



Hi3861 V100 / Hi3861L V100 lwIP

Development Guide

Issue	01
Date	2020-04-30

Copyright © HiSilicon (Shanghai) Technologies Co., Ltd. 2020. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of HiSilicon (Shanghai) Technologies Co., Ltd.

Trademarks and Permissions



HISILICON, and other HiSilicon icons are trademarks of HiSilicon Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between HiSilicon and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided “AS IS” without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

HiSilicon (Shanghai) Technologies Co., Ltd.

Address: New R&D Center, 49 Wuhe Road,
Bantian, Longgang District,
Shenzhen 518129 P. R. China

Website: <https://www.hisilicon.com/en/>

Email: support@hisilicon.com



About This Document

Purpose

This document introduces the technology, application development, network security, and FAQs of the lightweight TCP/IP stack (lwIP), providing guidance for adapting lwIP to Huawei LiteOS and implementing additional features such as the DNS client and DHCP server.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3861	V100
Hi3861L	V100



Intended Audience

This document is intended for:



- Test engineers
- Software development engineers

Symbol Conventions

The following table describes the symbols that may be found in this document.

Symbol	Description
	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.



Symbol	Description
 CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
01	2020-04-30	<p>This issue is the first official release.</p> <ul style="list-style-type: none">In 1.3 RFC Compliance, the description is updated.In 2.1 Supported Features, the descriptions of the Simple Network Time Protocol (SNTP), TCP window scale option, and developed features are deleted.In 2.2 Unsupported Features, the description of the lwIP miniaturization feature is added.In 3.4.2 Adding Netif and Driver Functions, the description of configuring IP addresses for interfaces is deleted.In 3.5.4 lwIP Macros, the description of the lwIP macro change is updated. The reference link to the lwIP 2.0 configuration macro is added. In Table 3-1, the descriptions of SNTP_MAX_REQUEST_RETRANSMIT, DNS_MAX_LABEL_LENGTH, LWIP_BSD_API, LWIP_WND_SCALE, and TCP_WND_MIN are deleted.In 3.7 Restrictions, the description of the restrictions on further tailoring the RAM and ROM is updated. The description of the <code>writenv()</code> interface is deleted. The restrictions on the <code>lwip_get_conn_info()</code> interface are deleted. The description of files that lwIP TFTP does not support is deleted.In 11. What will happen if duplicate address detection fails? of 5 FAQs, the description that this feature is not supported in the miniaturization scenario is added.



Issue	Date	Change Description
00B0 2	2020-01- 15	In 5 FAQs , item 13 and item 14 are added.
00B0 1	2019-11- 15	This issue is the first draft release.



Contents

About This Document.....	i
1 Introduction.....	1
1.1 Background.....	1
1.2 Third-Party Reference.....	1
1.3 RFC Compliance.....	1
2 Features.....	3
2.1 Supported Features.....	3
2.2 Unsupported Features.....	6
3 Development Guidance.....	8
3.1 Prerequisites.....	8
3.2 Dependencies.....	8
3.3 lwIP Usage.....	8
3.4 Porting Method.....	9
3.4.1 Initializing lwIP.....	9
3.4.2 Adding Netif and Driver Functions.....	9
3.5 lwIP Optimization.....	10
3.5.1 Throughput Optimization.....	11
3.5.2 Memory Optimization.....	11
3.5.3 Customization.....	12
3.5.4 lwIP Macros.....	12
3.6 Sample Code.....	18
3.6.1 Sample Code for Applications.....	18
3.6.1.1 Sample Code of UDP.....	18
3.6.1.2 Sample Code of TCP Client.....	20
3.6.1.3 Sample Code of TCP Server.....	21
3.6.1.4 Sample Code of DNS.....	23
3.6.2 Sample Code for Drivers.....	25
3.6.3 Sample Code for Services.....	26
3.6.3.1 Sample Code of SNTP.....	26
3.6.3.2 Sample Code of DHCP Client.....	26
3.6.3.3 Sample Code of DHCP Server.....	28
3.7 Restrictions.....	29



4 Cyber Security	34
4.1 Cyber Security Statement	34
5 FAQs	35
6 Terms	39



1 Introduction

[1.1 Background](#)

[1.2 Third-Party Reference](#)

[1.3 RFC Compliance](#)

1.1 Background

lwIP is developed to migrate the smart home chip to the lightweight operating system (OS) and TCP/IP protocol stack. It has the following features:

- Huawei Lite OS is a lightweight real-time operating system (RTOS) designed for the Cortex-M series, Cortex-R series, and Cortex-A series chips. It is customized for smart terminals and wearable devices and has mature commercial applications.
- The DHCP server and other additional features are implemented based on the open-source lwIP.

1.2 Third-Party Reference

This document provides the following reference for third-party applications:

- SNTP client: This lwIP does not re-implement the Simple Network Time Protocol (SNTP) client. However, the SNTP client has been implemented in the open-source lwIP contrib.
- OS adaptation-layer code: The lwIP contrib provides the code that adapts to the Linux platform. Huawei LiteOS supports most Portable Operating System Interfaces (POSIX), requiring the code during integration.

1.3 RFC Compliance

NOTE

According to the industry protocols, the chip can interact with other chips normally. There is no privatized protocol. Only a few of these protocols are not fully complied with.



lwIP complies with the following Request For Comments (RFC) standards:

- 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 Time Protocol (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 Features

[2.1 Supported Features](#)

[2.2 Unsupported Features](#)

2.1 Supported Features

lwIP supports the following features:

- Internet Protocol version 4 (IPv4)
IPv4 is a connectionless protocol used in packet switched network. IPv4 depends on the best-effort delivery approach, which means it does not guarantee that no packet is lost or that all packets arrive in the correct order without repetition.
- Internet Control Message Protocol (ICMP)
It is one of the core protocols in the Internet protocol suite. This protocol is used to send notification messages such as "Host Unreachable" and ping messages such as "Echo Request" and "Echo Reply".
- User Datagram Protocol (UDP)
It is one of the core protocols in the Internet protocol suite. This protocol uses a simple connectionless transmission model and a minimum protocol mechanism, without a handshake. Therefore, this protocol exposes the unreliability of the underlying network protocol to the user program, and cannot ensure that no packet loss, order, or repetition occurs. UDP provides checksums for data integrity and port numbers for addressing the source and destination of data packets.
- Transmission Control Protocol (TCP)
It is one of the core protocols in the Internet protocol suite. This protocol originates from the original network implementation and supplements IP. Therefore, the entire protocol suite is called TCP/IP. TCP provides reliable, ordered, and error-correcting transmission for host applications on IP networks. Applications that do not require reliable data flow services can use UDP, which provides connectionless datagram services with low latency and low reliability.
- Domain Name System (DNS) client

The DNS is a hierarchical distributed naming system for computers, services, or resources connected to the Internet or private networks. lwIP supports the DNS client (supporting domain name resolution) that is based on the RFC 1035 protocol and has few features.

In Huawei LiteOS lwIP, the DNS uses a configurable rollback mechanism between A records and AAAA records.

The rollback mechanism means that if the host name fails to be resolved to the A record, the DNS automatically attempts to resolve the host name to the AAAA record, or vice versa, without notifying the application that the host name fails to be resolved to the A record.

The DNS response depends on the **LWIP_DNS_ADDRTYPE_DEFAULT** parameter. The values of this parameter are as follows:

- **LWIP_DNS_ADDRTYPE_IPV4**
The DNS responds only to the A record of the host name requested by the client.
- **LWIP_DNS_ADDRTYPE_IPV6**
The DNS responds only to the AAAA record of the host name requested by the client.
- **LWIP_DNS_ADDRTYPE_IPV4_IPV6**
The A record is parsed first. If the parsing fails, the AAAA record is used.
- **LWIP_DNS_ADDRTYPE_IPV6_IPV4**
The AAAA record is parsed first. If the parsing fails, the A record is used.

Huawei LiteOS lwIP can obtain multiple resolved IP addresses for a single host name.

- The **count** parameter specifies the number of IP addresses to be resolved by the client.
- The minimum value of **count** must be **1**. If the value is less than 1, the DNS resolution fails and **ERR_ARG** is returned.
- The maximum value of **count** is equal to the value of **DNS_MAX_IPADDR**.
If the value of **count** is greater than the value of **DNS_MAX_IPADDR**, the IP address with the maximum resolution of the host name **DNS_MAX_IPADDR** is returned only to the application.

The DNS interface can be called in blocking or non-blocking mode.

NOTE

The DNS on lwIP 2.0 does not verify or change the resolved IP address provided by the DNS server. The DNS only checks whether the resolved IP address is correct. Therefore, the resolved IP address is returned directly to the application.

- **Dynamic Host Configuration Protocol (DHCP) client and server**
It is a standard network protocol used on IP networks to dynamically allocate network configuration parameters, such as interface IP addresses and service IP addresses. Through DHCP, a computer automatically requests an IP address and network parameters from the DHCP server, which reduces the workload of network administrators or users. lwIP supports the DHCP client and server based on the RFC 2131 protocol.
 - If the default DHCPv4 address range (address pool) is used, **start_ip** and **ip_num** must be **NULL**.



- If the DHCPv4 address range (address pool) needs to be manually configured, **start_ip** and **ip_num** cannot be **NULL**.
- Whether the default address range or the manually configured address range is used depends on whether the IP address of the server is within the address pool. If the IP address of the server is within the address pool, one less IP address is required. If the IP address is not in the address pool, the number of IP addresses is not one less.
Note: The default address pool starts from **2**.
- If the IP address of the DHCPv4 server is in the range of (ip_start, ip_start + ip_num), the total number of available addresses in the client pool is smaller than **ip_num** because one address in the pool is occupied by the DHCPv4 server.
- The total number of addresses in the DHCPv4 address range is the smallest value between **ip_num** and **LWIP_DHCP_MAX_LEASE**.
- When the DHCPv4 address range is manually configured, if the sum of **ip_start** and **ip_num** is greater than **x.y.z.254**, the total number of addresses in the address pool must start from (ip_start, x.y.z.254). If the subnet mask is **255.255.0.0** or **255.255.255.128**, the preceding conditions are not applicable.
- The DNS server option is returned by the DHCP server to the client. The DHCP Discover and Request packets do not carry this option. In lwIP 2.0, the DNS server address is as follows:
 - If an IPv4 DNS server is configured, the configured DNS address is returned (two DNS address may be returned).
 - If no DNS address is configured, the interface IP address of the DHCP server is returned.
- The DHCPv4 server responds only to the address of the primary DNS server. If two DNS server addresses are configured in the system, the system returns the addresses of both DNS servers.
- Deviation from RFC 2131
According to section 4.3.2 in RFC 2131, the DHCPv4 server checks whether the client is filled with **server_id** in the INIT-REBOOT, RENEWING, or REBINDING state of the DHCP Request packet. If the client is filled with **server_id**, the server will not accept the packet. However, in lwIP 2.0, the DHCPv4 server does not consider the INIT-REBOOT, RENEWING, or REBINDING state of the **server_id** packet.
- Ethernet Address Resolution Protocol (ARP)
ARP is a telecommunication protocol used to resolve the network layer address into the link layer address. It is an important function in the multi-access network. Huawei LiteOS lwIP based on RFC 826 supports the ARP protocol and configurable ARP table.
- Internet Group Management Protocol (IGMP)
IGMP is used to transmit information about multicast group members between hosts and local routers. Currently, lwIP supports only the IGMP v2 protocol based on RFC 2236.
- Socket API types
lwIP provides the following types of socket APIs:



- Non-thread-safe low-level APIs
- Thread-safe Netconn APIs
- Netconn APIs in the Berkeley Software Distribution (BSD) style are called internally. BSD APIs are provided to help applications smoothly migrate from the Linux TCP/IP stack to the lwIP TCP/IP stack.
- TCP selective acknowledgment option
- This option is used to acknowledge the missing sequence segments received by the stack so that these selective acknowledgment column segments can be skipped for retransmission during loss recovery.

2.2 Unsupported Features

lwIP does not support the following features or protocols:

- PPPoS/PPPoE
- SNMP agent (Currently, lwIP supports only private MIBs.)
- IP forwarding through multiple network interfaces
- Routing function (only for terminals)

lwIP miniaturization has the following features:

- Simple Network Time Protocol (SNTP)
- SNTP is a clock synchronization network protocol between computer systems based on packet switching and variable-delay data networks. lwIP supports SNTP version 4 based on RFC 2030.
- The **lwip_get_conn_info()** API has the following restrictions:
 - This function is used to obtain TCP or UDP connection information.
 - The null pointer is of **tcpip_conn** type.
 - In the LISTEN state, this API cannot be used to obtain TCP socket connection information.
- TCP window scale option
- lwIP supports the TCP window scale option based on the RFC 7323 protocol.
 - The TCP window scale option allows the use of a 30-bit window size instead of a 16-bit window size in a TCP connection.
 - The window scale extension expands the definition of the TCP window to 30 bits and then uses a scale factor to carry this 30-bit value in the 16-bit Window field of the TCP header.
 - The exponent of the scale factor is carried in a TCP option, Window Scale.
- lwIP supports the AutoIP module.
- lwIP supports the API for setting the Wi-Fi driver status.
- lwIP supports the API for setting the Wi-Fi driver status to the lwIP stack.
 - When the Wi-Fi driver is busy, the lwIP stack stops sending messages.
 - When the Wi-Fi driver is ready, the lwIP stack continues to send messages.

If the driver does not wake up from the busy state before **DRIVER _ WAKEUP _ INTERVAL** times out, all TCP connections using the netif driver will be



deleted. Therefore, blocking connection calls will wait for the netif driver to wake up, retransmit synchronize sequence numbers (SYN), or wait for TCP connection timeout to be cleared.

- lwIP provides the **PF_PACKET** option on the SOCK_RAW.

lwIP supports the SOCK_RAW of the **PF_PACKET** family. Applications can use this feature to create link-layer sockets. Therefore, the lwIP stack expects the application to include the link layer header when sending a packet. SOCK_RAW packets are transmitted between device drivers, and the packet data remains unchanged.

By default, all packets of a specified protocol type are delivered to a packet socket. To get packets only from a specific interface, bind the socket to the specified interface. The **sll_protocol** and **sll_ifindex** fields are used for binding. When an application sends packets, you need to specify only **sll_family**, **sll_addr**, **sll_halen**, and **sll_ifindex**. Other fields should be set to **0**. The received packets are set to **sll_hatype** and **sll_pkttype**.



3 Development Guidance

lwIP supports Ethernet and Wi-Fi. The lwIP and driver must be configured and adapted as required.

[3.1 Prerequisites](#)

[3.2 Dependencies](#)

[3.3 lwIP Usage](#)

[3.4 Porting Method](#)

[3.5 lwIP Optimization](#)

[3.6 Sample Code](#)

[3.7 Restrictions](#)

3.1 Prerequisites

lwIP must meet the following prerequisites:

- lwIP has been optimized to support Huawei LiteOS. Currently, lwIP does not support other operating systems. For details, see [3.5 lwIP Optimization](#).
- The driver code has been updated based on the lwIP configurations before lwIP is used. For details, see the pseudocode in [3.4 Porting Method](#).

3.2 Dependencies

The lwIP dependencies are as follows:

- Huawei LiteOS
- Ethernet or Wi-Fi driver module for data transmission and reception at the physical layer

3.3 lwIP Usage

lwIP provides the BSD TCP/IP socket APIs for applications to connect to the server. lwIP has the following advantages:



- The old application code running on the BSD TCP/IP protocol stack can be directly ported to lwIP.
- lwIP supports the DHCP client feature and can be used to configure dynamic IP addresses.
- lwIP supports the DNS client feature, which can be used by applications to resolve domain names.
- lwIP occupies few resources, which can meet the protocol stack requirements.

3.4 Porting Method

3.4.1 Initializing lwIP

lwIP is initialized by using **tcpip_init()**. This API receives an optional callback function and its parameter (the parameter value can be set to **NULL**). Once the initialization is complete, the callback function is called and the corresponding parameter is passed into it.

Sample code:

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

3.4.2 Adding Netif and Driver Functions

After the initialization is complete, at least one netif interface must be added to the application for communication.

The application needs to implement the callback function based on the driver type. The link layer type, MAC address, and TX callback function must be registered before they are used. To support the promiscuous mode of the raw socket interface, the callback function for implementing this functionality needs to be registered in the driver.

Sample code of Ethernet:

```
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_rcv 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_rcv(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_rcv : pbuf_alloc failed\n");
    return;
}
#ifdef ETH_PAD_SIZE
    pbuf_header(p, -ETH_PAD_SIZE); /* drop the padding word */
#endif
    memcpy(p->payload, data, len);
#ifdef 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 */
#ifdef 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);
#ifdef LWIP_NETIF_PROMISC
    g_netif.drv_config = eth_drv_config;
#endif
#ifdef 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();
#ifdef LWIP_WITH_DHCP_CLIENT
    netifapi_dhcp_start(&g_netif);
    do {
        msleep(20);
    } while (netifapi_dhcp_is_bound(&g_netif) != ERR_OK);
#endif
    netifapi_netif_set_up(&g_netif);
    printf("Network is up !!!\n");
}
```

3.5 lwIP Optimization

3.5.1 Throughput Optimization

To optimize the lwIP throughput in different application scenarios, you are advised to perform the following operations:

- Set the value of the macro **MEMP_NUM_UDP_PCB**, that is, the number of required UDP connections.
DHCP also creates a UDP connection, which needs to be considered when you set the value of this macro.
- Set the value of the macro **MEMP_NUM_TCP_PCB**, that is, the number of required TCP connections.
- Set the value of the macro **MEMP_NUM_RAW_PCB**, that is, the number of required RAW connections.
The LWIP_ENABLE_LOS_SHELL_CMD module uses a raw connection to execute the **ping** command.
- Set the value of the macro **MEMP_NUM_NETCONN**, that is, the total number of required UDP, TCP, and RAW connections.
 - If neither LWIP_ENABLE_LOS_SHELL_CMD nor RAW connection is used, disable LWIP_RAW.
 - If UDP is not in use, disable LWIP_UDP. If TCP is not used, disable LWIP_TCP.
 - If the memory of type **PBUF_RAM** fails to be allocated by calling **pbuf_alloc()** in the receive function of the driver, increase the value of **MEM_SIZE**.
 - If **driverif_input()** fails to be called because the packet space in the TCPIP mbox is unavailable, increase the values of **TCPIP_MBOX_SIZE** and **MEMP_NUM_TCPIP_MSG_INPKT**. If the driver module receives packets too fast, the packet space may be unavailable. As a result, **driverif_input** fails to be called.
- Disable all debugging options and do not define **LWIP_DEBUG**.
- If the architecture word length is 4, set **ETH_PAD_SIZE** to 2.

3.5.2 Memory Optimization

To save the lwIP memory space, you are advised to perform the following operations:

- Call **pbuf_alloc()** in the receive function of the driver to allocate the memory of type **PBUF_POOL**, set the value of **PBUF_POOL_SIZE** (the number of required PBUFs), and set the value of **PBUF_POOL_BUFSIZE** based on the maximum transmission unit (MTU).
- Set the values of **MEMP_NUM_TCP_PCB**, **MEMP_NUM_UDP_PCB**, **MEMP_NUM_RAW_PCB**, and **MEMP_NUM_NETCONN** based on the required TCP, UDP, and RAW data and the total number of connections.
- Reduce the values of **TCPIP_MBOX_SIZE** and **MEMP_NUM_TCPIP_MSG_INPKT** based on the actual amount of data sent by the peer end.
- Disable all debugging options and do not define **LWIP_DEBUG**.
Change to **#define LWIP_DBG_TYPES_ON LWIP_DBG_OFF** in **lwipopts.h**.



- Disable **ETHARP_TRUST_IP_MAC**.
 - If this macro is enabled, all received packets are used to update the ARP table.
 - If this macro is disabled, the entries in the ARP table are updated only when an ARP query is performed.

3.5.3 Customization

In actual applications, some functions may need to be customized. The usage of the macros newly defined in lwIP is described as follows:

- **LWIP_DHCP**
To enable the DHCP server, you need to enable the **LWIP_DHCPS** macro. If the **LWIP_DHCPS** macro is enabled, the **LWIP_DHCP** macro must also be enabled.
- **LWIP_DHCPS_DISCOVER_BROADCAST**
Generally, the DHCP server broadcasts or unicasts an Offer message based on the flag set by the client in the Discover message. If you want to keep broadcasting Offer messages, you can enable the **LWIP_DHCPS_DISCOVER_BROADCAST** macro.

3.5.4 lwIP Macros

NOTE

The lwIP 2.0 configuration macros are not fully described here because most of the code has been open-source. For details, visit https://www.nongnu.org/lwip/2_0_x/index.html. If you need to modify the configuration macros, contact the technical support.

The macros are described as follows (if you need to change the description, contact the technical support).

- The default values of the lwIP macros are defined in the **opt.h** and **lwipopts.h** header files.
- The macro definition in the **lwipopts.h** header file overwrites that in the **opt.h** header file.

Table 3-1 Macro list

Macro	Description
LWIP_AUTOIP	This macro is used to enable or disable the AUTOIP module.
MEM_SIZE	lwIP maintains the heap memory (including mem_malloc and mem_free) management module, which is used for dynamic memory allocation. This macro is used to define the size of the heap memory management module in lwIP.



Macro	Description
MEM_LIBC_MALLOC	If this macro is enabled, the system calls malloc() and free() to allocate all dynamic memory, and the heap memory management module code in lwIP will be disabled.
MEMP_MEM_MALLOC	lwIP provides a pool memory (including memp_malloc and memp_free) management module for frequently used structures.
MEM_ALIGNMENT	The value of this macro needs to be set based on the architecture. For example, if the 32-bit architecture is used, you need to set this macro to 4 .
MEMP_NUM_TCP_PCB	This macro is used to set the number of TCP connections required at the same time.
MEMP_NUM_UDP_PCB	This macro is used to set the number of UDP connections required at the same time. When setting the value of this macro, you need to consider the internal modules of lwIP, such as the DNS module and DHCP module. The DHCP module creates a UDP connection for its own communication.
MEMP_NUM_RAW_PCB	This macro is used to set the number of RAW connections required at the same time.
MEMP_NUM_NETCONN	This macro is used to set the total number of TCP, UDP, and RAW connections. The value of this macro must be the sum of the values of MEMP_NUM_TCP_PCB , MEMP_NUM_UDP_PCB , and MEMP_NUM_RAW_PCB .
MEMP_NUM_TCP_PCB_LISTEN	This macro is used to set the number of TCP connections required for concurrent listening.
MEMP_NUM_REASSDATA	This macro is used to set the number of IP packets (complete packets instead of fragmented packets) waiting for reassembling in a queue.
MEMP_NUM_FRAG_PBUF	This macro is used to set the number of IP packets (fragmented packets instead of complete packets) that are sent at the same time.
ARP_TABLE_SIZE	This macro is used to set the size of the ARP cache table.
ARP_QUEUEING	If this macro is enabled, multiple egress packets are queued during ARP resolution. If this macro is disabled, only the latest packets sent by the upper layer can be reserved for each destination address.



Macro	Description
MEMP_NUM_ARP_QUEUE	This macro is used to set the number of egress packets (PBUF) that are queued at the same time. These egress packets are waiting for the ARP response to resolve the destination address. This macro is applicable only when ARP_QUEUEING is enabled.
ETHARP_TRUST_IP_MAC	If this macro is enabled, the source IP address and source MAC address in the ARP cache table are updated by all ingress packets. If this macro is disabled, the entries in the ARP cache table can only be updated through ARP queries.
ETH_PAD_SIZE	This macro is used to set the number of bytes added before the Ethernet header to ensure that the payload is aligned after the header. The length of the IP header is 14 bytes. If the IP header is not padded, the addresses in the IP header are not aligned. For example, in a 32-bit architecture, setting the value of this macro to 2 can align the length of the IP header to 4 bytes or accelerate packet transmission.
ETHARP_SUPPORT_STATIC_ENTRIES	This macro supports the static update of the ARP cache table through the etharp_add_static_entry() and etharp_remove_static_entry() APIs.
IP_REASSEMBLY	This macro supports the reassembly of IP fragmentation packets.
IP_FRAG	This macro can fragment all egress packets at the IP layer.
IP_REASS_MAXAGE	This macro is used to set the timeout period for fragmenting ingress IP packets. lwIP can reserve IP_REASS_MAXAGE seconds for fragmenting packets. If packets cannot be reassembled within this period, the fragmented packets are discarded.
IP_REASS_MAX_PBUFS	This macro is used to set the maximum number of fragments allowed by an ingress IP packet.
IP_FRAG_USES_STATIC_BUF	If the macro is enabled, the static buffer is used for IP fragmentation. Otherwise, the dynamic memory is allocated.
IP_FRAG_MAX_MTU	This macro is used to set the maximum value of the MTU.
IP_DEFAULT_TTL	This macro is used to set the default time to live (TTL) of transport layer data packets.



Macro	Description
LWIP_RANDOMIZE_INITIAL_LOCAL_PORTS	This macro supports randomization of the local port of the first local TCP/UDP program control block (PCB). The default value is 1 . This function prevents the creation of predictable port numbers after the device is started. lwIP depends on the random number generator function of the system. Therefore, the strong random number generator function must be called to set the value of the LWIP_RAND macro.
LWIP_RAND	lwIP depends on the random number generator function of the system. Therefore, the strong random number generator function must be called to set the value of this macro. Random numbers are used to generate random client ports for DNS and TCP/UDP connections. In addition, random numbers are used to create transaction IDs in DHCP and DNS and to create ISS values in TCP.
LWIP_ICMP	This macro is used to enable the ICMP module.
ICMP_TTL	This macro is used to set the TTL value of an ICMP message.
LWIP_RAW	This macro enables the raw socket support in lwIP.
RAW_TTL	This macro is used to set the TTL value of a raw socket message.
LWIP_UDP	This macro enables the UDP connection support in lwIP.
UDP_TTL	This macro is used to set the TTL value of a UDP socket message.
LWIP_TCP	This macro enables the TCP connection support in lwIP.
TCP_TTL	This macro is used to set the TTL value of a TCP socket message.
TCP_WND	This macro is used to set the TCP window size.
TCP_MAXRTX	This macro is used to set the maximum number of retransmissions of TCP data packets.
TCP_SYNMAXRTX	This macro is used to set the maximum number of retransmissions of TCP SYN packets.
TCP_FW1MAXRTX	This macro is used to set the maximum number of retransmissions of connection close packets (that is, packets enter the FIN_WAIT_1 or CLOSING state).
TCP_QUEUE_OOSEQ	This macro can buffer received out-of-order data packets. If the memory of the user equipment is low, set this macro to 0 .



Macro	Description
TCP_MSS	This macro is used to set the maximum segment size (MSS) of a TCP connection.
TCP_SND_BUF	This macro is used to set the size of the TCP TX buffer.
TCP_SND_QUEUELEN	This macro is used to set the length of the TCP TX queue.
TCP_OOSEQ_MAX_BYTES	This macro is used to set the maximum number of bytes in the queue on the OOSEQ of each PCB. The default value is 0 (no limit). This macro is applicable only when TCP_QUEUE_OOSEQ is set to 0 .
TCP_OOSEQ_MAX_PBUFS	This macro is used to set the maximum number of PBUFs in the queue on the OOSEQ of each PCB. The default value is 0 (no limit). This macro is applicable only when TCP_QUEUE_OOSEQ is set to 0 .
TCP_LISTEN_BACKLOG	This macro supports the backlog function when TCP listening is enabled.
LWIP_DHCP	This macro is used to enable the DHCP client module.
LWIP_DHCP_SERVER	This macro is used to enable the DHCP server module.
LWIP_IGMP	This macro is used to enable the IGMP module.
LWIP_SNTP	This macro is used to enable the SNTP client module.
LWIP_DNS	This macro is used to enable the DNS client module.
DNS_TABLE_SIZE	This macro is used to set the size of the DNS cache table.
DNS_MAX_NAME_LENGTH	This macro is used to set the maximum length of a domain name. According to the DNS RFC, the domain name length is set to 255 . You are advised not to change the value.
DNS_MAX_SERVERS	This macro is used to set the number of DNS servers.
DNS_MAX_IPADDR	This macro is used to set the maximum IP address that can be cached by the DNS client.
PBUF_LINK_HLEN	This macro is used to set the number of bytes that must be allocated to the link-level header, including the actual length and ETH_PAD_SIZE .
LWIP_NETIF_API	This macro is used to enable the thread-safe netif interface module (netapi_*).
TCPIP_THREAD_NAME	This macro is used to set the name of the TCP IP thread.



Macro	Description
DEFAULT_RAW_RECVMBOX_SIZE	Each RAW connection maintains a packet queue for buffering packets until the application layer sends a receiving notification. This macro is used to set the size of the message receiving box queue.
DEFAULT_UDP_RECVMBOX_SIZE	Each UDP connection maintains a packet queue for buffering packets until the application layer sends a receiving notification. This macro is used to set the size of the message receiving box queue.
DEFAULT_TCP_RECVMBOX_SIZE	Each TCP maintains a packet queue for buffering packets until the application layer sends a receiving notification. This macro is used to set the size of the message receiving box queue.
DEFAULT_ACCEPTMBOX_SIZE	This macro is used to set the size of the message box queue to maintain the TCP connection.
TCPIP_MBOX_SIZE	This macro is used to set the size of the message box queue of the TCP/IP thread. The queue can maintain all operation requests sent by application threads and driver threads.
LWIP_TCPIP_TIMEOUT	This macro supports the feature of enabling any user-defined timer processing function on the running lwIP TCP/IP thread.
LWIP_SOCKET_START_NUM	This macro is used to set the start number of the socket file descriptor created by lwIP.
LWIP_COMPAT_SOCKETS	This macro can create a Linux BSD macro for all socket interfaces.
LWIP_STATS	This macro is used to collect statistics on all online connections.
LWIP_SACK	This macro is used to enable or disable the SACK function in lwIP. To enable the SACK function of the TX and RX ends, you need to enable this macro.
LWIP_SACK_DATA_SEG_PIGGYBACK	This macro is used to send the SACK option in the data segment with the ACK flag set. If this macro is disabled, the SACK option will only be sent in empty ACK data segments and in ACKs when bidirectional data transfers occur, not data segments.
MEM_PBUF_RAM_SIZE_LIMIT	<p>This macro is used to determine whether to limit the size of the memory of type PBUF_RAM allocated by using pbuf_alloc(). This limits the allocation of the OS memory of type PBUF_RAM, not the allocation of the internal buffer.</p> <p>This macro is applicable only when MEM_LIBC_MALLOC is enabled.</p>



Macro	Description
LWIP_PBUF_STATS	When MEM_PBUF_RAM_SIZE_LIMIT is enabled, this macro can be used to enable or disable the function of printing the statistics on allocation of the memory of type PBUF_RAM .
PBUF_RAM_SIZE_MIN	The minimum RAM memory needs to be set by calling pbuf_ram_size_set .
DRIVER_STATUS_CHECK	This macro can be used to enable the driver to send buffer status notifications to the protocol stack through the netifapi_stop_queue and netifapi_wake_queue APIs.
DRIVER_WAKEUP_INTERRUPT	If the netif driver state changes to Busy and is not restored to Ready before the timer expires, all TCP connections to the netif driver are cleared. The default value is 120000 ms.
PBUF_LINK_CHKSUM_LEN	This macro provides the length of the link-layer checksum padded by the Ethernet driver.
LWIP_DEV_DEBUG	This macro applies only to debugging by developers and must be disabled in the user environment.

3.6 Sample Code

This section describes the lwIP sample code and its usage. The lwIP sample code contains pseudocode that describes the changes to be made to the Ethernet or Wi-Fi driver module. The sample code, lwIP, and Huawei LiteOS are compiled by the cross compiler of the HiSilicon platform at the same time.

NOTICE

The sample code cannot be directly put into commercial use.

3.6.1 Sample Code for Applications

3.6.1.1 Sample Code of UDP

```
#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 */
    /* rcv */
    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);
    /* rcv */
}
```



```
    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 Sample Code of TCP Client

```
#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 Sample Code of TCP Server

```
#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 Sample Code of DNS

```
#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_errno;
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_errno);
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;
}
```

3.6.2 Sample Code for Drivers

```
#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_rcv 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_rcv(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_rcv : 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 Sample Code for Services

3.6.3.1 Sample Code of SNTP

```
#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("Receved 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 Sample Code of DHCP Client

```
#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 Sample Code of DHCP Server

```
#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 Restrictions

Before using lwIP, consider the following restrictions:

- The OS adaptation layer in lwIP is tightly coupled with Huawei LiteOS interfaces. Therefore, lwIP can run only on Huawei LiteOS.
- In current lwIP, the RAM and ROM are further tailored and controlled by configuring the macro in **lwipopts.h**. For more restrictions, see the macro **LWIP_SMALL_SIZE** in the **lwipopts.h** code.
- The DHCP client allows the UDP socket interfaces to be bound to the DHCP client port, while lwIP does not allow multiple network ports (including the netif structures of Ethernet and Wi-Fi) to be bound to the same port. Therefore, if **SO_BINDTODEVICE** is disabled, lwIP can run with two network ports (Ethernet and Wi-Fi) at the same time, but the DHCP client cannot.
- The DNS client supports only A or AAAA resource records in response messages. When multiple response records in the DNS response message are parsed, if any abnormal response record is encountered, the parsing is terminated and a record that is successfully parsed (if a record that is successfully parsed exists) is returned; otherwise, a record that fails to be parsed is returned.
- lwIP provides the following types of interfaces:
 - BSD interfaces
Thread safe
 - Netconn interfaces
Thread safe
 - Low-level interfaces
Not thread safe. Therefore, these interfaces are not recommended. If these interfaces are used, the core TCP/IP thread must be locked for the application thread and driver thread.
- The restrictions on using multiple threads of lwIP are as follows:
 - The lwIP core is not thread-safe. If applications in a multi-thread environment need to use lwIP, the interface layer (such as the netconn or socket interface layer) must be used. If low-level interfaces are used, the lwIP core must be protected.
 - **netif_xxx** and **dhcp_xxx** are not thread-safe interfaces. Therefore, the application should use the thread-safe interfaces **netifapi_netif_xxx** and **netifapi_dhcp_xxx** available in the netifapi module.
The thread safety of lwIP interfaces is described as follows:
 - All socket BSD APIs are thread-safe.
 - All APIs such as **netifapi_xxx** are thread-safe.
 - All APIs such as **netif_xxx** are not thread-safe.
 - Do not refer to the **api_shell.c** file. Although many non-thread-safe APIs are used in this file, the task executed by the APIs is **tcpip_thread**. Therefore, do not simply copy the code.



- The number of network ports created by applications cannot exceed 254 because the port index required for network port maintenance ranges from 1 to 254.
- Multi-thread consideration for the **select()** and **close()** interfaces:
If a socket is monitored by the **select()** interface in one thread but closed by the **close()** interface in another thread, the **select()** interface in lwIP returns from the locked state without marking the specific socket. In Linux, closing a socket in another thread has no effect on the **select()** interface.
- The **shutdown()** interface has the following restrictions:
 - If the **shutdown()** interface is called by using the **SHUT_RDWR** or **SHUT_RD** flag, any data to be received should be cleared by lwIP, the RST message is sent to the peer end, and the application must read data before calling the **SHUT_RDWR** or **SHUT_RD** flag.
 - When the transmission is blocked and **shutdown(SHUT_RDWR)** is called, **EINPROGRESS (115)** is returned.
- The **listen()** interface has the following restrictions:
 - The maximum value of backlog is **16** and the minimum value is **0**.
 - If the value of backlog is less than or equal to **0**, use **1**.
 - During listening, lwIP does not support automatic binding. Therefore, **bind()** must be called before **listen()**.
 - The **listen()** interface can be called in multiple channels. If listening is enabled for the socket, the socket updates the listening backlog.
 - The new backlog value applies only to new input connection requests.
- The **recv()** interface has the following restrictions:
 - When receiving the next expected data segment, lwIP updates the receiving cache list.
 - If an out-of-order data segment is adjacent to the received data segment, lwIP combines the two data segments into one, and places the combined data segment into the receiving cache list.
 - The UDP or RAW socket does not support **MSG_PEEK**.
 - UDP or RAW does not report memory allocation failures for received packets.
 - **MSG_WAITALL** is not supported, but **MSG_DONTWAIT** is supported.
 - When the **SO_RECVTIMEO** socket option is set, the socket is not marked as **O_NONBLOCK** (non-blocking). Because the socket times out, non-data is received, an error is returned, and the **ETIMEDOUT** set is displayed.
- The **recvfrom()** interface has the following restrictions:
 - The UDP or RAW socket does not support **MSG_PEEK**.
 - UDP or RAW does not report memory allocation failures for received packets.
 - **MSG_WAITALL** is not supported, but **MSG_DONTWAIT** is supported.
 - In a TCP socket, if possible, **recvfrom()** attempts to receive all the data from the receiving cache.
 - The TCP receive cache is a list that keeps the data segments received from the peer end. If the application calls the **recv** function to obtain

data, the function obtains the first entry from the list and returns it to the application. This function does not repeatedly receive entries from the list to populate the complete user buffer.

- When receiving the next expected data segment, lwIP updates the receiving cache list. If an out-of-order data segment is adjacent to the received data segment, lwIP combines the two data segments into one, and places the combined data segment into the receiving cache list.
- If **length** is set to **0**, the protocol stack returns **-1** and **errno** is set to **EINVAL**. However, the POSIX specification does not specify this setting. If the Linux implementation of **recv()** is incorrect, **0** is returned when the cache size is set to **0**.
- The **MSG_TRUNC** flag is not supported.
- The **send()** and **sendmsg()** interfaces have the following restrictions:
 - The maximum length of data that can be sent through UDP and RAW connections is 65,332 bytes. If longer data is sent, the return value is **-1** and **errno** is **ENOMEM**.
 - Only the **MSG_MORE** and **MSG_DONTWAIT** flags are supported. Other flags such as **MSG_OOB**, **MSG_NOSIGNAL**, and **MSG_EOR** are not supported.
- The **sendto()** interface has the following restrictions:
 - The maximum length of data that can be sent through the AF_INET/AF_INET6 UDP and RAW connections is 65,332 bytes. If longer data is sent, the return value is **-1** indicating failure and **errno** is **ENOMEM**.
 - Only the **MSG_MORE** and **MSG_DONTWAIT** flags are supported. Other flags such as **MSG_OOB**, **MSG_NOSIGNAL**, and **MSG_EOR** are not supported.
- The **socket()** interface has the following restrictions:
 - Only **SOCK_RAW** supports **PF_PACKET**.
 - The AF_INET socket supports the **SOCK_RAW**, **SOCK_DGRAM**, and **SOCK_STREAM** types.
 - AF_PACKET supports only the **SOCK_RAW** type.
- The **write()** interface has the following restrictions:
 - For sockets that are not marked with **O_NONBLOCK**, have the **SP_SENDFD** option set, and have a longer duration than the timeout period, lwIP fails and **errno** is **EAGAIN**.
- The **select()** interface has the following restrictions:
 - The **select()** interface does not update the timeout parameter to display the remaining duration.
 - **FD_SETSIZE** is the configurable compilation time in lwIP. The application must ensure that the boundary value is not violated. lwIP does not verify this during running.
- The **fcntl()** interface has the following restrictions:
 - Only the **F_GETFL** and **F_SETFL** commands are supported. For **F_SETFL**, the parameter **val** supports only **O_NONBLOCK**.
 - The PF_PACKET socket supports the **F_SETFL** and **F_GETFL** options.
- Any IP address change may cause the following behaviors:

- TCP
 - All existing TCP connections are disconnected, and **ECONNABORTED** is displayed for any operation on these connections.
 - All boundary IP addresses are changed to new IP addresses.
 - If there is a listening socket, it will accept the connection on the new IP address.
- UDP
 - The IP addresses of all sockets are changed to new IP addresses, and the communication continues on the new IP addresses.
 - Multiple binding-based call behaviors: A call to the **bind()** interface changes the port binding of the UDP socket.
 - A UDP socket changes the local binding port.
 - A UDP socket does not change the local binding port. If multiple binding-based calls are performed, a failure message is returned and **errno** is **EINVAL**.
 - On AF_INET/AF_INET6 SOCK_RAW sockets, multiple binding-based calls update the local IP address.
 - On the PF_PACKET socket, multiple binding-based calls change the binding to a new interface number.
 - The AF_INET/AF_INET6 SOCK_RAW binding does not check whether the socket address is available.
- The following interfaces do not support the **PF_PACKET** option:
 - accept()
 - shutdown()
 - getpeername()
 - getsockname()
 - listen()
- For ping6, if the application creates raw ICMPV6 messages, the protocol stack does not store the statistics of these messages. The application must keep the count and process the statistics.
- When a data packet is sent to a link-local address, the link-local address of the current netif must be set. Otherwise, the route is returned, indicating that the network is unreachable.
- IPv6 neighbor discovery resolution options are supported, which are defined in RFC 4861. To balance subsequent expansion and existing implementation, the protocol stack ignores any unrecognized option in the received ND packet and continues to process the packet. Perform verification to check the programs of the specific types that may affect the subsequent expansion.
- When the network mask is set to **0.0.0.0**, the following restrictions apply:

The network mask of an interface cannot be set to **0.0.0.0**. In this case, the behavior of the lwIP protocol stack is different from that of Linux.

For example, there is an **eth0** interface and a **loopback** interface. Run the following command:



```
ifconfig eth0 netmask 0.0.0.0
```

After the **ifconfig** command is executed successfully, the network mask of **eth0** is set to a specific value. However, interference occurs in the route, and even the loopback ping fails. As a result, the destination is unreachable.

- The **SO_ATTACH_FILTER** option in **setsockopt()** has the following restrictions:
LSF_LDX hack of the IP address header length cannot be loaded because **LSF_MSH** is not supported. Therefore, the following filtering modes are not supported:

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

- When a socket is bound to an IP address for which **bind()** has been configured, the IP address is changed. If an application calls **listen()**, the error message **ECONNABORTED** is displayed. The socket bound to the old IP address does not update the new IP address configured on the interface, but the status changes to **ERR_ABRT**. Therefore, when **listen()** is called, the error message **ECONNABORTED** is displayed.
- An IPv6 raw socket cannot receive IPv4 packets or packets containing IPv6 addresses mapped to IPv4.
- The restrictions on pinging an IPv4 address are as follows:
There is no matching of the ICMP identifier or ICMP sequence number to maintain the ping session, and there is no sequence of ping responses from the peer end.
- The **ifconfig shell** command has the following restrictions:

- When you run the following command to set the MAC address, the MAC address must be provided in the extended form.

```
ifconfig <ifname> hw ether <MAC>
```

For example, to set the MAC address to **ab:cd:0e:12:34:56**, run the following command:

```
ifconfig <ifname> hw ether ab:cd:0e:12:34:56
```

The command cannot be as follows:

```
ifconfig <ifname> hw ether ab:cd:e:12:34:56
```

- If the IPv6 address of the protocol stack is obtained from the DHCPv6 server, call the **netifapi_dhcp6_disable()** and **netifapi_dhcp6_cleanup()** interfaces to stop the service before changing the IPv6 address using the **ifconfig** command.



4 Cyber Security

4.1 Cyber Security Statement

4.1 Cyber Security Statement

To prevent sensitive information from being disclosed, pay attention to the following security precautions:

Debugging log: You can enable or disable **DebugFlag** by configuring the macro **LWIP_DBG_TYPES_ON** in the **opt.h** header file. This macro is disabled by default.

Table 4-1 Debugging disabling option

Selection Code	Recommended Value	Impact
LWIP_DEBUGF	LWIP_DBG_OFF	When LWIP_DBG_ON is set to 1 to enable the debugging function, some sensitive user information may be disclosed.

Read this statement carefully to prevent security risks. Pay attention to the following safety precautions:

- Define the **LWIP_RANDOM** function macro in the **lwipopts.h** header file. The DHCP and DNS use the corresponding functions to randomly generate service IDs. You should define the **LWIP_RANDOM** macro as an appropriate random number generator function.
- The application data sent to lwIP based on UDP or TCP is temporarily stored in the lwIP buffer. Before the buffer is released, the data in the buffer is not cleared explicitly. The preceding content is based on the assumption that the application does not provide sensitive user information to lwIP in plaintext. This is because applications usually encrypt sensitive user information using transmission security protocols (such as TLS or DTLS) and send the encrypted messages to lwIP.



5 FAQs

1. Does lwIP support the route command?

Answer: No. lwIP does not support the route command.

2. What is the routing mechanism of lwIP?

Answer: lwIP does not support route selection. If the **IP_FORWARD** flag is enabled, lwIP supports forwarding. Network port packets are sent through routes.

3. Can lwIP run properly when netif is not added?

Answer: No. Netif must be added and the callback function must be executed.

4. Does lwIP support IPv4 multicast?

Answer: Yes. This function can be enabled by setting **LWIP_IGMP** to 1 and using **setsockopt()** to set **IP_ADD_MEMBERSHIP** and **IP_DROP_MEMBERSHIP**.

5. What is the impact of IP address changes on TCP connections?

Answer: All existing TCP connections will be discarded, and **ECONNABORTED** will be returned for any TCP connection operation.

All bound IP addresses are changed to new IP addresses.

If there is a listening socket, it will accept the connection on the new IP address.

6. What is the impact of IP address changes on UDP connections?

Answer: The IP addresses of all sockets are changed to the new IP address. The communications are continued on the new IP address.

7. According to section 5 in RFC 2710, does the protocol stack support the sending of an MLD message including the solicited-node multicast address?

Answer: According to section 5 in RFC 2710, when the host is working, an MLD message should not be sent to all the node multicast addresses on the link, such as FF02::1.

According to section 4 in RFC 2710:

- If a node receives a multicast address report message from another node through an interface and the timer of the node is running, the node stops the timer and does not send the multicast address report message any more. This suppresses the replication of reports on the link.
- When a DONE message is sent, if the latest REPORT message of the node is suppressed because it is listening to other REPORT messages, it may stop sending messages because there is probably a device listening to this address on the same link. This mechanism can be disabled but is enabled by default.

8. What will happen if an unsolicited NA message whose destination address is a multicast address is received when the access status of the neighbor cache data is not INCOMPLETE?

Answer: The status is changed to **STALE**.

9. What happens when the protocol stack receives an RA message from a router?

Answer: After receiving an RA message from a router, the protocol stack sends an NS message to the default route address to resolve the default route address.

10. How many IPv6 addresses can be added?

Answer: A maximum of three addresses are supported, including link-local addresses. If more than two IPv6 addresses are added, the first address will be replaced by the new address, which will be further optimized in the miniaturization version.

11. What will happen if duplicate address detection fails?

Answer: A callback function will be registered to detect duplicate addresses. If duplicate address detection fails, this callback function can be called to delete the original IPv6 address and add a new address. This feature is not supported in the miniaturization scenario.

12. Can lwIP receive messages from the Peer_Node through the interfaces installed on the device on which lwIP runs?

Answer: When lwIP runs on a device equipped with multiple interfaces, lwIP can receive the message whose destination address is a broadcast IP address from the Peer_Node through these interfaces. Once the **recv()** function is called, the number of messages that can be received by the application depends on the number of times that the interface is configured.

For example:

Step 1 Configure the IP addresses as follows (the device has two interfaces):

Peer_Node Device (lwIP)

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

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

Step 2 Set the **SO_BROADCAST** socket option for the lwIP socket.

Step 3 Bind the lwIP socket to **INADDR_ANY**.

Step 4 Sends DATA messages received from the Peer_Node to the device whose destination address is **192.168.2.255**.

----End

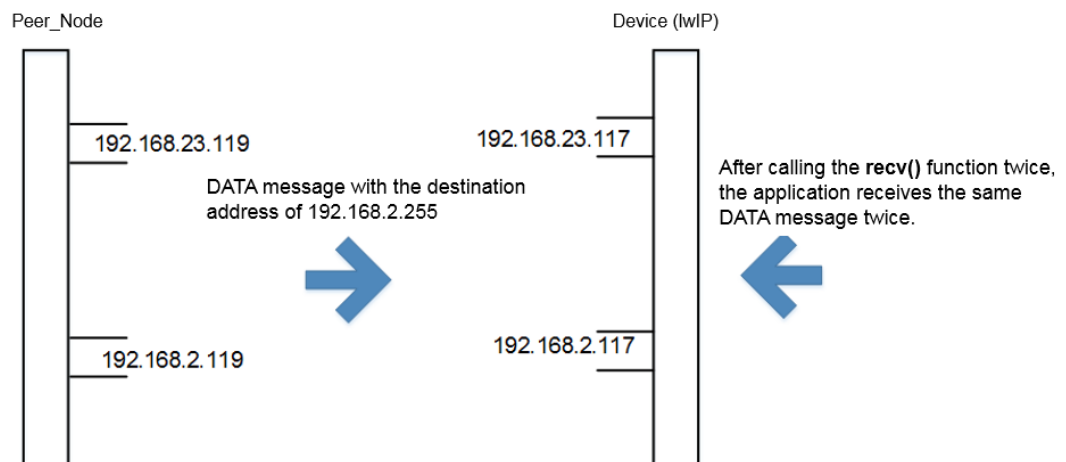
Result:

The device receives messages from the Peer_Node through the interfaces **192.168.23.117** and **192.168.2.117**. After calling the **recv()** function twice, the application receives the same message twice.

NOTE

When the interface **192.168.23.117** receives a message from the Peer_Node, it finds that the message is not sent to the correct interface. In this case, the interface forwards the message to the **192.168.2.117** interface.

Figure 5-1 Receiving a message from the Peer_Node



13. How do I adjust the retransmission times and interval of SYN packets for TCP connection setup?

Answer: You can change the values of **TCP_SYNMAXRTX** and the global variable **tcp_persist_backoff**. Note that the number of elements of **tcp_persist_backoff** must be at least **TCP_SYNMAXRTX + 1**. For example, you can perform the following operations to greatly shorten the timeout interval of the **connect** interface when the peer end does not reply with SYN+ACK.

```
#define TCP_SYNMAXRTX 3
static const u8_t tcp_persist_backoff[4] = { 1, 2, 4, 8 };
```

14. Why is a closed TCP connection displayed in state 10 after the netstat command is executed?

Answer: When the low-power mode is enabled, the PCB release delay is **TIME_WAIT**. That is, a TCP PCB is released only when no idle PCB is available



during the next allocation. Therefore, the connection in state 10 (**TIME_WAIT**) displayed by the **netstat** command does not affect the setup of a new connection. To disable the low-power mode, change the **LWIP_LOWPOWER** macro in the compilation option **opt.h** to **0**.

If the low-power mode is disabled, the connection is released after a period of time **TIME_WAIT**, which can be set by the **TCP_MSL** macro in **tcp_priv.h**.



6 Terms

Acronym / Abbreviation	Term	Description
ARP	Address Resolution Protocol	ARP is a network protocol that translates IP addresses into physical addresses.
DHCP	Dynamic Host Configuration Protocol	DHCP is a standard networking protocol used on IP networks. It dynamically allocates network configuration parameters, such as IP addresses, to interfaces and services.
lwIP	lightweight TCP/IP protocol stack	LwIP is a lightweight open-source TCP/IP protocol stack widely used in embedded systems.
Lite OS	Huawei LiteOS	Huawei LiteOS is an operating system developed by Huawei based on the CMSIS software standard.
ICMP	Internet Control Message Protocol	ICMP is used by network devices such as routers to send error messages to indicate that the requested service is unavailable or cannot reach the host.



Acronym / Abbreviation	Term	Description
IP	Internet Protocol	IP is a protocol in the TCP/IP protocol suite that controls the encapsulation of data packet fragments into packets, the routing of packets from the sending station to the destination network and station, and the composition of the original data information at the destination station. The IP protocol runs at the Internet layer of the TCP/IP model, corresponding to the network layer of the ISO/OSI model.
IoT	Internet of Things	IoT is a network that carries information, such as the Internet and traditional telecom networks. It enables all common objects that can perform independent functions to interconnect with each other.
TCP	Transmission Control Protocol	TCP is a protocol within TCP/IP that governs the breakup of data messages into packets to be sent using Internet Protocol (IP), and the reassembly and verification of the complete messages from packets received by IP. As a connection-oriented, reliable protocol that can help ensure error-free delivery, TCP corresponds to the transport layer in the ISO/OSI reference model.
UDP	User Datagram Protocol	UDP is a TCP/IP standard protocol that allows applications on one device to send datagrams to applications on another device. UDP uses IP to deliver datagrams, providing application programs with unreliable connectionless packet delivery services. That is, UDP messages may be lost, repeated, delayed, or out of order. The destination device does not proactively check whether the correct data packet is received.