP 可保留 IP_REASS_MAXAGE 秒用于分割报文。如果在该时间内无法重组报文，那么这些分片报文将被丢弃。
IP_REASS_MAX_PBUFS	该宏用于设置任何一个 IP 入报文允许的最大分片数。
IP_FRAG_USES_STATIC_BUF	如果该宏被启用，那么使用静态缓冲区进行 IP 分片；否则，分配动态内存。

宏	描述
IP_FRAG_MAX_MTU	该宏用于设置 MTU 的最大尺寸。
IP_DEFAULT_TTL	传输层数据报文存活时间的默认值。
LWIP_RAND	lwIP 依赖于系统的随机生成器函数，所以须调用强随机数发生器函数进行设置该宏的值。随机数用于为 DNS 和用户 TCP/UDP 连接生成随机客户端端口。此外，随机数也用于在 DHCP 和 DNS 中创建事务 ID 以及在 TCP 中创建 ISS 值。
LWIP_ICMP	该宏用于启用 ICMP 模块。
ICMP_TTL	该宏用于设置 ICMP 消息的 TTL 值。
LWIP_RAW	该宏支持在 lwIP 中启用原始套接字支持功能。
RAW_TTL	该宏用于设置原始套接字消息的 TTL 值。
LWIP_UDP	该宏支持在 lwIP 中启用 UDP 连接支持功能。
UDP_TTL	该宏用于设置 UDP 套接字消息的 TTL 值。
LWIP_TCP	该宏支持在 lwIP 中启用 TCP 连接支持功能。
TCP_TTL	该宏用于设置 TCP 套接字消息的 TTL 值。
TCP_WND	该宏用于设置 TCP 窗口大小。
TCP_MAXRTX	该宏用于设置 TCP 数据报文的最大重传次数。
TCP_SYNMAXRTX	该宏用于设置 TCP SYN 数据报文的最大重传次数。
TCP_FW1MAXRTX	该宏用于设置关闭连接报文（即进入 FIN_WAIT_1 或 CLOSING 状态）的最大重传次数。
TCP_QUEUE_OOSEQ	该宏支持缓存接收到的乱序数据报文。如果用户设备内存低，设置该宏的值为 0。
TCP_MSS	该宏用于设置 TCP 连接的最大分段大小。
TCP_SND_BUF	该宏用于设置 TCP 的发送数据缓冲区大小。
TCP_SND_QUEUELEN	该宏用于设置 TCP 的发送队列长度。
TCP_OOSEQ_MAX_BY	该宏用于设置每个 pcb 的 ooseq 上排队的最大字节数。

宏	描述
TES	默认值为 0（无限制）。仅适用于 TCP_QUEUE_OOSEQ=0。
TCP_OOSEQ_MAX_PBUFS	该宏用于设置每个 pcb 的 ooseq 上排队的最大 pbuf 数。默认值为 0（无限制）。仅适用于 TCP_QUEUE_OOSEQ=0。
TCP_LISTEN_BACKLOG	该宏支持启用 TCP 监听时的 backlog 支持功能。
LWIP_DHCP	该宏用于启用 DHCP 客户端模块。
LWIP_DHCPS	该宏用于启用 DHCP 服务器模块。
LWIP_IGMP	该宏用于启用 IGMP 模块。
LWIP_SNTP	该宏用于启用 SNTP 客户端模块。
LWIP_DNS	该宏用于启用 DNS 客户端模块。
DNS_TABLE_SIZE	该宏用于设置 DNS 缓存表的大小。
DNS_MAX_NAME_LENGTH	该宏用于设置域名支持的最大长度。根据 DNS RFC，域名长度设置为 255。不建议修改。
DNS_MAX_SERVERS	该宏用于设置 DNS 服务器数量。
DNS_MAX_IPADDR	该宏用于设置 DNS 客户端能够缓存的最大 IP 地址。
PBUF_LINK_HLEN	该宏用于设置必须分配给链路级报头的字节数，应该包括实际长度和 ETH_PAD_SIZE。
LWIP_NETIF_API	该宏用于启用线程安全 netif 接口模块（netiapi_*）。
TCPIP_THREAD_NAME	该宏用于设置 TCP IP 线程的名称。
DEFAULT_RAW_RECVMBOX_SIZE	每个 RAW 连接维持一个入报文队列，用于入报文缓存，直至应用层发出接收通知。该宏用于设置接收消息盒子队列的大小。
DEFAULT_UDP_RECVMBOX_SIZE	每个 UDP 连接维持一个入报文队列，用于入报文缓存，直至应用层发出接收通知。该宏用于设置接收消息盒子队列的大小。
DEFAULT_TCP_RECV	每个 TCP 维持一个入报文队列，用于入报文缓存，直至

宏	描述
MBOX_SIZE	应用层发出接收通知。该宏用于设置接收消息盒子队列的大小。
DEFAULT_ACCEPTMBOX_SIZE	该宏用于设置消息盒子队列的大小，以维持接入的 TCP 连接。
TCPIP_MBOX_SIZE	该宏用于设置 TCP IP 线程的消息盒子队列。该队列可以维护应用线程和驱动线程发出的所有操作请求。
LWIP_TCPIP_TIMEOUT	此宏支持启用运行 lwIP tcpip 线程上任意自定义计时器处理函数这一特性。
LWIP_SOCKET_START_NUM	该宏用于设置 lwIP 创建的套接字文件描述符的起始编号。
LWIP_COMPAT_SOCKETS	该宏可以为所有套接字接口创建一个 Linux BSD 宏。
LWIP_STATS	该宏用于统计所有在线连接。
LWIP_SACK	该宏用于启用或禁用 lwIP 内 SACK 功能。要启用发送端和接收端 SACK 功能时，需要启用该宏。
LWIP_SACK_DATA_SEG_PIGGYBACK	该宏用于发送已设置 ACK 标志的数据段中的 SACK 选项。如果该宏被禁用，那么 SACK 选项将只能在空的 ACK 数据段以及出现双向数据转移时的 ACK 中发送，而非数据段。
MEM_PBUF_RAM_SIZE_LIMIT	该宏用于决定是否限制通过 pbuf_alloc()申请 PBUF_RAM 类型内存大小。这限制了 PBUF_RAM 的操作系统内存分配，而非内部 buf 内存的分配。 该宏仅在 MEM_LIBC_MALLOC 被启用时适用。
LWIP_PBUF_STATS	MEM_PBUF_RAM_SIZE_LIMIT 被启用时，该宏可以启用或禁用 PBUF_RAM 内存分配统计的调试打印功能。
PBUF_RAM_SIZE_MIN	需要通过 pbuf_ram_size_set 接口设置的最小 RAM 内存。
DRIVER_STATUS_CHECK	该宏可以通过 netifapi_stop_queue 和 netifapi_wake_queue 接口启用向协议栈发送的驱动发送缓冲区状态通知。

宏	描述
DRIVER_WAKEUP_INTERVAL	当 netif 驱动状态变为 Busy 且在定时器超时前没有恢复到 Ready 状态时，所有链接到此 netif 的 TCP 连接都会被清除。默认值为 120000 毫秒。
PBUF_LINK_CHKSUM_LEN	该宏给出了由以太网驱动填充的链路层校验和的长度。
LWIP_DEV_DEBUG	该宏仅适用于开发人员调试。用户环境中需禁用该宏。

3.6 示例代码

本节列出了部分 lwIP 示例代码，对 lwIP 的用途进行了阐述。lwIP 示例代码包含一个伪代码，说明了在驱动（以太网或 Wi-Fi）模块上需要完成的更改。该示例代码与 lwIP 和 Huawei LiteOS 同时被海思平台交叉编译器编译。

须知

该示例代码禁止直接商用。

3.6.1 应用示例代码

3.6.1.1 UDP 示例代码

```
#include <stdio.h>
#include <netinet/in.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#define STACK_IP "192.168.2.5"
#define STACK_PORT 2277
#define PEER_PORT 3377
#define PEER_IP "192.168.2.2"
#define MSG "Hi, I am lwIP"
#define BUF_SIZE (1024 * 8)
```

```
typedef unsigned     char       u8_t;
typedef signed       int        s32_t;
u8_t g_buf[BUF_SIZE+1] = {0};
int sample_udp() {
s32_t sfd;
struct sockaddr_in srv_addr = {0};
struct sockaddr_in cln_addr = {0};
socklen_t cln_addr_len = sizeof(cln_addr);
s32_t ret = 0, i = 0;
/* socket creation */
printf("going to call socket\n");
sfd = socket(AF_INET,SOCK_DGRAM,0);
if (sfd == -1) {
printf("socket failed, return is %d\n", sfd);
goto FAILURE;
}
printf("socket succeeded\n");
srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr=inet_addr(STACK_IP);
srv_addr.sin_port=htons(STACK_PORT);
printf("going to call bind\n");
ret = bind(sfd,(struct sockaddr*)&srv_addr, sizeof(srv_addr));
if (ret != 0) {
printf("bind failed, return is %d\n", ret);
goto FAILURE;
}
printf("bind succeeded\n");
/* socket creation */
/* send */
cln_addr.sin_family = AF_INET;
cln_addr.sin_addr.s_addr=inet_addr(PEER_IP);
cln_addr.sin_port=htons(PEER_PORT);
printf("calling sendto...\n");
```

```
memset(g_buf, 0, BUF_SIZE);
strcpy(g_buf, MSG);
ret = sendto(sfd, g_buf, strlen(MSG),
0, (struct sockaddr *)&cln_addr,
(socklen_t)sizeof(cln_addr));
if (ret <= 0) {
printf("sendto failed,return is %d\n", ret);
goto FAILURE;
}
printf("sendto succeeded,return is %d\n", ret);
/* send */
/* recv */
printf("going to call recvfrom\n");
memset(g_buf, 0, BUF_SIZE);
ret = recvfrom(sfd, g_buf, sizeof(g_buf), 0,
(struct sockaddr *)&cln_addr, &cln_addr_len);
if (ret <= 0) {
printf("recvfrom failed,
return is %d\n", ret);
goto FAILURE;
}
printf("recvfrom succeeded, return is %d\n", ret);
printf("received msg is : %s\n", g_buf);
printf("client ip %x, port %d\n",
cln_addr.sin_addr.s_addr,
cln_addr.sin_port);
/* recv */
close(sfd);
return 0;
FAILURE:
printf("failed, errno is %d\n", errno);
close(sfd);
return -1;
```

```
}
int main() {
int ret;
ret = sample_udp();
if (ret != 0) {
printf("Sample Test case failed\n");
exit(0);
}
return 0;
}
```

3.6.1.2 TCP 客户端示例代码

```
#include <stdio.h>
#include <netinet/in.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#define STACK_IP "192.168.2.5"
#define STACK_PORT 2277
#define PEER_PORT 3377
#define PEER_IP "192.168.2.2"
#define MSG "Hi, I am lwIP"
#define BUF_SIZE (1024 * 8)
typedef unsigned char u8_t;
typedef signed int s32_t;
u8_t g_buf[BUF_SIZE+1] = {0};
/* Global variable for lwIP Network interface */
int sample_tcp_client() {
s32_t sfd = -1;
struct sockaddr_in srv_addr = {0};
struct sockaddr_in cln_addr = {0};
socklen_t cln_addr_len = sizeof(cln_addr);
s32_t ret = 0, i = 0;
/* tcp client connection */
```

```
printf("going to call socket\n");
sfd = socket(AF_INET,SOCK_STREAM,0);
if (sfd == -1) {
printf("socket failed, return is %d\n", sfd);
goto FAILURE;
}
printf("socket succeeded, sfd %d\n", sfd);
srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = inet_addr(PEER_IP);
srv_addr.sin_port = htons(PEER_PORT);
printf("going to call connect\n");
ret = connect(sfd, (struct sockaddr *)&srv_addr, sizeof(srv_addr));
if (ret != 0) {
printf("connect failed, return is %d\n", ret);
goto FAILURE;
}
printf("connec succeeded, return is %d\n", ret);
/* tcp client connection */
/* send */
memset(g_buf, 0, BUF_SIZE);
strcpy(g_buf, MSG);
printf("calling send...\n");
ret = send(sfd, g_buf, sizeof(MSG), 0);
if (ret <= 0) {
printf("send failed, return is %d,i is %d\n", ret, i);
goto FAILURE;
}
printf("send finished ret is %d\n", ret);
/* send */
/* recv */
memset(g_buf, 0, BUF_SIZE);
printf("going to call recv\n");
ret = recv(sfd, g_buf, sizeof(g_buf), 0);
```

```
if (ret <= 0) {
printf("recv failed, return is %d\n", ret);
goto FAILURE;
}
printf("recv succeeded, return is %d\n", ret);
printf("received msg is : %s\n", g_buf);
/* recv */
close(sfd);
return 0;
FAILURE:
close(sfd);
printf("errno is %d\n", errno);
return -1;
}
int main() {
int ret;
ret = sample_tcp_client();
if (ret != 0) {
printf("Sample Test case failed\n");
exit(0);
}
return 0;
}
```

3.6.1.3 TCP 服务端示例代码

```
#include <stdio.h>
#include <netinet/in.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#define STACK_IP "192.168.2.5"
#define STACK_PORT 2277
#define PEER_PORT 3377
#define PEER_IP "192.168.2.2"
```

```
#define MSG "Hi, I am lwIP"
#define BUF_SIZE (1024 * 8)
typedef unsigned     char       u8_t;
typedef signed       int        s32_t;
u8_t g_buf[BUF_SIZE+1] = {0};
/* Global variable for lwIP Network interface */
int sample_tcp_server() {
      s32_t sfd = -1;
    s32_t lsfd = -1;
struct sockaddr_in srv_addr = {0};
struct sockaddr_in cln_addr = {0};
socklen_t cln_addr_len = sizeof(cln_addr);
s32_t ret = 0, i = 0;
/* tcp server */
printf("going to call socket\n");
lsfd = socket(AF_INET,SOCK_STREAM,0);
if (lsfd == -1) {
printf("socket failed, return is %d\n", lsfd);
goto FAILURE;
}
printf("socket succeeded\n");
srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = inet_addr(STACK_IP);
srv_addr.sin_port = htons(STACK_PORT);
ret = bind(lsfd, (struct sockaddr *)&srv_addr, sizeof(srv_addr));
if (ret != 0) {
printf("bind failed, return is %d\n",    ret);
goto FAILURE;
}
ret = listen(lsfd, 0);
if (ret != 0) {
printf("listen failed, return is %d\n", ret);
goto FAILURE;
```

```
}
printf("listen succeeded, return is %d\n", ret);
printf("going to call accept\n");
sfd = accept(lsfd, (struct sockaddr *)&cln_addr, &cln_addr_len);
if (sfd < 0) {
printf("accept failed, return is %d\n", sfd);
}
printf("accept succeeded, return is %d\n", sfd);
/* tcp server */
/* send */
memset(g_buf, 0, BUF_SIZE);
strcpy(g_buf, MSG);
printf("calling send...\n");
ret = send(sfd, g_buf, sizeof(MSG), 0);
if (ret <= 0) {
printf("send failed, return is %d,
i is %d\n", ret, i);
goto FAILURE;
}
printf("send finished ret is %d\n", ret);
/* send */
/* recv */
memset(g_buf, 0, BUF_SIZE);
printf("going to call recv\n");
ret = recv(sfd, g_buf, sizeof(g_buf), 0);
if (ret <= 0) {
printf("recv failed, return is %d\n", ret);
goto FAILURE;
}
printf("recv succeeded, return is %d\n", ret);
printf("received msg is : %s\n", g_buf);
/* recv */
close(sfd);
```

```
close(lsfd);
return 0;
FAILURE:
close(sfd);
close(lsfd);
printf("errno is %d\n", errno);
return -1;
}
int main() {
int ret;
ret = sample_tcp_server();
if (ret != 0) {
printf("Sample Test case failed\n");
exit(0);
}
return 0;
}
```

3.6.1.4 DNS 示例代码

```
#include <netdb.h>
#include "lwip/opt.h"
//#include "lwip/sockets.h"
//#include "lwip/netdb.h"
#include "lwip/err.h"
#include "lwip/inet.h"
#include "lwip/dns.h"
int gmutexFail;
int gmutexFailCount;
struct gethostbyname_r_helper {
ip_addr_t *addr_list[DNS_MAX_IPADDR+1];
ip_addr_t addr[DNS_MAX_IPADDR];
char *aliases;
};
void dns_call_with_unsafe_api() {
```

```
struct hostent *result = NULL;
int i = 0;
ip_addr_t *addr = NULL;
char addrString[20] = {0};
char *hostname;
char *dns_server_ip;
ip_addr_t dns_server_ipaddr;
hostname = "www.huawei.com";
dns_server_ip = "192.168.0.2";
inet_aton(dns_server_ip, &dns_server_ipaddr);
dns_setserver(0, &dns_server_ipaddr);
result = gethostbyname(hostname);
if (result)
{
while (1)
{
addr = *(((ip_addr_t **)result->h_addr_list) + i);
if (addr == NULL)
{
break;
}
inet_ntoa_r(*addr, addrString, 20);
printf("dns call for %s, returns %s\n",
hostname, addrString);
i++;
}
}
else
{
printf("dns call failed\n");
}
}
void dns_call_with_safe_api()
```

```
{
int i = 0;
ip_addr_t *addr = NULL;
char addrString[20] = {0};
char *buf = NULL;
int buflen;
char *hostname = NULL;
char *dns_server_ip = NULL;
ip_addr_t dns_server_ipaddr;
struct hostent ret;
struct hostent *result = NULL;
int h_errnop;
hostname = "www.huawei.com";
dns_server_ip = "192.168.0.2";
inet_aton(dns_server_ip, &dns_server_ipaddr);
lwip_dns_setserver(0, &dns_server_ipaddr);
buflen = sizeof(struct gethostbyname_r_helper) +
strlen(hostname) + MEM_ALIGNMENT;
buf = malloc(buflen);
gethostbyname_r(hostname, &ret, buf, buflen, &result, &h_errnop);
if (result)
{
while (1)
{
addr = *(((ip_addr_t **)result->h_addr_list) + i);
if (addr == NULL)
{
break;
}
inet_ntoa_r(*addr, addrString, 20);
printf("dns call for %s, returns %s\n",
hostname, addrString);
i++;
```

```
}
}
else
{
printf("dns call failed\n");
}
free(buf);
}
/* To get the dns server address configured in lwIP */
/* Application can call dns_getserver() API and then decide whether to
change it */
/* If application needs to change it then, it needs */
void display_dns_server_address()
{
int i;
ip_addr_t addr;
int ret;
char addrString[20] = {0};
for (i = 0; i < DNS_MAX_SERVERS; i++)
{
ret = lwip_dns_getserver(i, &addr);
if (ret != ERR_OK)
{
printf("dns_getserver failed\n");
return;
}
memset(addrString, 0, sizeof(addrString));
inet_ntoa_r(addr, addrString, 20);
printf("dns server address configured \
at index %d, is %s\n", i,
addrString);
}
return;
```

```
}
int main() {
/* after doing lwIP init, driver init and netifapi_netif_add */
display_dns_server_address();
dns_call_with_unsafe_api();
dns_call_with_safe_api();
return 0;
}
```

说明

lwIP 内部通过 dns_servers 全局变量存储和管理 DNS 地址，通过 DNS_MAX_SERVERS 宏控制 DNS 存储的个数；在 IPv4 组网下，设备关联 AP 之后，通过 DHCP 的方式获取到 DNS 地址，通常是 AP 的网关地址；在 IPv4 和 IPv6 组网场景下，DNS 地址除了来源于 DHCP 报文，还会来源于 IPv6 组网内的 RA 报文，这种组网场景下，设备可能同时存在 IPv4 和 IPv6 的 DNS 地址，协议栈默认使用 dns_servers 数组索引 0 位置的地址(IPv4 或者 IPv6)，应用需要明确使用 v4 还是 v6 的 DNS 地址。

3.6.2 驱动相关示例代码

```
#include <string.h>
#include "lwip/ip.h"
unsigned char SUT_MAC[6] = { 0x46, 0x44, 0x2, 0x2, 0x3, 0x3 };
#define ETHER_ADDR_LEN 6
/* Global variable for lwIP Network interface */
struct netif g_netif;
/* user_driver_send mentioned below is the pseudocode for the */
/* driver send function. It explains the prototype for the driver send function.
*/
/* User should implement this function based on their driver */
void user_driver_send(struct netif *netif, struct pbuf *p)
{
/* This will be the send function of the
driver */
/* It should send the data in pbuf
p->payload of size p->tot_len */
}
```

```
void user_driver_init_func()
{
ip4_addr_t ipaddr, netmask, gw;
/* After performing user driver init
operation */
/* lwIP driver configuration needs to be
done*/
/* lwIP configuration starts */
IP4_ADDR(&gw, 192, 168, 2, 1);
IP4_ADDR(&ipaddr, 192, 168, 2, 5);
IP4_ADDR(&netmask, 255, 255, 255, 0);
g_netif.link_layer_type = ETHERNET_DRIVER_IF;
g_netif.hwaddr_len = ETHARP_HWADDR_LEN;
g_netif.drv_send = user_driver_send;
memcpy(g_netif.hwaddr, SUT_MAC, ETHER_ADDR_LEN);
netifapi_netif_add(&g_netif, &ipaddr, &netmask, &gw);
netifapi_netif_set_default(&g_netif);
/* lwIP configuratin ends */
}
/* user_driver_recv mentioned below is the pseudocode for the */
/* driver receive function. It explains how it should create pbuf */
/* and copy the incoming packets. User should implement this function */
/* based on their driver */
void user_driver_recv(char * data, int len)
{
/* This should be the receive function of
the user driver */
/* Once it receives the data it should do
the below */
struct pbuf *p;
p = pbuf_alloc(PBUF_RAW, (len +
ETH_PAD_SIZE), PBUF_RAM);
if (p == NULL)
```

```
{
printf("user_driver_recv : pbuf_alloc \
failed\n");
return;
}
#if ETH_PAD_SIZE
pbuf_header(p, -ETH_PAD_SIZE); /* drop the
padding word */
#endif
memcpy(p->payload, data, len);
#if ETH_PAD_SIZE
pbuf_header(p, ETH_PAD_SIZE); /* reclaim the
padding word */
#endif
driverif_input(&g_netif,p);
}
int main()
{
/* Call lwIP tcpip_init before driver init*/
tcpip_init(NULL, NULL);
user_driver_init_func();
return 0;
}
```

3.6.3 服务示例代码

3.6.3.1 SNTP 示例代码

```
#include "lwip/opt.h"
//#include "lwip/sntp.h"
/* Compile time configuration for SNTP
* 1) Configure SNTP server address
*/
int gmutexFail;
int gmutexFailCount;
```

```
int start_sntp()

{

int ret;

int server_num = 1;    /*Number of SNTP servers available*/

char *sntp_server = "192.168.0.2"; /*sntp_server : List of the available servers*/

struct timeval time_local; /*Output Local time of server, which will be received in NTP response from server*/

memset(&time_local, 0, sizeof(time_local));

ret = lwip_sntp_start(server_num, &sntp_server, &time_local);

printf("Recevied time from server = [%li]sec [%li]u sec\n", time_local.tv_sec, time_local.tv_usec);

/* After the SNTP time synchronization is complete, the time calibration is not performed periodically.

*/

return ret;

}

int main()

{

/* after doing lwIP init, driver init and

netifapi_netif_add */

start_sntp();

return 0;

}
```

3.6.3.2 DHCP 客户端示例代码

```
#include <unistd.h>

#include "lwip/opt.h"

#include "lwip/netifapi.h"

#include "lwip/inet.h"

//#include "lwip/netif.h"

struct netif g_netif;

int gmutexFail;

int gmutexFailCount;

int dhcp_client_start(struct netif *pnetif)

{
```

```
int ret;
char addrString[20] = {0};
/* Calling netifapi_dhcp_start() will start
initiating DHCP configuration
* process by sending DHCP messages */
ret = netifapi_dhcp_start(pnetif);
if (ret == ERR_OK)
{
printf("dhcp client started \
successfully\n");
}
else
{
printf("dhcp client start failed\n");
}
/* After doing this it will get the IP and
update to netif, once it finishes
the process with DHCP server. Application
need to call netifapi_dhcp_is_bound()
API to check whether DHCP process is
finished or not */
do
{
sleep(1); /* sleep for sometime,
like 1 sec */
ret = netifapi_dhcp_is_bound(pnetif);
} while(ret != ERR_OK);
memset(addrString, 0, sizeof(addrString));
inet_ntoa_r(pnetif->ip_addr, addrString, 20);
printf("ipaddr %s\n", addrString);
memset(addrString, 0, sizeof(addrString));
inet_ntoa_r(pnetif->netmask, addrString, 20);
printf("netmask %s\n", addrString);
```

```
memset(addrString, 0, sizeof(addrString));
inet_ntoa_r(pnetif->gw, addrString, 20);
printf("gw %s\n", addrString);
return 0;
}
int main()
{
/* after doing lwIP init, driver init and
netifapi_netif_add */
dhcp_client_start(&g_netif);
/* Later if application wants to stop the
DHCP client then it should
call netifapi_dhcp_stop() and
netifapi_dhcp_cleanup() */
/* netifapi_dhcp_stop(&g_netif); */
/* netifapi_dhcp_cleanup(&g_netif); */
return 0;
}
```

3.6.3.3 DHCP 服务端示例代码

```
#include "lwip/opt.h"
#include "lwip/netifapi.h"
#include "lwip/inet.h"
#include "lwip/netif.h"
struct netif g_netif;
int gmutexFail;
int gmutexFailCount;
int dhcp_server_start(struct netif *pnetif, char *startIP, int ipNum)
{
int ret;
/* Calling netifapi_dhcps_start() will start DHCP server */
if ( startIP == NULL )
{
/* For Automatic Configuration */
```

```
ret = netifapi_dhcps_start(pnetif, NULL, NULL);
}
else
{
/* For Manual Configuration */
ret = netifapi_dhcps_start(pnetif, startIP, ipNum);
}
if (ret == ERR_OK)
{
printf("dhcp server started successfully\n");
}
else
{
printf("dhcp server start failed\n");
}
}
int main()
{
/* after doing lwIP init, driver init and netifapi_netif_add */
/* DHCP Server Address Pool Configuration */
char *startIP; // IP from where the DHCP Server address pool has to start
int ipNum;       // Number of IPs that is to be offered by DHCP Server starting from
startIP
/*****************************************/
char manualDHCPServerConfiguration = 'Y';
if ( manualDHCPServerConfiguration == 'Y' )
{
/* For Manual Configuration */
startIP = "192.168.0.5";
ipNum = 15;
}
else
{
/* For Automatic Configuration */
```

```
startIP = NULL;
ipNum = 0;
}
dhcp_server_start(&g_netif, startIP, ipNum);
/* Later if application wants to stop the DHCP client then it should call
netifapi_dhcps_stop()*/
/*netifapi_dhcps_stop(&g_netif);*/
return 0;
}
```

3.7 限制条件

在使用 lwIP 之前，需要考虑以下限制条件：

- lwIP 内的操作系统适配层与 Huawei LiteOS 接口紧耦合，所以 lwIP 只支持在 Huawei LiteOS 上运行。
- lwIP 目前的 RAM 和 ROM 进一步裁剪，通过配置 lwipopts.h 中的宏控制，更多限制条件请参见 lwipopts.h 代码宏 LWIP_SMALL_SIZE。
- DHCP 客户端允许 UDP 套接字接口绑定在 DHCP 客户端端口上，而 lwIP 不允许多个网口（包括以太网和 Wi-Fi 的 netif 结构体）绑定到同一个端口。因此，如果 SO_BINDTODEVICE 被禁用，则 lwIP 可以与两个网口（以太网和 Wi-Fi）同时运行，但 DHCP 客户端不可以。
- DNS 客户端仅支持响应消息中的 A 或 AAAA 类资源记录。解析 DNS 响应消息中的多个应答记录时，如果遇到任何异常的应答记录，则解析终止并返回已成功解析的记录（如果存在成功解析的记录）；否则返回解析失败的记录。
- lwIP 提供了以下类型的接口：
  - BSD 接口

    均能够确保线程安全。
  - netconn 接口

    线程安全接口。
  - 低阶接口

    低阶接口不是线程安全接口。因此，不建议使用该接口。如果用户使用了这种接口，应用线程和驱动线程需要注意锁定核心 TCPIP 线程功能。
- lwIP 的多线程使用有如下限制：

  - lwIP 核心不是线程安全。如果多线程环境中的应用需要使用 lwIP，则应使用接口层（例如：netconn 或 socket 接口层）。但如果使用了低阶接口，则应保护 lwIP 核心。
  - netif_xxx 和 dhcp_xxx 不是线程安全接口。因此应用应该使用 netifapi 模块中可用的线程安全接口 netifapi_netif_xxx 和 netifapi_dhcp_xxx。

    对于 lwip 接口的线程安全性，说明如下：

    - 所有的 socket BSD API 都是线程安全的。
    - 所有形如 netifapi_xxx 的 API 都是线程安全的。
    - 所有形如 netif_xxx 的 API 都是非线程安全的。
- 由于网口维护所需接口索引的变化范围为 1 ~ 254，因此应用创建的网口数不能超过 254。
- lwip_select 和 closesocket 接口的多线程考虑：

  如果某个套接字受到 lwip_select()接口监测，并在另一线程被 closesocket()接口关闭，则 lwIP 中的 lwip_select()接口会从锁闭状态返回，不标记具体套接字。而 Linux 中，在另一线程关闭套接字对 lwip_select()接口没有影响。
- lwip_shutdown()接口有如下限制：
  - 如果通过 SHUT_RDWR 或 SHUT_RD 标志调用了 lwip_shutdown()接口，则任何待接收的数据应由 lwIP 清除，RST 被发送给对端，并且应用在调用 SHUT_RDWR 或 SHUT_RD 之前必须读取数据。
  - 当发送被阻塞且 lwip_shutdown(SHUT_RDWR)被调用时，将返回 EINPROGRESS (115)。
- lwip_listen()接口有如下限制：
  - Backlog 的最大值为 16，最小值为 0。
  - 如果 backlog 数值≤0，则 backlog 值使用 1。
  - 监听操作中，lwIP 不支持自动绑定，所以 lwip_bind()必须在 lwip_listen()之前调用。
  - lwip_listen()接口支持多路调用。如果 socket 已开启监听，则 socket 会更新监听 backlog。
  - 新的 backlog 数值只适用于新的输入连接请求。
- lwip_recv()接口有如下限制：
  - 当 lwIP 接收下一个期望的数据段时，lwIP 会更新接收缓存列表。

- 如果与接收数据段相邻的是一个乱序数据段，lwIP 会合并这两个数据段为一个，并放入接收缓存列表。
- UDP 或 RAW 套接字不支持 MSG_PEEK。
- UDP 或 RAW 不会报告接收包内存分配失败。
- 不支持 MSG_WAITALL，支持 MSG_DONTWAIT。
- 设置 SO_RECVTIMEO 套接字选项，socket 不会被标记为 O_NONBLOCK（无阻塞），由于 socket 超时时间，会接收非数据，返回错误并显示 ETIMEDOUT 集。

- recvfrom()接口有如下限制：
  - UDP 或 RAW 套接字不支持 MSG_PEEK。
  - UDP 或 RAW 不会报告接收包内存分配失败。
  - 不支持 MSG_WAITALL，支持 MSG_DONTWAIT。
  - 在 TCP 套接字中，如果可以实现，recvfrom()会尝试接收所有来自接收缓存器的数据。
  - TCP 接收缓存器是一个列表，保留从对端接收的数据段。如果应用调用 recv 函数来获取数据，该函数会从列表中获取首个条目并将它传回给应用。该函数不重复接收来自列表的条目来填充完整用户缓冲区。
  - 当 lwIP 接收下一个期望的数据段时，lwIP 会更新接收缓存列表。如果与接收数据段相邻的是个乱序数据段，lwIP 会合并这两个数据段为一个，并放入接收缓存列表。
  - 如果“length”参数设为零，则协议栈将返回的值为-1，且 errno 将设为 EINVAL。但是，POSIX 规范未对该设置进行明确说明。如果 lwip_recv()的 Linux 实现出现偏差，则当缓存大小设为 0 时，会返回值 0。
  - 不支持 MSG_TRUNC 标志。
- lwip_send()接口和 lwip_sendmsg()接口有如下限制：
  - UDP 和 RAW 连接能发送的数据长度最大为 65332。如果发送更长的数据，返回的值为-1，且 errno 将设为 ENOMEM。
  - 只支持 MSG_MORE 和 MSG_DONTWAIT 标志。不支持如 MSG_OOB/MSG_NOSIGNAL/MSG_EOR 的其他标志。
- lwip_sendto()接口有如下限制：
  - AF_INET/AF_INET6 UDP 和 RAW 连接能发送的数据长度最大为 65332。如果发送更长的数据，返回失败，值为-1，errno 将设为 ENOMEM。

- 只支持 MSG_MORE 和 MSG_DONTWAIT 标志。不支持如 MSG_OOB/MSG_NOSIGNAL/MSG_EOR 的其他标志。

- socket()接口有如下限制：
  - 只有 SOCK_RAW 支持 PF_PACKET。
  - AF_INET 套接字支持 SOCK_RAW | SOCK_DGRAM | SOCK_STREAM 类型。
  - AF_PACKET 只支持 SOCK_RAW 类型。
- lwip_write()接口有如下限制：
  - 对于没有标记 O_NONBLOCK、设置了 SP_SENDTIMEO 选项且历时时间比超时时间长的 socket，lwIP 失败并且提示 errno EAGAIN。
- lwip_select()接口有如下限制：
  - select()接口不会更新超时参数来显示剩余时长。
  - FD_SETSIZE 是 lwIP 中可配置的编译时间，应用必须确保不违反这个边界值，lwIP 不在运行过程中对此进行验证。
- lwip_fcntl()接口有如下限制：
  - 只支持 F_GETFL and F_SETFL 命令。对于 F_SETFL，val 只支持 O_NONBLOCK。
  - PF_PACKET 套接字支持 F_SETFL 和 F_GETFL 选项。
- 任何方式的 IP 地址变更会导致如下行为：
  - TCP
    - 所有现有的 TCP 连接掉线，此种连接上任何操作均显示 ECONNABORTED。
    - 所有界限地址被更改为新的 IP 地址。
    - 如果有一个监听套接字，它会在新 IP 地址上接受连接。
  - UDP
    - 所有套接字的 IP 地址更改为新 IP 地址，通信会在新 IP 地址上继续。
    - 多种绑定调用行为：bind()接口调用会更改 UDP 套接字的端口绑定。
      - UDP 套接字会更改本地绑定端口。
      - TCP 套接字上不允许，如果执行多种绑定调用，则会返回失败，并显示 errno EINVAL。
      - 在 AF_INET/AF_INET6 SOCK_RAW 套接字上，多种绑定调用行为会更新本地 IP 地址。

  - 在 PF_PACKET 套接字上，多种绑定调用行为会更改绑定为新的接口编号。
  - AF_INET/AF_INET6 SOCK_RAW 绑定不会检查套接字地址是否可用。

- 以下所列接口不支持 PF_PACKET 选项：
  - lwip_accept()
  - lwip_shutdown()
  - lwip_getpeername()
  - lwip_getsockname()
  - lwip_listen()
- 对于 ping6，如果应用创建原始 ICMPV6 消息，协议栈不会储存这些消息的统计数据。应用必须保留计数并处理这些统计数据。
- 当数据包发往一个链路本地地址时，必须设置当前 netif 的链路本地地址。否则，路由会返回，显示网络不可达。
- 支持 IPv6 邻居发现功能解析选项，这些解析选项定义在 RFC 4861 中。为了确保后续扩建与现有实现的合理共存，协议栈忽略接收的 ND 数据包中任何不可识别的选项，并继续处理数据包。执行验证，对可能影响后续扩建的具体类型进行程序检查。
- 当网络掩码配置为 0.0.0.0 时，以下限制可适用：

  不支持将接口的网络掩码设为 0.0.0.0。此种条件下的 lwIP 协议栈行为与 Linux 行为有偏差。

  例如：有一个 eth0 接口和 loopback 接口。如果运行如下命令：

ifconfig eth0 netmask 0.0.0.0

ifconfig 成功将 eth0 的网络掩码设为具体数值，但在路由中会产生干扰结果，甚至会导致环回 ping 失败，显示目的地不可达。

- setsockopt()中的 SO_ATTACH_FILTER 选项有如下限制：

  因为不支持 LSF_MSH，所以不支持加载 IP 地址首部长度的 LSF_LDX hack。因此不支持下列过滤模式：

LSF_LDX+LSF_B+LSF_MSH X <- 4*(P[k:1]&0xf)*

- 当套接字绑定到一个 IP 地址，而这个 IP 地址配置了使用 bind()的接口，则该 IP 地址被更改。如果一个应用调用 listen()，会返回并显示 error ECONNABORTED。绑定在旧 IP 地址的套接字不会更新配置在接口的新 IP 地址，但状态更改为 ERR_ABRT。所以调用 listen()会返回并显示 error ECONNABORTED。

- 在 IPv6 原始套接字中，不支持 IPv4 的映射 IPv6[双栈支持]，即不支持 IPv6 原始套接字接收 IPv4 数据包或包含 IPv4 映射的 IPv6 地址的数据包。
- ping IPv4 地址有如下限制：

  没有 ICMP 标识符或 ICMP 序列号的匹配来维持 ping 会话，也没有来自对端的各自 ping 响应排序。

4 网络安全

4.1 网络安全声明

4.1 网络安全声明

为避免敏感信息泄露，用户须关注以下安全注意事项：

调试日志：通过配置 opt.h 头文件中的宏 LWIP_DBG_TYPES_ON，可启用或禁用 DebugFlag。该宏默认被禁用。

表4-1 调试关闭选项

选择码	建议值	影响
LWIP_DEBUGF	LWIP_DBG_OFF	当 LWIP_DBG_ON 被启用时，如果该值设为 1，则部分用户敏感信息可能被泄露。

注意阅读本声明，防止安全相关风险的发生。用户须关注以下安全注意事项：

- 用户在 lwipopts.h 头文件中定义 LWIP_RAND 函数宏。DHCP 和 DNS 会使用对应的函数随机生成业务的 ID。用户应该将 LWIP_RAND 宏定义为合适的随机数发生器函数。
- 基于 UDP 或 TCP 向 lwIP 发送的应用数据将先暂时保存在 lwIP 缓冲区中，在释放内存之前，缓冲区内的数据并不会被明确地清理掉。上述内容是基于这样一个假设，即应用不会将用户敏感信息以明文形式提供给 lwIP。这是因为应用通常会通过传输安全协议（如 TLS 或 DTLS）加密用户敏感信息，并将加密消息发送给 lwIP。

5 常见问题

1. lwIP 支持 route 命令吗?

答: lwIP 不支持 route 命令。

2. lwIP 的路由机制是什么?

答: lwIP 不支持路由选择，但如果启用了 IP_FORWARD 标记，则 lwIP 支持转发。网口报文是通过路由发送。

3. 未添加 netif 时，lwIP 是否可能正常运行?

答: 不可能。必须添加 Netif，并执行回调。

4. lwIP 支持 IPv4 组播?

答: 是的。通过设置 LWIP_IGMP 为 1，并使用 setsockopt()设置 IP_ADD_MEMBERSHIP 和 IP_DROP_MEMBERSHIP 等，可以启用该功能。

5. IP 地址变化对 TCP 连接有什么影响?

答: 现有的 TCP 连接都将被丢弃，并且任何 TCP 连接操作都将返回 ECONNABORTED。

所有绑定的地址都修改为新的 IP 地址。

如果有一个监听套接字，它将接受新 IP 地址上的连接。

6. IP 地址变化对 UDP 连接有什么影响?

答: 所有套接字的 IP 地址被修改为新的 IP 地址。在新的 IP 地址上继续通信。

7. 按照 RFC 2710 第 5 节的要求，协议栈是否支持发送包括被请求节点组播地址在内的 MLD 消息?

答: 按照 RFC 2710 第 5 节的要求，在主机部分工作时，MLD 消息不应发送给链路范围内全部节点组播地址，例如 FF02::1。

按照 RFC 2710 第 4 节的要求:

- 如果某个节点在通过接口接收到另一个节点的组播地址 REPORT 消息时，其计时器正在运行，那么该节点会停止计时器且不再发送组播地址 REPORT 消息，从而抑制在链路上复制报告。
- 当发送 DONE 消息时，如果该节点的最近 REPORT 消息因监听其它 REPORT 消息而被抑制，那么它可能停止发送消息，因为很有可能在同一链路上存在一个监听该地址的设备。这种机制可以被禁用，但默认是被开启的。

8. 如果邻居缓存数据存取状态不是 INCOMPLETE 时接收到目标地址为组播地址的未被请求的 NA 消息，会发生什么?

答: 状态变为 STALE。

9. 协议栈接收到路由器发送的 RA 消息时会发生什么?

答: 当协议栈接收到路由器发送的 RA 消息时，会向默认路由地址发送 NS 来及时解析默认的路由地址。

10. 可添加多少 IPv6 地址?

答: 最多 3 个，包括链路本地地址。如果添加的 IPv6 地址超过两个，那么第一个地址将被新地址替代，小型化会进一步优化。

11. 重复地址检测失败会发生什么?

答: 会注册回调函数来执行重复地址检测。重复地址检测失败时，可调用该回调函数来删除原有的 IPv6 地址，并添加新的地址，小型化已经不支持此特性。

12. lwIP 在配有多个接口的设备上运行时能否通过这些接口接收对端的消息?

答: 当 lwIP 在配有多个接口的设备上运行时，如果通过这些接口从对端接收的消息中的目标地址为广播 IP 地址，那么 lwIP 可以接收该消息。一旦调用 recv()函数，配置多少次接口，应用就可接收多少消息。

举例如下：

步骤 1 设备有两个接口，假设 IP 地址按如下方式配置：

Peer_Node 设备（lwIP）

192.168.23.119 （p5p2） <--> 192.168.23.117 （eth15）

192.168.2.119 （eth2） <--> 192.168.2.117 （eth1）

步骤 2 lwIP 套接字设置有 SO_BROADCAST 套接字选项。

步骤 3 将 lwIP 套接字与 INADDR_ANY 绑定。

步骤 4 将从 Peer_Node 接收的 DATA 消息发送至目标地址为 192.168.2.255 的设备。

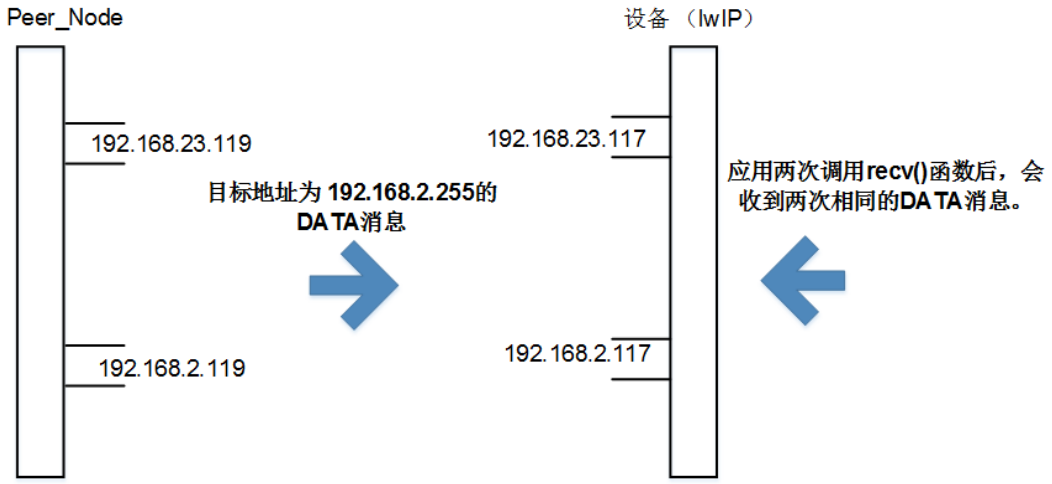
结果：

设备通过 192.168.23.117 和 192.168.2.117 两个接口接收 Peer_Node 的消息。应用两次调用 recv()函数后，会接收两次同样的消息。

说明

当 192.168.23.117 接口接收 Peer_Node 消息时，发现该消息并不是发送给该接口。这时，该接口会将该消息转发给 192.168.2.117 接口。

图5-1 接口接收对端的消息



----结束

20. 定制同时支持修改 TCP 或 UDP 的最大连接个数?

答： 修改 lwipopts.h 中的宏定义，会覆盖 opt.h 中宏的配置，如表 5-1 所示。

表5-1 lwipopts.h 的宏定义

宏	说明
MEMP_NUM_TCP_PCB	设置同一时间内所需的 TCP 连接数，目前设置为 4。
MEMP_NUM_UDP_PCB	设置同一时间内所需 UDP 连接数。设置该宏的值时，用户须考虑 lwIP 内部模块（例如：DNS 模块，DHCP 模块、LIBCOAP 模块），因此设置为 7+LWIP_MPL。
LWIP_NUM_SOCKETS_MAX	设置同一时间支持最大的 socket 个数为 9 个，包含：4 个 UDP socket、4 个 TCP socket、1 个 RAW socket。
MEMP_NUM_NETCONN	设置 TCP、UDP 和 RAW 连接总数。该宏的值必须是 MEMP_NUM_TCP_PCB、MEMP_NUM_UDP_PCB 和 MEMP_NUM_RAW_PCB 宏的值之和，目前设置为 DEFAULT_LWIP_NUM_SOCKETS=9。
MEMP_NUM_TCP_PCB_LISTEN	设置同时监听所需 TCP 连接数，目前设置为 2。

须知

- LWIP_NUM_SOCKETS_MAX 和 DEFAULT_LWIP_NUM_SOCKETS、MEMP_NUM_NETCONN 的值须保持一致。
- 调整 TCP 或 UDP 的最大连接个数需要考虑 LWIP_NUM_SOCKETS_MAX、DEFAULT_LWIP_NUM_SOCKETS、MEMP_NUM_NETCONN 的总数量。
- 对于 TCP Server 端，accept 建立连接需要 1 个 socket。
- 如果增加 TCP 或 UDP 的个数，会增加相应的 RAM 和 ROM 资源，请评估测试后使用。
- 默认场景下，支持的 socket 资源最大为 9 个，超过 9 个之后会创建 socket 失败。

19. 调整 lwIP 接收窗口相关配置?

答：通过减少 lwIP 接收窗口，降低内存占用，但是一定程度上会降低跑流峰值性能，建议合入修改后测试，确保满足实际应用场景。

说明

跑流场景下最多可减少内存占用约 8KB。

- 修改 1

Components/lwip_sack/include/lwip/lwipopts.h

```
#define TCP_SND_BUF        (65535 / 3)
```

修改为:

#define TCP_SND_BUF (12*1024)

- 修改 2

#define TCP_WND ((TCP_SND_BUF * 2) / 3)

修改为:

#define TCP_WND (8*1024)

- 修改 3

#define TCP_OOSEQ_MAX_PBUFS 8

修改为:

#define TCP_OOSEQ_MAX_PBUFS 5

18. LwIP 的路由机制是什么?

答: LwIP 路由机制是基于子网匹配，从 netif 列表查找合适的网络接口来发送报文。这里未采用最长前缀匹配算法，而是优先选择第一个与出报文目标 IP 匹配，即在同个子网的网络接口。如果未找到匹配，则使用默认的网络接口发送报文。如果通过 setsockopt()启用 SO_BINDTODEVICE，则只能通过绑定的网络接口发送报文；如果通过 setsockopt()启用 SO_DONTROUTE，则会使用同样匹配原则查找网络接口，但是在未找到匹配时，不会使用默认的网络接口。

17. 双接口共存的时候，发送数据包不成功处理方式?

答: 当前针对路由选择需要指定相应的接口，应用层可以使用下面的方式，比如需要往 ap 发包指定 AP0，需要往 wlan 的时候， 需要指定 wlan0。

ret = lwip_setsockopt(sfd, SOL_SOCKET, SO_BINDTODEVICE, "wlan0", IFNAMSIZ);

```
if (ret == -1) {
hi_at_printf("setsockopt: unknown iface %s\n", src_iface);
return -1;
}
```

16. 当系统内存不足，修改 tcp 减少丢包情况下接收端乱序队列个数?

答：当前 tcp 的实现为了高吞吐量，将 tcp 的接收端乱序队列设置为 8，当环境干扰大的时候，接收端最多缓存 8 个 tcp 包，由于 liteos 实现每个包最多占用 1748 字节，因此对占用内存，如果某些业务场景下不需要高带宽，需要设置 lwipopts.h 里面这个宏配置：TCP_QUEUE_OOSEQ。

15. AT 命令的 ifconfig 部分参数处理的行为?

答：lwip 的设计只允许 gw 和 ip 在相同子网内，lwip ifconfig 命令允许同时配置 ip、netmask、gw。当新配的 ip 是与旧 ip 不在同一子网的合法值，但 gw 非法时，新 ip 配置成功但原有的 gw 被清零。

14. 已关闭的 TCP 连接，用 netstat 命令查看一直处于状态 10，为什么?

答：在打开低功耗模式的情况下，对 TIME_WAIT pcb 采取延迟释放策略，即在下一次分配 tcp pcb 时如果没有空闲 pcb 才释放。因此，netstat 显示的状态 10（TIME_WAIT）的连接并不影响建立新连接的建立。如果希望关闭低功耗模式，可将编译选项 opt.h 中的 LWIP_LOWPOWER 宏改为 0。

如果关闭了低功耗模式，TIME_WAIT 连接也会等待一段时间再释放，等待时间可通过 tcp_priv.h 中的 TCP_MSL 宏设置。

13. 如何调整 TCP 建链的 SYN 报文的重传次数和重传间隔?

答：可以修改 TCP_SYNMAXRTX（SYN 报文最大重传次数），以及全局变量 tcp_persist_backoff（决定间隔周期数），注意 tcp_persist_backoff 的元素个数必须至少是 TCP_SYNMAXRTX+1。例如：进行以下修改，可以大幅缩短 connect 接口在对端不回复 SYN+ACK 的场景下的超时等待时间。

#define TCP_SYNMAXRTX 3

```
static const u8_t tcp_backoff[4] = { 1, 2, 4, 8 };
```

A 术语

缩写	术语	描述
ARP	地址解析协议	地址解析协议是一种将 IP 地址转换为物理地址的网络协议。
DHCP	动态主机配置协议	动态主机配置协议是 IP 网络上使用的一种标准化组网协议，用于为接口和业务动态分配 IP 地址等网络配置参数。
lwIP	轻量级 TCP/IP 协议栈	lwIP 是一款广泛应用于嵌入式系统的轻量级开源 TCP/IP 协议栈。
LiteOS	LiteOS	LiteOS 是华为公司基于 CMSIS 软件标准开发的一种操作系统。
ICMP	互联网控制消息协议	互联网控制消息协议是路由器等网络设备发送错误消息使用的协议，用以说明请求的服务不可用或无法到达主机。
IP	互联网协议	TCP/IP 协议簇中的一种协议，控制将分割的数据报文封装入包中、从发送站到目的网络和站点的包的路由选择以及在目标站组合成原始数据信息。IP 协议运行在 TCP/IP 模型的互联网层，对应于 ISO/OSI 模型的网络层。
IoT	物联网	是互联网、传统电信网等信息承载体，让所有能行使独立功能的普通物体实现互联互通的网络。
TCP	传输控制协议	TCP/IP 中的协议，用于将数据信息分解成信息包，使之经过 IP 协议发送；并对利用 IP 协议接收来的

缩写	术语	描述
		信息包进行校验并将其重新装配成完整的信息。TCP 是面向连接的可靠协议，能够确保信息的无误发送，它与 ISO/OSI 基准模型中的传输层相对应。
UDP	用户数据报协议	TCP/IP 标准协议，允许一个设备上的应用程序向另一个设备上的应用程序发送数据报。UDP 使用 IP 传送数据报，为应用程序提供不可靠的无连接报文传送服务。即 UDP 消息可能丢失、重复、延迟或乱序。目的设备不主动确认是否收到正确的数据包。