TCPIP Dual Stack System User Guide

Version 1.50

For use with systems using the TCP/IP IPv4 and IPv6 Stack modules.

For use with module versions 6.35 and above.

Exported on 10/09/2018

All rights reserved. This document and the associated software are the sole property of HCC Embedded. Reproduction or duplication by any means of any portion of this document without the prior written consent of HCC Embedded is expressly forbidden.

HCC Embedded reserves the right to make changes to this document and to the related software at any time and without notice. The information in this document has been carefully checked for its accuracy; however, HCC Embedded makes no warranty relating to the correctness of this document.

Table of Contents

1 System Overview 7
1.1 Introduction 8
Application Layer Components 9
Transport Layer Components 10
Network Layer Components 10
Differences between IPv4 and IPv6 11
1.2 Feature Check 12
Network Interfaces 12
User/Application Programming Interfaces 12
Security 12
Protocol Support 13
1.3 Packages and Documents 14
Packages 14
Documents 15
1.4 Change History 16

2 Stack Overview 17
2.1 ARP - IPv4 Only 17
Example 17
2.2 ICMP 18
ICMPv4 18
ICMPv4 Message Types 19
ICMPv6 20
2.3 IGMP - IPv4 Only 20
2.4 MLD - IPv6 Only 21
Queries and the Querier 21
State Change Reports 21
Networks with both MLDv1 and MLDv2 22
2.5 Neighbor Discovery - IPv6 Only 22
Packet Types 22
Neighbor Unreachability Detection (NUD) 23
Duplicate Address Detection (DAD) 23

3 Source File List 24
3.1 API Header Files 24

Copyright HCC Embedded 2018

3.2 Version File 24
3.3 Configuration Files 25
3.4 Source Files 26

4 Configuration Options 27
4.1 IP Options 27
General Options 27
IP Options 28
Checksum Options 28
Route Options 28
Defragmentation Options 29
Default Buffer Counts 29
4.2 ARP Options - IPv4 Only 30
4.3 ICMPv4 Options - IPv4 only 31
4.4 ICMPv6 Options - IPv6 only 32
4.5 IGMP Options - IPv4 Only 32
4.6 MLD Options - IPv6 Only 33
4.7 ND Options - IPv6 Only 34
4.8 Network Driver Memory Pool Options 35

5 Application Programming Interface 36
5.1 Module Management 37
ip_stack_init 37
ip_stack_start 38
ip_stack_stop 39
ip_stack_delete 40
5.2 IP Functions 41
ip_enter_task 42
ip_exit_task 43
ip_addr_to_str 44
ip_str_to_addr 45
ip_get_config 46
ip_set_config 47
ip_get_fqdn 50
ip_get_default_fqdn 51
ip_set_default_fqdn 52
ip_ifc_get_mtu 53

ip_register_config_ntf 54
5.3 IP Functions - IPv4 only....55
ip_v4_set_alias....56
ip_v4_delete_alias....57
5.4 IP Function - IPv6 only....58
ip_v6_get_host_addr - IPv6 only....59
5.5 ARP Functions - IPv4 Only....60
arp_add_static_entry....61
arp_del_entry....62
arp_set_lease_time....63
arp_find_first_entry....64
arp_find_next_entry....65
arp_get_hwaddr_from_ipaddr....66
arp_get_ipaddr_from_hwaddr....67
5.6 ICMP Functions....68
icmp_ping....69
icmp_ping_state....70
icmp_ping_cb....71
5.7 IGMP Functions - IPv4 Only....72
igmp_add_membership....73
igmp_drop_membership....74
5.8 MLD Function - IPv6 only....75
mld_set_mc_listen....75
5.9 Network Driver Memory Pool Functions....77
ip_nwd_get_buf....78
ip_nwd_get_buf_ext....79
ip_nwd_release_buf....80
ip_nwd_release_buf_ext....81
ip_pool_buf_config....82
ip_pool_add....83
ip_pool_del....84
ip_pool_get_prop....85
ip_ifc_get_pool....86
ip_ifc_set_pool....87
5.10 Routing Functions....88
ip_route_add....89
ip_route_del....90
ip_route_get....91

ip_route_get_hdl 92
ip_route_get_hdl_fqdn..93
ip_route_get_config 94
5.11 Error Codes..95
5.12 Types and Definitions 98
t_ip_addr 98
t_ip_opt 98
t_ip_route 99
t_ip_config...100
IP Config Flags..100
IP Config Link States 101
t_ip_port..101
t_ip_ntf 101
t_ip_get_buf_tn 102
t_ip_get_buf..102
t_arp_find..103
t_ip_pool_qprop 103
t_ping_cb_fn 104
Ping States...104
IP Configuration Options..105
Invalid Handles 105
Timeout Values 105
Notifications..106
t_mld_listen 107
t_ip_config_ntf_fn 107
6 Code Examples...108
6.1 Initializing the Stack 109
6.2 Starting the Stack 110
6.3 Stopping the Stack...111
7 Integration..112
7.1 Utilities...112
7.2 OS Abstraction Layer: OAL...112
7.3 PSP Porting 113
8 Memory Management...114
8.1 Network Drivers..114

8.2 Buffer Pools 114
8.3 Protected and Unprotected Network Driver Functions 115

9 Checking the Status of the Physical Ethernet Cable 116

1 System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

- Introduction – describes the main elements of the module. This section includes a diagram showing the position of this module within HCC's TCP/IP stack.
- Feature Check – summarizes the main features of the module as bullet points.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

Note: To download this manual as a PDF, see TCP/IP PDFs.

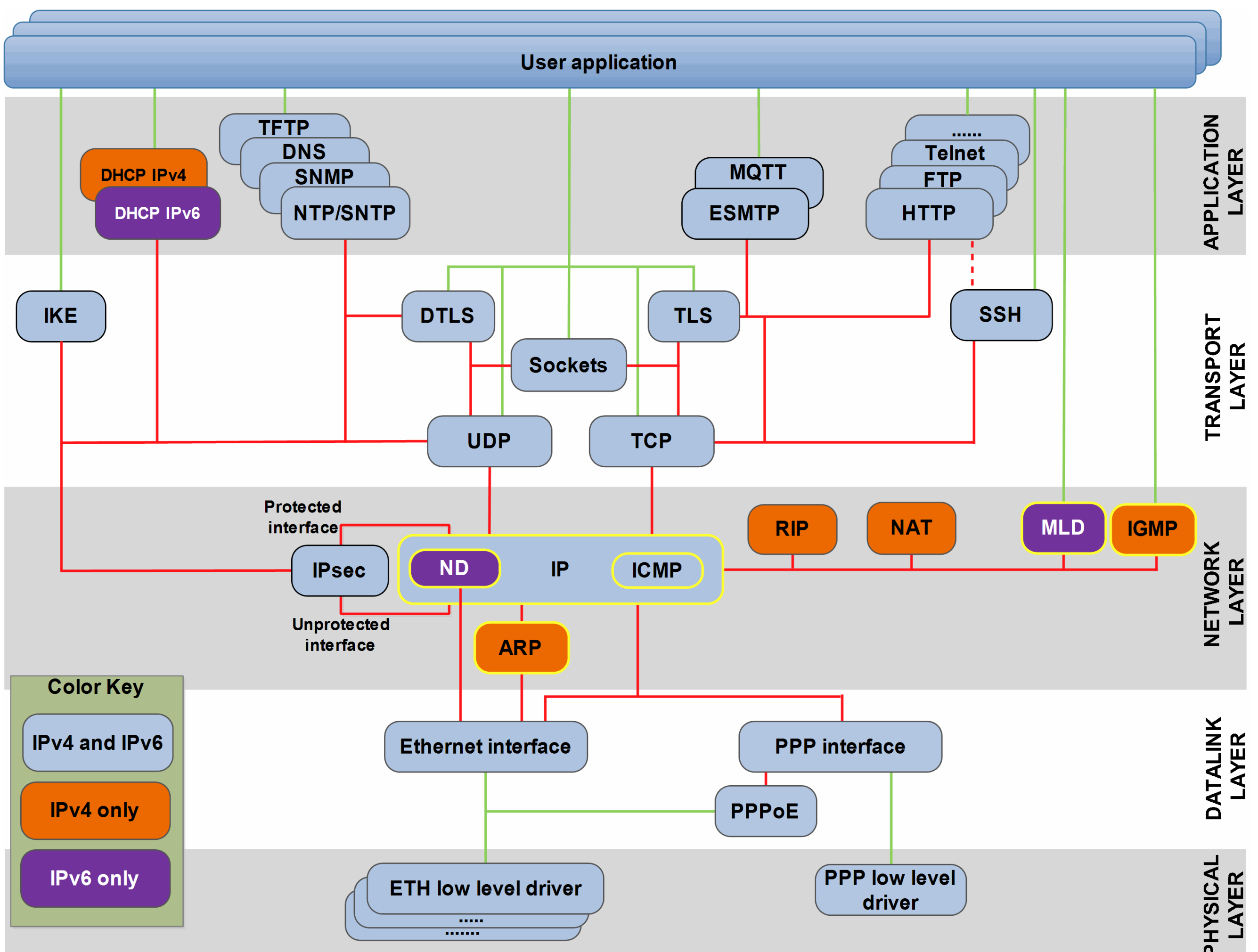
1.1 Introduction

This guide is for those who want to implement a TCP/IP dual stack system. The HCC dual stack can be used in three different base configurations:

- IPv4 standalone.
- IPv6 standalone.
- Dual stack - both IPv4 and IPv6.

This guide describes the organization of HCC Embedded's MISRA-compliant TCP/IP stack and documents the base TCP/IP system, to which the modular components of the stack are added. The system is modular in construction, so that just setting the defines for IPv4 and IPv6 sets the system up. Once this is done, the software (and this manual) can be used seamlessly.

The following diagram shows the stack components. Green lines show interfaces available to users of the stack and red lines show interfaces internal to the TCP/IP system. The key shows which IP versions the components relate to.



This manual covers the IP core modules marked with a yellow border in this diagram - ARP, ICMP, IGMP and Neighbor Discovery (ND) - plus routing and the network driver memory pool. Other HCC manuals cover

specific areas such as network drivers, TCP, and UDP. In addition, each Application Layer package has its own manual.

Application Layer Components

This table lists the Application Layer stack components:

Module	Description
UDP-based	
DHCP IPv4	Dynamic Host Control Protocol, used by a client (a computer or other device) to get an IP address automatically from a remote DHCP server. There are separate IPv4 DHCP client and server applications.
DHCP IPv6	IPv6 DHCP client application.
TFTP	Trivial File Transfer Protocol, used to transfer files between hosts over a TCP-based network.
DNS	Domain Name System,a distributed naming system for resources connected to the Internet or other networks.
SNMP	Simple Network Management Protocol, handles communication between SNMP agents and managers.
NTP/SNTP	Network Time Protocol/Simple Network Time Protocol, used to synchronize clocks on computers in packet-switched networks.
TCP-based	
MQTT	A "publish and subscribe" lightweight messaging protocol for use over TCP/IP.
ESMTP	Extended Simple Mail Transfer Protocol (ESMTP), used for email transmission.
Telnet	A standard method of interfacing terminal devices and terminal-oriented processes to each other.
FTP	File Transfer Protocol, used to transfer files between hosts over a TCP-based network.
HTTP	Hypertext Transfer Protocol. There is also a secure version, HTTPS.

Transport Layer Components

This table lists the Transport Layer stack components:

Module	Description
IKE	Internet Key Exchange, part of the IP security architecture.
DTLS	Datagram Transport Layer Security (TLS), provides secure communication over UDP-based networks.
TLS	Transport Layer Security (TLS), provides secure communication over TCP-based networks.
Sockets	BSD Sockets, allows communication across a TCP/IP network using the standard BSD Socket calls.
UDP	User Datagram Protocol, used to send datagrams to other hosts.
TCP	Transmission Control Protocol.
SSH	Secure Shell, a portable low footprint server that includes SSH's Authentication, Transport Layer, and Connection Layer protocols.

Network Layer Components

This table lists the Network Layer stack components:

Module	Description
IPsec	Internet Protocol Security, part of the IP security architecture.
ND	Neighbor Discovery, the IPv6 replacement for ARP.
IP	Internet Protocol.
ICMP	Internet Control Message Protocol.
RIP	Routing Information Protocol, used by IP stacks to exchange routing table information.
NAT	Network Address Translation. This allows client IP hosts on a stub network connected to the Internet to access Internet hosts without having to obtain and assign "real" IP addresses for each host.
MLD	Multicast Listener Discovery, used by IPv6 routers to discover multicast listeners on a directly attached link.
IGMP	Internet Gateway Message Protocol. This manages local subnet group membership in IPv4.
ARP	Address Resolution Protocol - in IPv4 this finds the hardware address of a host from its known IP address.

Copyright HCC Embedded 2018

Differences between IPv4 and IPv6

The main differences between IPv4 and IPv6 are the following:

- Address length.
- IPv6 uses a hierarchical addressing structure.
- IPv4 distributed network addresses fairly randomly across the world (and beyond), giving a lot of responsibility for handling this to routers.
- ARP is used in IPv4, Neighbor Discovery in IPv6.
- In IPv4 IGMP is used to manage local subnet group membership. In IPv6 this is done by Multicast Listener Discovery (MLD).
- IPv4 must be configured either manually or through DHCP. IPv6 does not require DHCP.

Copyright HCC Embedded 2018

1.2 Feature Check

The key features of the HCC TCP/IP stack are the following:

- Fully MISRA-compliant.
- Conforms to the HCC Advanced Embedded Framework.
- Supports both IPv4 and IPv6, but allows either to be disabled if required.
- Designed for integration with both RTOS and non-RTOS based systems.
- Small RAM and ROM footprint.
- High performance.
- Supports multiple network interfaces.
- Routing module provided.
- Provides both native and Sockets interfaces.
- Wide range of TCP and UDP applications is available.

Network Interfaces

These are included in the base system but documented in separate user guides. The package:

- Supports multiple network interfaces.
- Supports routing between network interfaces
- Provides fast/zero copy between network interfaces where common memory pools are defined.
- Works with HCC's network driver interface specification.
- Range of tested drivers for standard micro-controllers and external Ethernet controllers is available.

User/Application Programming Interfaces

These are included in the base system but documented in separate user guides. The package:

- Provides Sockets User API.
- Provides MISRA-compliant user API.

A wide range of TCP- and UDP-based application software is available.

Security

The following extensions to the base system are available, though not included in the package:

- TLS/DTLS.
- IPsec.
- IKE.
- Embedded Encryption Manager (EEM).

Protocol Support

The base system components support the following protocols:

- IPv4 - RFC 791.
- IPv6 - RFC 2460. For dual stack operation, see RFC 4213.
- TCP - RFC 793.
- UDP - RFC 768.
- ARP - RFC 826 with ARP probing (RFC 5227).
- ICMPv4 - RFC 792.
- ICMPv6 - RFC 4443.
- IGMPv3 - RFC 4604. IGMPv3 is used to manage membership in IPv4 multicast groups.
- ND - RFC 4861.
- MLDv2 - RFC 4604. MLDv2 is used to manage membership in IPv6 multicast groups.
- BSD Sockets

All of the listed RFCs are referenced and their features used appropriately for a deeply embedded design.

1.3 Packages and Documents

Packages

The table below lists the full set of packages. For all systems the IP base package is mandatory. All other components are optional and depend on your particular system's design and requirements.

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
mip_base	The IP base module with components not specific to IPv4 or IPv6.
ip_base_v4	The IPv4 base module with components specific to IPv4. Either this, **ip_base_v6**, or both shall be set.
ip_base_v6	The IPv6 base module with components specific to IPv6.
mip_tcp	The TCP module (if TCP is required).
mip_udp	The UDP module (if UDP is required).
mip_sec	The IPsec package (if IPsec support is required).
ip_app_dhcp_v4	This is needed if IP addresses need to be obtained from an IPv4 DHCP server.
ip_app_dhcp_v6	This is needed if IP addresses need to be obtained from an IPv6 DHCP server.
psp_template_base	The base Platform Support Package (PSP).
oal_base	The base OS Abstraction Layer (OAL) package.
nw_drv_base	The base network driver package.
mutil_timer	The MISRA-compliant timer utility.

Documents

For an overview of HCC's TCP/IP stack software, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow this *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC TCP/IP Dual Stack System User Guide

This is this document.

HCC TCP User Guide

This document describes the HCC TCP module that allows an application to send and receive data using TCP connections across a TCP/IP network.

HCC UDP User Guide

This document describes the HCC UDP module that allows an application to send datagrams.

HCC Network Driver User Guide

This document describes the network driver base system and the network driver interface specification.

Other Documents

User manuals for components like network interfaces, user interfaces, and TCP applications are included in the component packages.

> **Note:** Although every attempt has been made to simplify the system's use, you need a good understanding of the requirements of the systems you are designing in order to obtain the maximum practical benefits. HCC Embedded offers hardware and firmware development consultancy to help you implement your system; contact support@hcc-embedded.com.

1.4 Change History

This section describes past changes to this manual.

- To download this manual or a PDF describing an earlier software version, see TCP/IP PDFs.
- For the history of changes made to the package code itself, see History: mip_base.

The current version of this manual is 1.50. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
1.50	2018-10-09	6.35	Added IP_V4_ALIAS_COUNT configuration option. Added functions **ip_v4_set_alias()** and **ip_v4_delete_alias()**. Added two alias-related error codes.
1.40	2018-05-23	6.30	Added ARP_PROBE_ENABLE configuration option for ARP probing.
1.30	2018-02-06	6.28 R2	IGMP files moved from **mip_base** to **ip_base_v4**.
1.20	2017-08-17	6.19	Changes to *Packages* list and to *IP Options*.
1.10	2017-06-20	6.18	New *Change History* format. Manual numbering changed to n.10, n.20, etc.
1.04	2017-06-07	6.17	Added **ip_v6_get_host_addr()** function.
1.03	2017-03-28	6.13	Updated Change History.
1.02	2017-03-03	6.12	Changed "TCP" to "UDP" on page 8.
1.01	2017-03-02	6.12	Changed description of *t_ip_get_buf_tn.*
1.00	2017-03-01	6.12	First online version.

2 Stack Overview

This section describes the TCP/IP stack components.

2.1 ARP - IPv4 Only

The Address Resolution Protocol (ARP) is used to resolve IPv4 addresses into network (link) layer addresses. It does not support IPv6 addresses.

Address resolution means converting IP Addresses to 48 bit Ethernet Addresses for transmission over Ethernet Hardware. ARP was originally developed for 10MBit Ethernet but now supports other transmission media as well.

Example

In this example, two computers on a LAN in a building are connected to each other using Ethernet cables and network switches, with no gateways or routers between them. Computer X wants to send to a packet to Computer Y. The process is:

1. Computer X uses DNS to obtain Computer Y's IP address, 192.168.3.14.
2. Computer X also needs Computer Y's MAC address. It starts by using its cached ARP table to look up 192.168.3.14 for any existing records of Computer Y's MAC address (00:ed:12:b2:12:ac).
3. If it finds the MAC address, Computer X broadcasts an Ethernet frame with the destination address 00:ed:12:b2:12:ac onto the link; this contains the IP packet.
4. If Computer X does not find 192.168.3.14 in its cache, it sends an ARP broadcast message (destination FF:FF:FF:FF:FF:FF MAC address), requesting an answer for 192.168.3.14. This broadcast will be accepted by all computers that receive it.
5. Computer Y responds, giving its MAC and IP addresses. It may insert an entry for Computer X into its ARP table in case it is needed in future.
6. Computer X caches the response information in its ARP table.
7. Computer X sends the packet to Computer Y.

2.2 ICMP

Different versions of the Internet Control Message Protocol (ICMP) are used in IPv4 and IPv6 networks.

ICMPv4

ICMP is used by a gateway or destination host when it needs to communicate with a source host, typically to report an error in datagram processing. ICMP is an integral part of IP and must be implemented by every IP module. These messages are used to provide feedback about problems on the network.

ICMP messages are sent by gateways to source hosts in situations including the following:

- A datagram cannot reach its destination - according to the gateway's routing tables, the destination network specified is unreachable.
- The gateway does not have sufficient buffering capacity to forward a datagram.
- The gateway can direct the host to send traffic on a shorter route.
- A gateway needs to fragment a datagram before it can forward it, but the Don't Fragment flag is on. In this case the gateway discards the datagram and may return a Destination Unreachable message.

A destination host may send an ICMP Destination Unreachable message to the source host if the IP module cannot deliver the datagram because the indicated protocol module or process port is not active.

ICMP packets have a Time to Live value in seconds. This field is decremented at each machine that processes the datagram, so should be at least as great as the number of gateways that the datagram will traverse; this is configurable.

ICMPv4 Message Types

ICMP message types are as follows:

Value	Meaning	Description
0	Echo Reply	Used to reply to an ICMP ping.
3	Destination Unreachable	A gateway uses this to notify a source host that the destination host is unreachable. A gateway uses this when a datagram must be fragmented for forwarding yet the Don't Fragment flag is on. A destination host may send this message to the source host if the protocol module or process port specified in a datagram is not active.
4	Source Quench	A gateway that discards datagrams due to lack of buffer space may send this message to the source host. A destination host may send this message if datagrams arrive too fast to be processed.
5	Redirect	A gateway uses this to advise a host to send its traffic for a network directly to a different gateway as this is a shorter path to the destination.
8	Echo	This is used to ping an address.
11	Time Exceeded	After a gateway finds the Time to Live field is 0 so discards a datagram, it may notify the source host by using this message. A host that cannot reassemble a fragmented datagram within its time limit (due to missing fragments) discards the datagram and may then send this message.
12	Parameter Problem	A gateway or host processing a datagram finds a problem in the header so cannot process it. It discards the datagram and may send the source host this message.
13	Timestamp	This is used for time synchronization.
14	Timestamp Reply	This is used to reply to a Timestamp message.
15	Information Request	A host may use this to find out the number of the network it is on.
16	Information Reply	A gateway uses this to reply to an Information Request message.

Copyright HCC Embedded 2018

ICMPv6

ICMPv6 messages are transported by IPv6 packets. The message values have changed from those in the above table, though this is not significant here, and there is one additional error message:

Value	Meaning	Description
2	Packet too big	A router sends this in response to a packet that it cannot forward because the packet is larger than the outgoing link's MTU (Maximum Transmission Unit).

ICMPv6 packets are also used for the messages used in the following IPv6 protocols:

- Multicast Listener Discovery - messages like Multicast Listener Query and Multicast Listener Report.
- Neighbor Discovery - IPv6 Only - messages including router solicitation, router advertisement, neighbor solicitation, router advertisement and redirect.

2.3 IGMP - IPv4 Only

Internet Gateway Message Protocol v3 (IGMPv3) is used by hosts and routers on IPv4 networks to manage group membership of multicast groups on local subnets. It is a network layer protocol.

Hosts request membership of a group from their local router; a router listens for these requests and also sends out subscription queries at set intervals.

The main enhancement of IGMPv3 over IGMPv2 is support for source-specific multicast and Membership Report aggregation.

IGMP uses three types of message:

Membership Queries

These are sent by a multicast router to determine which multicast addresses are of interest to systems attached to its network. There are two types:

- General queries - sent by a router at intervals to refresh the group membership state for all systems on its network.
- Group-Specific queries - used to determine the reception state for a specific multicast address.

These queries allow a router to determine whether any devices want to receive messages sent to a multicast group from a source address specified in a list of unicast addresses.

Membership Reports

Membership reports are used by hosts to report to neighboring routers either their interfaces' current multicast reception state, or any changes in the multicast reception state.

Copyright HCC Embedded 2018

Leave Group Messages

A host sends a Leave Group message when it wants to end its membership of a group.

2.4 MLD - IPv6 Only

IPv6 routers use Multicast Listener Discovery (MLD) in two ways:

- To discover the presence of multicast listeners (nodes that wish to receive multicast packets) on directly attached links.
- To discover which multicast addresses these neighboring nodes are interested in.

MLD's purpose is to allow each multicast router to learn, for each of its directly attached links, the multicast addresses and sources that have interested listeners. Information gathered by MLD is provided to whichever multicast routing protocol the router uses, to ensure that multicasts are delivered to all links on which there are listeners interested in such packets.

MLDv2 is interoperable with MLDv1 but in MLDv2 a node can report its interest in receiving IPv6 multicast group transmissions defined as follows:

- With a particular multicast address only from specific source addresses.
- From all sources except for specified source addresses.

This is termed "source filtering".

A multicast router may itself listen on one or more multicast addresses. In this case it acts as both multicast router and multicast address listener.

The IPv4 equivalent of MLDv2 is IGMPv3.

Queries and the Querier

If there are several multicast routers on the same subnet, a mechanism is used to elect a single multicast router to act as the *Querier*. All multicast routers on the subnet listen to messages sent by multicast address listeners, so that they can take over the querier role if the current Querier fails. The Querier is the only router that sends periodical or triggered query messages on the subnet.

Note: If an MLDv1 router is present on the link, the Querier MUST use the lowest version of MLD present on the network.

State Change Reports

if the listening state of a node changes, it immediately reports the change by sending a State Change Report message. This report contains either Filter Mode Change records, Source List Change records, or both. To increase protocol robustness, the messages may be re-transmitted several times, in case they are missed by any multicast routers (this "robustness" option is configurable).

Networks with both MLDv1 and MLDv2

If a link has both MLDv1 routers and MLDv2 routers, the Querier must use the lowest version of MLD present on the network.

MLDv2 routers may be added to a network where some hosts have not yet been upgraded to MLDv2. To be compatible with the MLDv1 hosts, MLDv2 routers MUST operate in version 1 compatibility mode (this option is configurable).

2.5 Neighbor Discovery - IPv6 Only

Neighbor Discovery (ND) is a network layer/link layer protocol.

IPv6 nodes (hosts and routers) use ND as follows:

- To determine the link layer addresses of neighbors known to reside on attached links and to purge quickly any cached values that become invalid.
- To find neighboring routers that are willing to forward packets on their behalf.
- To keep track dynamically of which neighbors are reachable (their "reachability") and which are not, and to detect changed link layer addresses. When a router or the path to a router fails, a host actively searches for functioning alternate routes.

Packet Types

ND defines five ICMP packet types for messages:

Router Solicitation

When an interface becomes enabled, a host may send this out, requesting routers to generate Router Advertisements immediately rather than at their next scheduled time.

Router Advertisement

This is used by a router to advertise its presence, either periodically or in response to a Router Solicitation. On links that support multicasts, each router periodically multicasts a Router Advertisement packet announcing its availability. A host receives Router Advertisements from all routers and uses them to build a list of default routers.

The packet contains a list of prefixes used for on-link determination and/or autonomous address configuration; flags associated with prefixes specify the prefix's intended uses. Hosts use the advertised on-link prefixes to build and maintain a list that is used in deciding whether a packet's destination is on-link or beyond a router.

The packet contains Internet parameters such as the hop limit that hosts should use in outgoing packets.

Neighbor Solicitation

This is used by a node to determine the link layer address of a neighbor, or to verify that a neighbor is still reachable via a cached address. These are also used for Duplicate Address Detection (see below).

A node multicasts a Neighbor Solicitation that includes its link layer address, asking the target node to return its link layer address. The target returns its link layer address in a unicast Neighbor Advertisement. A single request/response pairing allows the node and neighbor to resolve each others' link layer addresses.

Neighbor Advertisement

This is a response to a Neighbor Solicitation. A node may also send these unsolicited to announce a link layer address change.

Redirect

This is used by a router to inform hosts of a better first hop for a destination.

Neighbor Unreachability Detection (NUD)

This process detects the failure of a neighbor or of the forward path to it. It seeks confirmation from two sources:

1. When possible, upper layer protocols provide a positive confirmation that previously sent data was delivered correctly (for example, have new acknowledgments been received recently?).
2. When no positive confirmation is seen, a node sends unicast Neighbor Solicitation messages to solicit Neighbor Advertisements from the next hop.

Duplicate Address Detection (DAD)

In Duplicate Address Detection, a host sends Neighbor Solicitation messages without a source address, targeting its own proposed address. Such messages trigger any node already using the address to respond with a multicast Neighbor Advertisement, indicating that the address is already in use.

3 Source File List

The following sections describe all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

Note: Do not modify any files except the configuration files.

3.1 API Header Files

These files in the directory **src/api** should be included by any application using the system. These are the only files that should be included by an application using this module. For details of the API functions, see Application Programming Interface. For the application developer, only the ICMP API file is of interest since the IP and ARP files are used internally by different stack modules.

Package/file	Description
mip_base	
api_ip.h	Internet Protocol (IP) API.
api_ip_icmp.h	Internet Control Message Protocol (ICMP) API.
api_ip_pool.h	Network Driver memory pool API.
ip_base_v4	
api_ip_arp.h	Address Resolution Protocol (ARP) API.
api_ip_igmp.h	Internet Gateway Message Protocol (IGMP) API.
ip_base_v6	
api_ip_mld.h	Multicast Listener Discovery (MLD) API.

3.2 Version File

The file **src/version/ver_ip.h** in the **mip_base** package contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

3.3 Configuration Files

These files in the directory **src/config** contain all the configurable parameters of the system and are the only files in the module that you should modify. You can configure the parameters as required. For detailed explanation of these options, see Configuration Options.

Package/file	Description
mip_base	
config_ip.h	Internet Protocol (IP) configuration parameters.
config_ip_pool.h	Network Driver memory pool configuration parameters.
ip_base_v4	
config_ip_arp.h	ARP configuration parameters.
config_ip_icmp_v4.h	ICMP IPv4 configuration parameters.
config_ip_igmp.h	IGMP configuration parameters.
ip_base_v6	
config_ip_icmp_v6.h	ICMP IPv6 configuration parameters.
config_ip_mld.h	MLD parameters.
config_ip_nd.h	Neighbor Discovery configuration parameters.

3.4 Source Files

These files are in the directory **src/ip/stack/core**. **These files should only be modified by HCC**.

Package/file	Description
mip_base files	
icmp.c and **.h**	Implement the ICMP protocol.
ifc.c and **.h**	Implement the link between a network interface and the IP stack.
ip.c and **.h**	Implement the IP protocol.
pool.c and **.h**	Implement memory pool allocation.
route.c and **.h**	Implement the IP routing module.
stack_core.c and **.h**	Implement functions that are required by multiple IP modules.
ip_base_v4 files	
arp.c and **.h**	Implement the ARP protocol.
icmp_v4.c and **.h**	Implement the ICMP protocol for IPv4.
igmp.c and **.h**	Implement the IGMP protocol.
ip_v4.c and **.h**	Implement the IP protocol for IPv4.
ip_base_v6 files	
icmp_v6.c and **.h**	Implement the ICMP protocol for IPv6.
ip_v6.c and **.h**	Implement the IP protocol for IPv6.
mld.c and **.h**	Implement the MLD protocol.
nd.c and **.h**	Implement the Neighbor Discovery protocol.

4 Configuration Options

The following sections describe all the configuration options for the system.

4.1 IP Options

Set these configuration options in the **mip_base** package's **src/config/config_ip.h** file. The sections below list the available IP options and their default values.

General Options

IP_IFC_TASK_STACK_SIZE

The interface task stack size. The default is 1024.

IP_MULTICAST_ENABLE

Keep this at the default of 1 to enable use of multicasts. Set it to 0 to disable these.

> **Note:** IPv6 uses multicasts in place of IPv4 broadcasts; there are no broadcasts as such in IPv6.

IP_TCP_ENABLE

Keep this at the default of 1 to enable TCP. Set it to 0 to disable TCP.

IP_UDP_ENABLE

Keep this at the default of 1 to enable UDP. Set it to 0 to disable UDP.

IP_MAX_INTERFACE

The maximum number of network interfaces that can attach to the IP stack. The default is 2.

IP_MAX_TASK

The maximum number of user tasks that can simultaneously access the IP stack. The default is 16.

IP_MAX_FQDN_SIZE

The maximum Fully Qualified Domain Name (FQDN) length. This is only used if DHCP or DNS is enabled. The default is 32.

IP_IPSEC_ENABLE

Set this to 1 to enable IPsec support. The default is 0.

IP_V4_ENABLE

Keep this at the default of 1 to enable IPv4. Set it to 0 to disable IPv4.

IP_V6_ENABLE

Set this to 1 to enable IPv6. The default is 0.

IP_DHCP_ENABLE

Keep this at the default of 1 to enable DHCP. Set it to 0 to disable DHCP.

IP_DHCP_V6_ENABLE

Set this to 1 to enable DHCP for IPv6. The default is 0.

IP_V4_ALIAS_COUNT

The maximum number of IPv4 address aliases. The default is 0.

IP Options

IP_TTL_DEFAULT

The default Time to Live (TTL) value to use for sent packets. The default is 128.

IP_TOS_DEFAULT

The default Type of Service (TOS) value. The default is 0.

Checksum Options

IP_RX_CHECKSUM_ENABLE

Keep this at the default of 1 to enable checksum verification for RX packets. Set it to 0 to disable this.

IP_TX_CHECKSUM_ENABLE

Keep this at the default of 1 to enable checksum verification for TX packets. Set it to 0 to disable this.

Route Options

IP_ROUTE_ENABLE

Keep this at the default of 1 to enable routing. Set it to 0 to disable routing.

IP_ROUTE_STATIC_COUNT

The maximum size of the route table (if enabled by the previous option). The default is 4.

Defragmentation Options

IP_DEFRAG_ENABLE

Keep this at the default of 1 to enable defragmentation. Set it to 0 to disable defragmentation.

IP_MAX_DEFRAG_QUEUES

The maximum number of defragmentation queues. This limits the number of frames that can be defragmented simultaneously. The default is 4.

IP_DEFRAG_TIMEOUT

The defragmentation timeout in seconds. If this expires, the frame that was being received is discarded. The default is 1.

Default Buffer Counts

IP_DEF_MIN_BUF_COUNT

The default number of minimum size buffers. The default is 4.

IP_DEF_MTU_BUF_COUNT

The default number of Maximum Transmission Unit (MTU) size buffers. The default is 8.

4.2 ARP Options - IPv4 Only

Set these configuration options in the **ip_base_v4** package's **src/config/config_ip_arp.h** file. The available options and their default values are as follows.

ARP_DYNAMIC_COUNT

The maximum number of dynamic ARP entries. The default is 4.

ARP_STATIC_COUNT

The maximum number of static ARP entries. The default is 2.

ARP_RETRY_COUNT

The retry count for an ARP request in seconds. The default is 3.

ARP_RETRY_TIMEOUT

The retry timeout for an ARP request in seconds. The default is 5.

ARP_DEFAULT_LEASE_TIME

The default lease time in seconds. The default is 600.

ARP_LEASE_TIME_MIN

The minimum allowed lease time in seconds, used to check for a valid range at **arp_set_lease_time()**. The default is 600.

ARP_LEASE_TIME_MAX

The maximum allowed lease time in seconds, used to check for a valid range at **arp_set_lease_time()**. The default is 1200.

ARP_PROBE_ENABLE

Set this to 1 to enable the ARP Probe feature for IP v4. The options are:

- 0 (the default) – the IP address is used as soon as a link is up. In this case IP conflicts cannot be detected.
- 1 – before using a static IP address, the host checks whether the address is already in use by another device. If the IP is already in use, the interface will not enter IP_V4_STATE_CONFIGURED state and the IP_NTF_IP_ADDR_CONFLICT notification is used.

Note: ARP probing needs ARP_PROBE_RETRY_COUNT * ARP_PROBE_RETRY_TIMEOUT seconds to finish.

4.3 ICMPv4 Options - IPv4 only

Set these ICMPv4 configuration options in the **ip_base_v4** package's **src/config/config_ip_icmp.h** file. The available options and their default values are as follows.

ICMP_V4_REQUEST_TIMEOUT

The ICMPv4 timeout in seconds. If the host does not answer within this time, the ping request times out. The default value is 10.

This field is decremented at each machine which processes the datagram, so set a value at least as great as the number of gateways that the datagram will traverse.

ICMP_V4_ECHO_TABLE_SIZE

The maximum number of simultaneous ICMPv4 requests on IPv4. The default value is 2.

ICMP_V4_RX_CHECKSUM_ENABLE

Keep the default of 1 to enable checksum verification for ICMPv4 RX packets. Set the value to 0 to disable this.

ICMP_V4_TX_CHECKSUM_ENABLE

Keep the default of 1 to enable checksum verification for ICMPv4 TX packets. Set the value to 0 to disable this.

4.4 ICMPv6 Options - IPv6 only

Set these ICMPv6 configuration options in the **ip_base_v6** package's **src/config/config_ip_icmp6.h** file. The available options and their default values are as follows.

ICMP_V6_REQUEST_TIMEOUT

The ICMPv6 ping timeout in seconds. If the host does not answer within this time the ping request will timeout. The default is 10.

ICMP_V6_ECHO_TABLE_SIZE

The maximum number of simultaneous ICMPv6 ping requests on IPv6. The default is 10.

ICMP_V6_ERR_MSG_DELAY

The delay in seconds between subsequent ICMPv6 error messages. The default is 1.

ICMP_V6_RX_CHECKSUM_ENABLE

Keep the default of 1 to enable checksum verification for ICMPv6 RX packets. Set the value to 0 to disable this.

ICMP_V6_TX_CHECKSUM_ENABLE

Keep the default of 1 to enable checksum verification for ICMPv6 TX packets. Set the value to 0 to disable this.

4.5 IGMP Options - IPv4 Only

Set the IGMP - IPv4 Only options in the **ip_base_v4** package's **src/config/config_ip_igmp.h** file.

IGMP_MAX_GROUPS

The maximum number of multicast groups on all interfaces. The default is 4.

Note: The same multicast group on two different interfaces requires two separate groups.

IGMP_RX_CHECKSUM_ENABLE

The default of 1 enables checksum verification for IGMP RX packets. Set the value to 0 to disable this.

IGMP_TX_CHECKSUM_ENABLE

The default of 1 enables checksum verification for IGMP TX packets. Set the value to 0 to disable this.

4.6 MLD Options - IPv6 Only

Set these configuration options in the file **src/config/config_ip_mld.h**. For more details of the protocol, see MLD - IPv6 Only.

MLD_MULTICAST_ADDR_TBL_SIZE

The size of the table containing multicast addresses. The default is 1.

MLD_SOURCE_ADDR_TBL_SIZE

The size of the table containing multicast source addresses. The default is 2.

MLD_TIMER_PERIOD

The MLD timer period in milliseconds. The default is 100.

MLD_V1_COMPATIBILITY_MODE

MLDv2 routers may be added to a network where some hosts still use MLDv1. To be compatible with the MLDv1 hosts, MLDv2 routers MUST operate in version 1 compatibility mode.

Keep the default of 1 to enable interoperation between MLDv1 and MLDv2.

MLD_SUPRESS_MC_LISTN_REPORT_V1

An MLDv2 host may be placed on a link where there are MLDv1 hosts. The host may allow its MLDv2 Multicast Listener Report to be suppressed by a Version 1 report. Keep the default of 1 to enable this.

MLD_ROBUSTNES_CNT

The Robustness variable, the number of State Change Report retransmissions. On a lossy link, the value of this may be increased.

The maximum value is 15. The default is 2. **Do not use a value below 2.**

MLD_INIT_QUERY_INTERVAL

The initial interval between general queries sent by the Querier. The default is 125. The query interval is updated when query messages are received.

MLD_INIT_QUERY_RESP_INTERVAL

The initial query response interval, the maximum Response Delay in milliseconds used to calculate the Maximum Response Code inserted into periodic general queries. Larger values smooth the traffic as host responses are spread out over a larger interval. The default is 10000.

MLD_UNS_REPORT_INTERVAL

The Unsolicited Report Interval, the time between repetitions of a node's initial report of interest in a multicast address. The default is 1000.

4.7 ND Options - IPv6 Only

Set the Neighbor Discovery configuration options in the **ip_base_v6** package's **src/config/config_ip_nd.h** file. This section lists the available configuration options and their default values.

For more details of the protocol, see Neighbor Discovery - IPv6 Only.

ND_NEIGHBOR_CACHE_SIZE

The maximum number of neighbor cache entries. The default is 8.

ND_ROUTER_CACHE_SIZE

The maximum number of router cache entries. The default is 2.

ND_PREFIX_CACHE_SIZE

The maximum number of prefix cache entries. The default is 4.

ND_DEST_CACHE_SIZE

The maximum number of destination cache entries. The default is 8.

ND_HOP_LIMIT

The default hop limit value. This may be overwritten by router advertisement. The default is 255.

ND_DUP_ADDR_DETECT_TXS

The number of Neighbor Solicitation messages sent during auto-configuration. The default is 3.

ND_MAX_WAIT_PKTS

The maximum number of packets waiting for address resolution. The default is 3.

Note: Setting this number too high can result in packets never being released from ND's wait queue.

ND_MAX_UNICAST_SOLICIT

The default number of Neighbor Solicitation messages sent during address resolution. The default is 3.

ND_MAX_RTR_SOLICITATION_DELAY

The delay between subsequent Router Solicitation messages in seconds. The default is 1.

ND_MAX_RTR_SOLICITATIONS

The maximum number of Router Solicitations sent by the host. The default is 3.

ND_REACHABLE_TIME

The default neighbor reachability base time in seconds. (This can be overwritten by router advertisement.) The default is 30.

ND_RETRANS_TIMER

The default neighbor solicitation retransmit time in seconds. The default is 1.

ND_ROUTER_LIFE_TIME

The default router lifetime in seconds. The default is 9000.

4.8 Network Driver Memory Pool Options

Set these configuration options in the **mip_base** package's **src/config/config_ip_pool.h** file. The available options and their default values are as follows.

IP_MAX_POOLS

The maximum number of pools in the system. The default is IP_MAX_INTERFACE.

IP_POOL_MAX_QUEUES

The maximum number of queues in a pool. The default is 3.

5 Application Programming Interface

This section documents the Application Programming Interface (API). It includes all the functions that are available to an application program.

The module provides most of the API functions provided by the TCP and UDP modules, described in the relevant manuals. Some of the IP stack functions are also provided.

5.1 Module Management

These functions initialize, start, stop, and delete the IP stack.

The function calls are the same for IPv4 and IPv6, but when IPv6 is enabled the functions also initialize/start/stop/delete the ND and MLD modules.

ip_stack_init

Use this function to initialize the IP stack and the interface, IP, ROUTE, TCP, and UDP modules.

When IPv4 is enabled, this call also initializes the ARP and ICMP modules. When IPv6 is enabled, this call also initializes the ND and MLD modules.

Note: Call this function first.

Format

```
t_ip_ret ip_stack_init ( void )
```

Arguments

None.

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_OS	OS resource creation error.
Else	See Error Codes.

ip_stack_start

Use this function to start the IP stack and the interface, IP, ROUTE, TCP, and UDP modules.

When IPv4 is enabled, this call also starts the ARP and ICMP modules. When IPv6 is enabled, this call also starts the ND and MLD modules.

Note: Call **ip_stack_init()** before this function.

Format

```
t_ip_ret ip_stack_start ( void )
```

Arguments

None.

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

ip_stack_stop

Use this function to stop the IP stack and the interface, IP, ROUTE, TCP, and UDP modules.

When IPv4 is enabled, this call also stops the ARP and ICMP modules. When IPv6 is enabled, this call also stops the ND and MLD modules.

Format

```
t_ip_ret ip_stack_stop ( void )
```

Arguments

None.

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

ip_stack_delete

Use this function to delete the IP stack and the interface, IP, ROUTE, TCP, and UDP modules.

When IPv4 is enabled, this call also deletes the ARP and ICMP modules. When IPv6 is enabled, this call also deletes the ND and MLD modules.

Format

```
t_ip_ret ip_stack_delete ( void )
```

Arguments

None.

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_OS	OS resource creation error.
IP_ERROR	An unspecified error occurred.
Else	See Error Codes.

5.2 IP Functions

These functions work the same way for both IPv4 and IPv6 addresses.

Note: This section documents the complete API of the IP stack. Normally you will use either the Sockets interface or the native TCP or UDP interfaces for communication. This means you should not need to use this API except to manage the IP module; that is, to initialize, start, stop, and delete the IP-related modules.

Function	Description
ip_enter_task()	Allows the caller task to call IP stack functions.
ip_exit_task()	Exits the caller task.
ip_addr_to_str()	Converts an IP address to a string.
ip_str_to_addr()	Converts a string to an IP address.
ip_get_config()	Gets the IP configuration of an interface.
ip_set_config()	Sets the IP configuration of an interface.
ip_get_fqdn()	Gets the Fully Qualified Domain Name (FQDN) of this host.
ip_get_default_fqdn()	Gets the default value of the FQDN of this host.
ip_set_default_fqdn()	Sets the FQDN of this host.
ip_ifc_get_mtu()	Gets the Maximum Transmission Unit (MTU) size of an interface.
ip_register_config_ntf()	Registers a notification if IP changes between a configured and not configured state.

ip_enter_task

Use this function to allow the caller task to call IP stack functions.

> **Note:** All tasks using the network stack **must** have their priority set lower than OAL_HIGHEST_PRIORITY. You must ensure that this is always true, even when the task has its priority increased in a "priority inheritance" situation.

Format

```
t_ip_ret ip_enter_task ( void )
```

Arguments

None.

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_MORE_ENTRY	More tasks are trying to access the stack than the system is configured for. IP_MAX_TASK has been exceeded.
Else	See Error Codes.

Copyright HCC Embedded 2018

ip_exit_task

Use this function to exit the caller task.

After this the caller task cannot call any functions in the IP stack.

Format

```
t_ip_ret ip_exit_task ( void )
```

Arguments

None.

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_TASK_NOT_FOUND	The task was not found.
Else	See Error Codes.

ip_addr_to_str

Use this function to convert an IP address to a string.

Format

```
t_ip_ret ip_addr_to_str(
    const t_ip_addr * const   p_ip_addr,
    char_t                    str[],
    const uint8_t             str_len )
```

Arguments

Argument	Description	Type
p_ip_addr	A pointer to the IP address to convert.	t_ip_addr *
str[]	The string to receive the converted address.	char_t
str_len	The maximum length of the string.	uint8_t

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_SIZE	The string is too small.
IP_ERR_INVALID_PARAM	The IP address is invalid.
Else	See Error Codes.

Copyright HCC Embedded 2018

ip_str_to_addr

Use this function to convert a string to an IP address.

Format

```
t_ip_ret ip_str_to_addr (
    const char_t         str[],
    t_ip_addr * const    p_ip_addr )
```

Arguments

Argument	Description	Type
str[]	The string to convert.	char_t
p_ip_addr	Where to write the IP address.	t_ip_addr *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_PARAM	The string is invalid.
Else	See Error Codes.

ip_get_config

Use this function to get the IP configuration of an interface.

Format

```
t_ip_ret ip_get_config (
    const t_ip_ifc_hdl    ifc_hdl,
    t_ip_config * const   p_ip_config )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
p_ip_config	Where to write the current configuration.	t_ip_config *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

ip_set_config

Use this function to set the IP configuration of an interface.

Note: This function has no effect if the target network interface has been enabled with DHCP.

Format

```
t_ip_ret ip_set_config (
    const t_ip_ifc_hdl    ifc_hdl,
    t_ip_config * const   p_ip_config,
    const uint8_t         flag )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
p_ip_config	A pointer to the new configuration.	t_ip_config *
flag	A flag indicating the configuration to set: IP_CONFIGURE_IP_V4 or IP_CONFIGURE_IP_V6.	uint8_t

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

Example

The following example shows how to set the IP configuration of an interface.

```
void ip_config_ntf ( const t_ip_ifc_hdl ifc_hdl
                     , const uint32_t ntf
                     , const t_ip_config * p_ip_config
                     , const char_t * const p_fqdn )
{
    char_t str[40];
    t_ip_addr  ip_addr;

    if ( ifc_hdl == eth0_ifc_hdl )
    {
        printf( "ETH 0:" );

    if ( ntf == IP_NTF_CONFIG_CHANGE )
    {
        printf( " Configuration changed." );
    }
    else
    {
        printf( " Configuration state has changed." );
    }

    switch ( p_ip_config->ipc_state & IP_STATE_LINK_MASK )
    {
        case IP_STATE_STOPPED:
            printf( "\r\nINTERFACE IS STOPPED" );
            break;

        case IP_STATE_LINK_DOWN:
            printf( "\r\nINTERFACE IS LINKED DOWN" );
            break;

        case IP_STATE_LINK_UP:
            printf( "\r\nINTERFACE IS LINKED UP ");
            break;

        default:
            printf( "\r\nUNKNOW STATE" );
            break;
    } /* switch */
#if IP_V4_ENABLE
    if ( ( p_ip_config->ipc_state & IP_V4_STATE_CONFIGURED ) != 0U )
    {
        ip_addr_to_str( &(p_ip_config->ipc_ipaddr), str, 40);
        printf( "\r\nIP address : ");
        printf((char *)str);

        ip_addr.ipa_version = IPV_IP_V4;I would
        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_netmask, 4U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( "\r\nNetmask    : ");
        printf((char *)str);

        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_gateway, 4U );
        ip_addr_to_str( &(ip_addr), str, 40);
```

```
        printf( "\r\nGateway     : ");
        printf((char *)str);

        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_dns1_addr, 4U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( "\r\nDNS1        : ");
        printf((char *)str);

        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_dns2_addr, 4U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( "\r\nDNS2        : ");
        printf((char *)str);

        printf( "\r\nFQDN        :  ");
        printf((char *)p_fqdn );

    }
#endif
#if IP_V6_ENABLE
    if ( ( p_ip_config->ipc_state & IP_V6_STATE_CONFIGURED ) != 0U )
    {
        printf( "\r\nIP6 address : " );
        ip_addr_to_str( &(p_ip_config->ipc_ip_v6_addr), str, 40);
        printf( str );

        printf( "\r\nIP6 DNS1 address : " );
        ip_addr.ipa_version = IPV_IP_V6;
        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_ip_v6_dns1, 16U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( str );

        printf( "\r\nIP6 DNS2 address : " );
        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_ip_v6_dns2, 16U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( str );
    }
#endif
    printf( "\r\n" );
    }
} /* ip_config_ntf */
```

ip_get_fqdn

Use this function to get the Fully Qualified Domain Name (FQDN) of this host.

Format

```
t_ip_ret ip_get_fqdn (
    const t_ip_ifc_hdl    ifc_hdl,
    char_t                fqdn[] )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
fqdn[]	Where to write the FQDN.	char_t

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

ip_get_default_fqdn

Use this function to get the default value of the Fully Qualified Domain Name (FQDN) of this host.

Format

```
t_ip_ret ip_get_default_fqdn (
    const t_ip_ifc_hdl   ifc_hdl,
    char_t               default_fqdn[] )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
default_fqdn[]	Where to write the default FQDN.	char_t

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

ip_set_default_fqdn

Use this function to set the default Fully Qualified Domain Name (FQDN) of this host.

If DHCP is enabled and the server supports FQDN, this name is requested from the server (if it has been set). The default FQDN is used in all DHCP requests. That is, if the host connects to another network with a different DHCP server, an earlier assigned FQDN name does not influence the new request.

Format

```
t_ip_ret ip_set_default_fqdn (
    const t_ip_ifc_hdl    ifc_hdl,
    const char_t          default_fqdn[] )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
default_fqdn[]	The default FQDN.	char_t

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_SIZE	Invalid *default_fqdn[]* size.
Else	See Error Codes.

ip_ifc_get_mtu

Use this function to get the Maximum Transmission Unit (MTU) size of an interface.

Format

```
t_ip_ret ip_ifc_get_mtu  (
    const t_ip_ifc_hdl   ifc_hdl,
    uint16_t * const     p_mtu_size )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
p_mtu_size	Where to write the MTU size.	uint16_t *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_HDL	The handle is invalid.
Else	See Error Codes.

ip_register_config_ntf

Use this function to register a notification if IP changes between a configured and not configured state.

This can be useful if DHCP is enabled, to determine when the IP address was assigned (or whether it is not available due to disconnection/stop).

Format

```
void ip_register_config_ntf ( t_ip_config_ntf_fn ip_config_ntf_fn )
```

Arguments

Argument	Description	Type
ip_config_ntf_fn	The configured state notification.	t_ip_config_ntf_fn

Return Values

None.

5.3 IP Functions - IPv4 only

These functions are for IPv4 addresses only.

Function	Description
ip_v4_set_alias()	Sets an interface's IP alias configuration.
ip_v4_delete_alias()	Deletes an interface's IP alias configuration.

Copyright HCC Embedded 2018

ip_v4_set_alias

Use this function to set an interface's IP alias configuration.

Format

```
t_ip_ret ip_v4_set_alias (
    const t_ip_ifc_hdl   ifc_hdl,
    uint8_t              ip_addr[IP_V4_ADDR_SIZE],
    uint8_t              netmask[IP_V4_ADDR_SIZE] )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
ip_addr[IP_V4_ADDR_SIZE]	A pointer to the new IP address.	uint8_t
netmask[IP_V4_ADDR_SIZE]	A pointer to the network mask to use with the new IP address.	uint8_t

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_ALIAS_SIZE	There is not enough room for the IP alias; increase the value of configuration option IP_V4_ALIAS_COUNT.
IP_ERR_ALIAS_CONFIGURED	There is already an IP alias configured with the given IP address.
Else	See Error Codes.

ip_v4_delete_alias

Use this function to delete an interface's IP alias configuration.

Format

```
t_ip_ret ip_v4_delete_alias (
    const t_ip_ifc_hdl   ifc_hdl,
    uint8_t              ip_addr[IP_V4_ADDR_SIZE] )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
ip_addr[IP_V4_ADDR_SIZE]	A pointer to the new IP address.	uint8_t

Return Values

Code	Description
IP_SUCCESS	Successful execution; the alias configuration was deleted.
IP_ERROR	The alias configuration was not found.

5.4 IP Function - IPv6 only

This function is for IPv6 addresses only.

Function	Description
ip_v6_get_host_addr()	Gets the local IP address.

ip_v6_get_host_addr - IPv6 only

Use this function to get the local IP address.

This function is needed because **ip_get_config()** can retrieve only the IPv6 Link local address. An IPv6 interface can have multiple addresses assigned.

Format

```
t_ip_ret ip_v6_get_host_addr (
    const t_ip_ifc_hdl    ifc_hdl,
    uint16_t              idx,
    uint8_t * const       p_addr )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
idx	0-based index of the address to return. Index 0 is the link local address. Index 1..n are SLAC addresses when DHCPv6 is disabled; otherwise, the DHCP address is returned.	uint16_t
p_addr	Where to write the IPv6 address of the host.	uint8_t *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_MORE_ENTRY	More tasks are trying to access the stack than the system is configured for; IP_MAX_TASK has been exceeded.

5.5 ARP Functions - IPv4 Only

This section documents the Address Resolution Protocol (ARP) API functions.

Note: In IPv6 the ARP has been replaced by Neighbor Discovery.

The ARP functions are the following.

Function	Description
arp_add_static_entry()	Adds a new static entry to the ARP table, or updates an existing entry.
arp_del_entry()	Deletes an entry from the ARP table.
arp_set_lease_time()	Sets the ARP table lease time in seconds.
arp_find_first_entry()	Finds the first valid entry in the ARP table.
arp_find_next_entry()	Finds the next valid entry in the ARP table.
arp_get_hwaddr_from_ipaddr()	Gets the hardware address corresponding to an IP address in the ARP table.
arp_get_ipaddr_from_hwaddr()	Gets the IP address corresponding to a hardware address in the ARP table.

arp_add_static_entry

Use this function to add a new static entry to the ARP table, or to update an existing entry.

Format

```
t_ip_ret arp_add_static_entry (
    const t_ip_addr *       p_ip_addr,
    const uint8_t * const   p_hw_addr )
```

Arguments

Argument	Description	Type
p_ip_addr	A pointer to the IP address of the entry to add/update.	t_ip_addr *
p_hw_addr	A pointer to the hardware address.	uint8_t *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

arp_del_entry

Use this function to delete an entry from the ARP table.

Format

```
t_ip_ret arp_del_entry ( const t_ip_addr * p_ip_addr )
```

Arguments

Argument	Description	Type
p_ip_addr	A pointer to the IP address of the entry to delete.	t_ip_addr *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

arp_set_lease_time

Use this function to set the ARP table lease time in seconds.

The lease time is the dynamic entry expiry time.

Format

```
void arp_set_lease_time ( const uint16_t sec )
```

Arguments

Argument	Description	Type
sec	The ARP lease time in seconds.	uint16_t

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_PARAM	The parameter is invalid.
Else	See Error Codes.

arp_find_first_entry

Use this function to find the first valid entry in the ARP table.

Format

```
t_ip_ret arp_find_first_entry ( t_arp_find * const p_arpf )
```

Arguments

Argument	Description	Type
p_arpf	Where to write the entry's data.	t_arp_find *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_MORE_ENTRY	There are no valid entries in the table.

arp_find_next_entry

Use this function to find the next valid entry in the ARP table.

Note: You must call **arp_find_first_entry()** before this function.

Format

```
t_ip_ret arp_find_next_entry ( t_arp_find * const p_arpf )
```

Arguments

Argument	Description	Type
p_arpf	Where to write the next entry's data.	t_arp_find *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_MORE_ENTRY	There are no more valid entries in the table.

Copyright HCC Embedded 2018

arp_get_hwaddr_from_ipaddr

Use this function to get the hardware address corresponding to an IP address in the ARP table.

Format

```
t_ip_ret arp_get_hwaddr_from_ipaddr (
    const t_ip_addr *    p_ip_addr,
    uint8_t * const      p_hw_addr )
```

Arguments

Argument	Description	Type
p_ip_addr	A pointer to the IP address to search for.	t_ip_addr *
p_hw_addr	Where to write the hardware address.	uint8_t *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

arp_get_ipaddr_from_hwaddr

Use this function to get the IP address corresponding to a hardware address in the ARP table.

Format

```
t_ip_ret arp_get_ipaddr_from_hwaddr (
    const uint8_t * const   p_hw_addr,
    t_ip_addr * const       p_ip_addr )
```

Arguments

Argument	Description	Type
p_hw_addr	A pointer to the hardware address to search for.	uint8_t *
p_ip_addr	Where to write the IP address.	t_ip_addr *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

5.6 ICMP Functions

This section documents the Internet Control Message Protocol (ICMP) API functions.

Function	Description
icmp_ping()	Starts an ICMP echo request.
icmp_ping_state()	Checks the status of an echo request sent by using **icmp_ping()**.
icmp_ping_cb()	Starts an ICMP echo request with timeout. (Currently only supports IPv4 addresses.)

icmp_ping

Use this function to start an echo request. The result can be polled using **icmp_ping_state()**.

Format

```
t_ip_ret icmp_ping (
    const t_ip_addr *     p_dst_ip_addr,
    t_ping_hdl * const    p_ping_hdl )
```

Arguments

Argument	Description	Type
p_dst_ip_addr	A pointer to the destination IP address.	t_ip_addr *
p_ping_hdl	Where to write the ping handle.	t_ping_hdl *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_MORE_ENTRY	No more entries (ICMP_ECHO_TABLE_SIZE exceeded).
Else	See Error Codes.

icmp_ping_state

Use this function to check the status of an echo request sent by using **icmp_ping()**.

Format

```
t_ip_ret icmp_ping_state (
    const t_ping_hdl       ping_hdl,
    t_ping_state * const   p_ping_state )
```

Arguments

Argument	Description	Type
ping_hdl	The ping handle that identifies the echo request.	t_ping_hdl
p_ping_state	Where to write the state of the ping.	t_ping_state *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_HDL	The handle is not valid.
Else	See Error Codes.

icmp_ping_cb

Use this callback function to start an ICMP echo request with timeout.

The function is called when the result is ready.

> **Note:**
> - This function currently only supports IPv4 addresses.
> - It is the user's responsibility to provide this callback function. Providing this function is optional.

Format

```
t_ip_ret icmp_ping_cb (
    const t_ip_addr *    p_dst_ip_addr,
    uint16_t             ping_timeout,
    t_ping_cb_fn         p_ping_cb_fn,
    uint32_t             ping_cb_param )
```

Arguments

Argument	Description	Type
p_dst_ip_addr	A pointer to the destination IP address.	t_ip_addr *
ping_timeout	The echo request timeout in seconds.	uint16_t
p_ping_cb_fn	A pointer to the callback function.	t_ping_cb_fn
ping_cb_param	The parameter for the callback function.	uint32_t

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_MORE_ENTRY	No more entries (ICMP_ECHO_TABLE_SIZE exceeded).
Else	See Error Codes.

5.7 IGMP Functions - IPv4 Only

This section documents the Internet Gateway Message Protocol (IGMP) API functions.

Function	Description
igmp_add_membership()	Makes an interface a member of a multicast group.
igmp_drop_membership()	Ends an interface's membership of a multicast group.

Copyright HCC Embedded 2018

igmp_add_membership

Use this function to make an interface a member of a multicast group.

Format

```
t_ip_ret igmp_add_membership (
    const t_ip_ifc_hdl        ifc_hdl,
    const t_ip_addr * const   p_group_address )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
p_group_address	A pointer to the group address.	t_ip_addr *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_PARAM	A parameter is invalid.
IP_ERR_NO_MORE_ENTRY	There are no more free entries (the number available is given by IGMP_MAX_GROUPS).
Else	See Error Codes.

igmp_drop_membership

Use this function to drop membership of a multicast group.

Format

```
t_ip_ret igmp_drop_membership (
    const t_ip_ifc_hdl        ifc_hdl,
    const t_ip_addr * const   p_group_address )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
p_group_address	A pointer to the group address.	t_ip_addr *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_PARAM	A parameter is invalid.
Else	See Error Codes.

5.8 MLD Function - IPv6 only

There is a single Multicast Listener Discovery (MLD) function, **mld_set_mc_listen()**.

mld_set_mc_listen

Use this function to set the multicast listener on an interface.

To start listening on a multicast address from all source addresses, configure the mode in the *t_mld_listen* structure to MLD_MODE_EXCLUDE and the number of source addresses to 0.

To stop listening on multicast addresses, change the mode to MLD_MODE_INCLUDE and the number of source addresses to 0.

Format

```
t_ip_ret mld_set_mc_listen (
    t_ip_ifc_hdl          ifc_hdl,
    const t_mld_listen *  p_listen )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
p_listen	A pointer to the structure describing the multicast listener.	t_mld_listen *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_MORE_ENTRY	There is no free entry for the multicast address.
Else	See Error Codes.

Example

```
const uint8_t g_mc_addr[16U] =
{
    0xFFU, 0x02, 0x0U, 0x0U, 0x0U, 0x0U, 0x0U, 0x0U,
    0x0U,  0x0U, 0x0U, 0x5U, 0x4U, 0x3U, 0x2U, 0x1U
};

const uint8_t  src_addr1[] =
{
    0xfeU, 0x80U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x69U, 0x8eU, 0xe8U, 0x1eU, 0x67U, 0x87U, 0x31U, 0x24U
};

const uint8_t  src_addr2[] =
{
    0xfeU, 0x80U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x69U, 0x8eU, 0xe8U, 0x1eU, 0x67U, 0x87U, 0x31U, 0x01U
};

const uint8_t  src_addr3[] =
{
    0xfeU, 0x80U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U, 0x00U,
    0x69U, 0x8eU, 0xe8U, 0x1eU, 0x67U, 0x87U, 0x31U, 0x02U
};

const uint8_t  * src_list[3] =
{
    src_addr1
    , src_addr2
    , src_addr3
};

t_ip_ret configure_multicast( void )
{
    t_mld_listen list;

    list.p_mldl_mc_addr = g_mc_addr;   /* Set multicast address to listen to */
    list.mldl_mode = MLD_MODE_EXCLUDE; /* Set mode to exclude given source addresses */
    list.mldl_src_count = 3U;          /* Set number of source addresses */
    list.p_mldl_src_list = src_list;   /* Set source addresses that will be rejected */

    return mld_set_mc_listen( eth0_ifc_hdl, &list );
}
```

5.9 Network Driver Memory Pool Functions

Note: Only use this section if you who want to do either of the following:

- Write a network driver.
- Externally assign a memory pool to a network driver.

A network driver can obtain a memory pool in two ways:

- By using a dedicated memory pool you assign to it during development.
- By using a memory pool dynamically assigned to the network driver at run time before the driver is started.

The functions are as follows:

Function	Description
ip_nwd_get_buf()	Gets the network driver a frame of a specified length.
ip_nwd_get_buf_ext()	Requests a frame for the network driver from a specific pool.
ip_nwd_release_buf()	Releases a frame obtained by using one of the above functions.
ip_nwd_release_buf_ext()	Releases a frame obtained by using one of the above functions.
ip_pool_buf_config()	Sets a pool's buffer configuration.
ip_pool_add()	Adds a new pool to the system.
ip_pool_del()	Deletes a pool from the system.
ip_pool_get_prop()	Gets the address and size of the buffer the pool is using.
ip_ifc_get_pool()	Gets the pool handle of an interface.
ip_ifc_set_pool()	Sets the pool handle of an interface.

Copyright HCC Embedded 2018

ip_nwd_get_buf

Use this function to get the network driver a frame of a specified length.

This function can be used if called from a network driver protected function.

Note: It is the responsibility of the caller to provide mutex protection.

Format

```
t_ip_ret ip_nwd_get_buf (
    const uint32_t              param,
    const uint16_t              req_len,
    t_ip_get_buf_tn * const     p_get_buf_tn,
    uint8_t * * const           pp_frame )
```

Arguments

Argument	Description	Type
param	The parameter (the interface index) received when **p_nwfn_start()** was called by the IP stack.	uint32_t
req_len	The requested buffer length.	uint16_t
p_get_buf_tn	A pointer to the structure holding the timeout and/or notification used if a buffer is not immediately available. Note that you may call **ip_nwd_get_buf()** from the notification.	t_ip_get_buf_tn *
pp_frame	Where to write the pointer to the frame.	uint8_t * *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_REQUEST	The task was suspended due to an event wait and the interface was stopped or changed in the meantime.
Else	See Error Codes.

ip_nwd_get_buf_ext

Use this function to request a frame for the network driver from a specific pool.

Format

```
t_ip_ret ip_nwd_get_buf_ext (
    const uint32_t              param,
    const uint16_t              req_len,
    t_ip_get_buf_tn * const     p_get_buf_tn,
    uint8_t * * const           pp_frame )
```

Arguments

Argument	Description	Type
param	The parameter received when **p_nwfn_start()** was called by the IP stack	uint32_t
req_len	The requested buffer length.	uint16_t
p_get_buf_tn	A pointer to the structure holding the timeout and/or notification used if a buffer is not immediately available. Note that you may call **ip_nwd_get_buf()** from the notification.	t_ip_get_buf_tn *
pp_frame	Where to write the pointer to the frame.	uint8_t * *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_REQUEST	The task was suspended due to an event wait and the interface was stopped or changed in the meantime.
Else	See Error Codes.

ip_nwd_release_buf

Use this function to release a frame obtained by using **ip_nwd_get_buf()** or **ip_nwd_get_buf_ext()**.

This function can be used if called from a network driver protected function.

Use this if you allocated a receive buffer with **ip_nwd_get_buf()** or **ip_nwd_get_buf_ext()** but, due to an error condition (for example, the frame received was corrupt), you want to release the buffer and return the frame to the pool.

Note:

- Providing mutex protection is the caller's responsibility.
- Never call this after the **p_nwfn_stop()** or **p_nwfn_delete()** functions as when these are called the IP stack automatically returns all buffers allocated by the network driver back to the specific pool.

Format

```
void ip_nwd_release_buf (
    const uint32_t    param,
    uint8_t * const   p_frame )
```

Arguments

Argument	Description	Type
param	The parameter (the interface index).	uint32_t
p_frame	A pointer to the frame.	uint8_t *

Return Values

None.

ip_nwd_release_buf_ext

Use this function to release a frame obtained by using **ip_nwd_get_buf()** or **ip_nwd_get_buf_ext()**.

This function can be used if called from a network driver protected function.

Use this if you allocated a receive buffer with **ip_nwd_get_buf()** or **ip_nwd_get_buf_ext()** but, due to an error condition (for example, the frame received was corrupt), you want to release the buffer and return the frame to the pool.

Note:

- Providing mutex protection is the caller's responsibility.
- Never call this after the **p_nwfn_stop()** or **p_nwfn_delete()** functions as when these are called the IP stack automatically returns all buffers allocated by the network driver back to the specific pool.

Format

```
void ip_nwd_release_buf_ext (
    const uint32_t    param,
    uint8_t * const   p_frame )
```

Arguments

Argument	Description	Type
param	The parameter (the interface index).	uint32_t
p_frame	A pointer to the frame.	uint8_t *

Return Values

None.

ip_pool_buf_config

Use this function to set a pool's buffer configuration.

Format

```
t_ip_ret ip_pool_buf_config (
    const t_ip_pool_hdl     pool_hdl,
    const t_ip_pool_qprop   pool_qprop[],
    const uint8_t           cnt )
```

Arguments

Argument	Description	Type
pool_hdl	The pool handle.	t_ip_pool_hdl
pool_qprop[]	An array of queue properties.	t_ip_pool_qprop
cnt	The number of elements in *pool_qprop*. This must not be greater than IP_MAX_POOLS in the file **config_ip_pool.h**.	uint8_t

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_CONFIG	One of the following: • The number of queues in the *pool_qprop* array exceeds IP_POOL_MAX_QUEUES. • The queue sizes are not in order. • The maximum queue buffer size is less than required for the pool.
IP_ERR_INVALID_STATE	The pool is active; you cannot reconfigure it.
Else	See Error Codes.

ip_pool_add

Use this function to add a new pool to the system.

Format

```
t_ip_ret ip_pool_add (
    uint8_t * const           p_buf,
    const uint32_t            buf_size,
    t_ip_pool_hdl * const     p_pool_hdl )
```

Arguments

Argument	Description	Type
p_buf	A pointer to the buffer.	uint8_t *
buf_size	The buffer size.	uint32_t
p_pool_hdl	Where to write the pool handle.	t_ip_pool_hdl *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_CONFIG	Invalid configuration, one of the following: • The pool is active. • The new buffer overlaps with the old buffer but its beginning or size is different.
IP_ERR_NO_MORE_ENTRY	The pool table is full; IP_MAX_POOLS was exceeded.
Else	See Error Codes.

Copyright HCC Embedded 2018

ip_pool_del

Use this function to delete a pool from the system.

This function only succeeds if the pool was added with **ip_pool_add()** and no interfaces are using it.

Format

```
t_ip_ret ip_pool_del ( const t_ip_pool_hdl pool_hdl )
```

Arguments

Argument	Description	Type
pool_hdl	The pool handle.	t_ip_pool_hdl

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_POOL_BUSY	The pool is busy so cannot be deleted. This happens if one or more buffers have not been released from the pool.
Else	See Error Codes.

ip_pool_get_prop

Use this function to get the address and size of the buffer the pool is using.

Format

```
t_ip_ret ip_pool_get_prop (
    const t_ip_pool_hdl    pool_hdl,
    uint8_t * * const      pp_buf,
    uint32_t * const       p_buf_size )
```

Arguments

Argument	Description	Type
pool_hdl	The pool handle.	t_ip_pool_hdl
pp_buf	Where to write the pointer to the buffer.	uint8_t * *
p_buf_size	Where to write the size of the buffer.	uint32_t *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

ip_ifc_get_pool

Use this function to get the pool handle of an interface.

Format

```
t_ip_ret ip_ifc_get_pool (
    const t_ip_ifc_hdl      ifc_hdl,
    t_ip_pool_hdl * const   p_pool_hdl )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
p_pool_hdl	Where to write the pool handle.	t_ip_pool_hdl *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

ip_ifc_set_pool

Use this function to set the pool handle of an interface.

Format

```
t_ip_ret ip_ifc_set_pool (
    const t_ip_ifc_hdl    ifc_hdl,
    const t_ip_pool_hdl   pool_hdl )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
pool_hdl	The pool handle.	t_ip_pool_hdl

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_POOL	The interface already has an assigned pool, or the pool handle is not valid.
Else	See Error Codes.

5.10 Routing Functions

This section documents the routing API functions. These are defined in the file **api_ip.h**.

Function	Description
ip_route_add()	Adds a static interface to the route table.
ip_route_del()	Deletes a static route entry.
ip_route_get()	Obtains all valid entries in the route table.
ip_route_get_hdl()	Gets the route handle, based on the destination IP address.
ip_route_get_hdl_fqdn()	Searches for the route handle, based on the domain name in the Fully Qualified Domain Name (FQDN).
ip_route_get_config()	Gets the IP configuration from the route handle.

ip_route_add

Use this function to add a static interface to the route table. The maximum number of static interfaces is defined by IP_ROUTE_STATIC_COUNT.

If the interface IP address is changed or deleted, the added static routes belonging to the interface are dropped automatically.

Format

```
t_ip_ret ip_route_add (
   const t_ip_addr * const    p_nwaddr,
   const t_ip_addr * const    p_netmask,
   const t_ip_addr * const    p_gateway,
   t_ip_route_hdl * const     p_route_hdl )
```

Arguments

Argument	Description	Type
p_nwaddr	A pointer to the network address.	t_ip_addr *
p_netmask	A pointer to the netmask.	t_ip_addr *
p_gateway	A pointer to the gateway address.	t_ip_addr *
p_route_hdl	Where to write the route handle. After a successful call, this handle can be used to access the new route.	t_ip_route_hdl *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_ROUTE	The specified route is not working. The route may already be in the table, or the gateway does not belong to a single interface. (That is, it belongs to no interfaces or more than one interface).
IP_ERR_NO_MORE_ENTRY	There are no more free entries in the route table. Increase the size of IP_ROUTE_STATIC_COUNT.
Else	See Error Codes.

ip_route_del

Use this function to delete a static route entry.

Note: Only routes added by **ip_route_add()** can be deleted with this function.

Format

```
t_ip_ret ip_route_del ( const t_ip_route_hdl route_hdl )
```

Arguments

Argument	Description	Type
route_hdl	The route handle of the route to delete.	t_ip_route_hdl

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_HDL	The handle is not valid.
Else	See Error Codes.

ip_route_get

Use this function to obtain all valid entries in the route table.

If an entry has an *ipr_nwaddr* of 0 it is the default gateway entry.

You can use the function **ip_route_get_config()** to obtain the IP configuration parameters the route entry belongs to.

Format

```
t_ip_ret ip_route_get (
    const uint8_t                b_first,
    t_ip_route_get_hdl * const   p_route_get_hdl,
    t_ip_route * const           p_ip_route )
```

Arguments

Argument	Description	Type
b_first	Set this TRUE to reset *route_get_hdl* and return the first valid entry.	uint8_t
p_route_get_hdl	A pointer to the get handle.	t_ip_route_get_hdl *
p_ip_route	Where to write the route.	t_ip_route *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_MORE_ENTRY	No more entries are available.
Else	See Error Codes.

ip_route_get_hdl

Use this function to get the route handle, based on the destination IP address.

Format

```
t_ip_ret ip_route_get_hdl (
    const t_ip_addr * const    p_dst_ip_addr,
    t_ip_route_hdl * const     p_route_hdl )
```

Arguments

Argument	Description	Type
p_dst_ip_addr	A pointer to the destination IP address.	t_ip_addr *
p_route_hdl	Where to write the route handle.	t_ip_route_hdl *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

Copyright HCC Embedded 2018

ip_route_get_hdl_fqdn

Use this function to search for the route handle, based on the domain name in the Fully Qualified Domain Name (FQDN).

Format

```
t_ip_ret ip_route_get_hdl_fqdn (
    const char_t            fqdn[],
    t_ip_route_hdl * const   p_route_hdl )
```

Arguments

Name	Description	Type
fqdn[]	The FQDN.	char_t
p_route_hdl	Where to write the route handle.	t_ip_route_hdl *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
IP_ERR_ROUTE	The route was not found.
Else	See Error Codes.

ip_route_get_config

Use this function to get the IP configuration from the route handle.

Format

```
t_ip_ret ip_route_get_config (
    const t_ip_route_hdl   route_hdl,
    t_ip_config * const    p_ip_cfg )
```

Arguments

Argument	Description	Type
route_hdl	The route handle.	t_ip_route_hdl
p_ip_cfg	Where to write the IP configuration.	t_ip_config *

Return Values

Code	Description
IP_SUCCESS	Successful execution.
Else	See Error Codes.

5.11 Error Codes

If a function executes successfully, it returns with IP_SUCCESS, a value of 0. This table show the meaning of the IP return codes:

Error code	Value	Description
IP_SUCCESS	0	Successful execution.
IP_MORE_DATA	1	There is more data pending.
IP_CONN_PENDING	2	The connection was initiated but is not complete.
IP_DISCONNECT_WAIT	3	Waiting for disconnection to complete.
IP_ERROR	4	An unspecified error occurred.
IP_ERR_OS	5	OS resource creation error.
IP_ERR_INIT	6	Initialization error.
IP_ERR_NWDRV	7	Network driver error.
IP_ERR_NOT_READY	8	The connection is not ready.
IP_ERR_NOT_CONFIGURED	9	Incompatible with the current configuration.
IP_ERR_NO_DATA	10	No data is available.
IP_ERR_NO_MORE_ENTRY	11	More tasks are trying to access the stack than the system is configured for; IP_MAX_TASK has been exceeded.
IP_ERR_NO_CONNECTION	12	This connection does not exist.
IP_ERR_NO_BUFFER	13	Function **tcp_get_buf()** failed to allocate a buffer due to an empty buffer pool.
IP_ERR_INVALID_PARAM	14	There is an invalid parameter in the requested operation.
IP_ERR_INVALID_HDL	15	An invalid handle was used in the requested operation.
IP_ERR_INVALID_FRAME	16	An invalid frame was received.
IP_ERR_INVALID_PROTOCOL	17	An IP frame with an invalid protocol identifier was received.
IP_ERR_INVALID_SIZE	18	There is an error in the size field of the received IP frame.

Error code	Value	Description
IP_ERR_INVALID_REQUEST	19	An invalid operation was requested.
IP_ERR_INVALID_STATE	20	An invalid state has been reached.
IP_ERR_INVALID_CONFIG	21	One of the following: • An interface was started and the interface has an associated pool with an invalid configuration. • An interface was added which wants to use an active pool with network driver parameters that are incompatible with it. • Pool properties were manually configured but the maximum required buffer size is smaller than that requested by the added interface. • **ip_pool_buf_config()** was called with invalid queue properties. This occurs if the number of elements in the queue properties exceeds IP_POOL_MAX_QUEUES, the queue buffer sizes are not in ascending order, or the maximum buffer size is less than required by an interface initialized earlier. • A pool was added with **ip_pool_add()** but the buffer overlaps with an existing active pool buffer and the new buffer size or start address is different.
IP_ERR_INVALID_BUFFER	22	An interface was started but the required pool buffer queue properties cannot be applied for the pool.
IP_ERR_INVALID_POOL	23	An interface was started without an associated pool.
IP_ERR_POOL_BUSY	24	A pool cannot be deleted because the system did not release all buffers in the active pool. This can only happen if the user has obtained frame buffers by using **tcp_get_buf()** and these have not been released yet by using **tcp_release_buf()**.
IP_ERR_ROUTE	25	The specified route is not working.
IP_ERR_LINK_DOWN	26	The physical link requested is down.
IP_ERR_PORT_OPENED	27	The requested port is already open.
IP_ERR_BAD_CHECKSUM	28	A frame with a checksum error was received.
IP_ERR_DUP_FRAGMENT	29	A duplicate fragment was received.
IP_ERR_TASK_NOT_FOUND	30	The task associated with the operation does not exist.

Error code	Value	Description
IP_ERR_TIMEOUT	31	The operation timed out.
IP_PKT_SENT	32	Packet sent.
IP_ERR_ALIAS_SIZE	33	There is not enough room for the IP alias; increase the value of configuration option IP_V4_ALIAS_COUNT.
IP_ERR_ALIAS_CONFIGURED	34	An IP alias with the given IP address already exists.

5.12 Types and Definitions

This section describes the main elements (structures, typedefs, options, and flags) that are defined in the API Header files.

t_ip_addr

The *t_ip_addr* structure stores IPv4 and IPv6 addresses in big-endian mode:

Element	Type	Description
ipa_address[IP_ADDR__MAX_LENGTH]	uint8_t	The IP address.
ipa_version	t_ip_ver	The IP address version, either IPV_IP_V4 or IPV_IP_V6.

t_ip_opt

The *t_ip_opt* structure defines the IP packet options:

Element	Type	Description
ipo_ttl	uint8_t	The IP header's TTL field.
ipo_tos	uint8_t	The IP header's TOS field.

Copyright HCC Embedded 2018

t_ip_route

The *t_ip_route* structure defines the IP route entry.

For IPv6 addresses the network mask and gateway are discarded. For IPv6 the route is chosen by selecting the address with the longest matching prefix.

Element	Type	Description
ipr_state	uint8_t	The link state.
ipr_ifc_hdl	t_ip_ifc_hdl	The interface handle.
ipr_nwaddr	t_ip_addr	The network address. If this is 0 the entry is the default gateway entry.
ipr_netmask[IP_V4_ADDR_SIZE]	uint8_t	The netmask. **This is only used for IPv4.**
ipr_gateway[IP_V4_ADDR_SIZE]	uint8_t	The gateway address. **This is only used for IPv4.**
ipr_route_hdl	t_ip_route_hdl	The route handle.

t_ip_config

The *t_ip_config* structure defines the IP configuration.

> **Note:** The IPv4 and IPv6 elements only apply if the respective IP_V4_ENABLE and IP_V6_ENABLE options are set.

Element	Type	Description
ipc_state	uint8_t	The state of the IP config. This is only used in **ip_get_config()**.
The following are only used for IPv4:		
ipc_ipaddr	t_ip_addr	The IPv4 address.
ipc_netmask[IP_V4_ADDR_SIZE]	uint8_t	The network mask.
ipc_gateway[IP_V4_ADDR_SIZE]	uint8_t	The gateway address.
ipc_dns1_addr[IP_V4_ADDR_SIZE]	uint8_t	The first DNS server address.
ipc_dns2_addr[IP_V4_ADDR_SIZE]	uint8_t	The second DNS server address.
The following are only used for IPv6:		
ipc_ip_v6_addr	t_ip_addr	The IPv6 address.
ipc_ip_v6_dns1[IP_V6_ADDR_SIZE]	uint8_t	The IPv6 DNS first address.
ipc_ip_v6_dns2[IP_V6_ADDR_SIZE]	uint8_t	The IPv6 DNS second address.

IP Config Flags

The possible IP config flags (*ipc_state*) are as follows:

State	Value	Description
IP_V4_STATE_CONFIGURED	4	Flag indicating IPv4 address is configured.
IP_V6_STATE_CONFIGURED	8	Flag indicating IPv6 address is configured.

IP Config Link States

The link state can be extracted from the field *ipc_state* by applying the mask IP_STATE_LINK_MASK. The possible link states are as follows:

State	Value	Description
IP_STATE_STOPPED	0	The route is unavailable.
IP_STATE_LINK_DOWN	1	The link is down.
IP_STATE_LINK_UP	2	The link is working.
IP_STATE_LINK_OFFLINE	3	The link is offline.

t_ip_port

The *t_ip_port* structure is the IP port descriptor:

Element	Type	Description
ipp_ip_addr	t_ip_addr	The IP address of the port.
ipp_port	uint16_t	The port number.

t_ip_ntf

The *t_ip_ntf* structure is the IP notification descriptor:

Element	Type	Description	Notes
ntf_fn	t_ip_ntf_fn	The notification function to call.	User variable.
ntf_param	uint32_t	The parameter to send with the notification.	User variable.
p_ntf_next	struct s_ip_ntf *	A pointer to the next entry.	For internal use only.

t_ip_get_buf_tn

The *t_ip_get_buf_tn* structure is the notification structure used by any user attempting to get a buffer from the pool. The **tcp_get_buf()**, **udp_get_buf()**, **ip_nwd_get_buf()** and **ip_nwd_get_buf_ext()** functions all use this to get a notification.

This structure is used to tell **tcp_get_buf()** and **udp_get_buf()** the requested timeout and/or notification, in case the requested buffer is not available. It has the following elements:

Element	Type	Description
igb_timeout	uint32_t	This timeout tells **tcp_get_buf()** or **udp_get_buf()** how long to wait if the requested buffer is not available. Valid values are IP_WAIT_NONE, IP_WAIT_FOREVER, or the time in milliseconds. This member is only valid if the IP stack is used with an OS. If it is used without an OS, all functions are non-blocking.
p_igb_ntf	t_ip_ntf *	A pointer to a **t_ip_ntf** structure that contains the pointer to the notification function and the parameter passed in the notification. The notification function is called if the buffer is not available when a **tcp_get_buf()** or **udp_get_buf()** function fails and, after returning, a buffer becomes free.

t_ip_get_buf

The *t_ip_get_buf* structure defines the IP Get Buffer property. This structure contains information on the buffer obtained by using **tcp_get_buf()** or **udp_get_buf()**.

Element	Type	Description
p_igb_buf	uint8_t *	A pointer to the buffer.
igb_buf_len	uint16_t	The length of the allocated buffer.

t_arp_find

The *t_arp_find* structure is used by the ARP to find entries:

Element	Type	Description
arpf_type	uint8_t	The type of ARP entry (Static or Dynamic).
arpf_hw_addr [NWDRIVER_HW_ADDRESS_SIZE]	uint8_t	The physical address.
arpf_ip_addr	t_ip_addr	The IP address.
arpf_route_hdl	t_ip_route_hdl	The route handle.
arpf_lease_time	uint16_t	The ARP entry lease time.
arpf_base_lease_time	uint16_t	The default lease time for ARP entries.
arpf_pos	uint16_t	The next entry position in the ARP table. **For internal use only**.

t_ip_pool_qprop

The *t_ip_pool_qprop* structure defines the IP pool queue property:

Element	Type	Description
ipq_size	uint16_t	The size of the buffers in the queue.
ipq_cnt	uint16_t	Count: the minimum number of *ipq_size* buffers that must be present in the queue.

t_ping_cb_fn

The **t_ping_cb_fn** typedef defines the ping callback function.

Format

```
typedef void (* t_ping_cb_fn )(
    const uint32_t       param,
    const t_ping_state   ping_state )
```

Arguments

Element	Type	Description
param	uint32_t	The parameter passed.
ping_state	t_ping_state	The ping state.

Ping States

The *t_ping_state* typedef defines the ping states:

Parameter	Description
PING_ST_FREE	There is no active ping for the given handle.
PING_ST_WAITING	Ping is waiting for reply.
PING_ST_DONE	Ping completed successfully.
PING_ST_ERR	Ping failed.

IP Configuration Options

The IP configuration options are as follows:

Return Value	Value	Description
IP_IFC_OPT_DHCP_ENABLE	1	Enable DHCP for this interface.
IP_IFC_OPT_DEF_GATEWAY	2	Set this interface to be the default gateway.
IP_V6_IFC_OPT_DHCP_ENABLE	4	Enable DHCP V6 for this interface.
IP_V6_IFC_OPT_DEF_GATEWAY	8	Set this IPv6 interface to be the default gateway.

Invalid Handles

The invalid handles are as follows:

Return Value	Value	Description
IP_INVALID_IFC_HDL	UINT8_MAX	Invalid interface handle.
IP_INVALID_ROUTE_HDL	UINT32_MAX	Invalid route handle.

Timeout Values

The timeout values are as follows:

State	Value	Description
IP_WAIT_NONE	0	Do not wait.
P_WAIT_FOREVER	UINT32_MAX	Wait indefinitely.

Notifications

The notifications are as follows:

Notification	Value	Description
IP_NTF_BUF_AVAIL	0x00000001	Buffer available.
IP_NTF_TX_RDY	0x00000002	Ready to transmit.
IP_NTF_TX_ERR	0x00000004	Transmit error.
IP_NTF_RX_RDY	0x00000008	Ready to receive.
IP_NTF_CON_ESTD	0x00000010	Connection established.
IP_NTF_CON_CLOSE	0x00000020	Connection closed.
IP_NTF_CON_ABORT	0x00000040	Connection aborted.
IP_NTF_DCON_WAIT	0x00000080	Disconnect wait.
IP_NTF_PORT_CLOSE	0x00000100	Port closed.
IP_NTF_PORT_ABORT	0x00000200	Port aborted.
IP_NTF_CONFIG_CHANGE	0x00010000	Configuration has changed.
IP_NTF_IP_ADDR_CONFLICT	0x00020000	IP address conflict: another device on the network has the same IP address.

Copyright HCC Embedded 2018

t_mld_listen

The *t_mld_listen* structure is used by Multicast Listener Discovery to describe a multicast listener:

Element	Type	Description
p_mldl_mc_addr	uint8_t *	The multicast address to listen to.
mldl_mode	uint8_t	The filtering mode, one of the following: • MLD_MODE_EXCLUDE - addresses in the source address list are rejected. • MLD_MODE_INCLUDE - only addresses in the source address list are accepted.
mldl_src_count	uint8_t	The number of source addresses.
p_mldl_src_list	uint8_t **	A pointer to the IP source addresses list. See the example in mld_set_mc_listen.

t_ip_config_ntf_fn

The **t_ip_config_ntf_fn** typedef specifies the format of a configured notification type.

Register this with the module by calling **ip_register_config_ntf()**.

Format

```
typedef void ( * t_ip_config_ntf_fn )(
    const t_ip_ifc_hdl           ifc_hdl,
    const uint32_t               ntf,
    const t_ip_config * const    p_ip_config,
    const char_t * const         p_fqdn )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
ntf	The notification being signaled.	uint32_t
p_ip_config	A pointer to the configuration.	t_ip_config *
p_fqdn	A pointer to the FQDN descriptor.	char_t *

6 Code Examples

The following examples show how to initialize, start, and then stop the stack.

6.1 Initializing the Stack

This example code shows how to initialize the stack.

```
/************************** Global variables ********************************/
t_ip_ifc_hdl  eth0_ifc_hdl;
t_ip_config  ip_config;

/************************** Function definitions ****************************/
t_ip_ret hcc_init_ip ( void )
{
    t_ip_ret  ret_val;

    /* ---- Network Stack initialization ---- */
    ret_val = IP_ERROR;
    psp_tick_init();

    if ( hcc_timer_init() == HCC_TIMER_SUCCESS )
    {
        ret_val = ip_stack_init();
        if ( ret_val == IP_SUCCESS )
        {
            /* Network Controller initialization: see manual for specific network controller */
            ret_val = ip_eth_ifc_init( synopsys_eth_drv_init
                                        , 0
                                        , IP_IFC_OPT_DHCP_ENABLE
                                        , &eth0_ifc_hdl );
            if ( ret_val == IP_SUCCESS )
            {
                ret_val = ftp_init();
                if ( ret_val == FTP_SUCCESS )
                {
                    ret_val = tftp_init();
                    if ( ret_val == TFTP_SUCCESS )
                    {
                        ret_val = ftp_add_user( "user", "user", "/" );
                        if ( ret_val == FTP_SUCCESS )
                        {
                            ret_val = ftp_user_init();
                            if ( ret_val == FTP_SUCCESS )
                            {
                                ret_val = tftp_user_init();
                            }
                        }
                    }
                }
            }
        }
    }
    return ret_val;

} /* hcc_init_ip */
```

6.2 Starting the Stack

This example code shows how to start the stack.

```
t_ip_ret hcc_start_ip ( void )

{
    t_ip_ret  ret_val = IP_ERR_INIT;      /* For the IP function's return value */
    if ( ip_stack_start() == IP_SUCCESS )
    {
        if ( ip_eth_ifc_start( eth0_ifc_hdl ) == IP_SUCCESS )
        {
            if ( hcc_timer_start() == HCC_TIMER_SUCCESS )
            {
                if ( ftp_start() == FTP_SUCCESS )
                {
                    if ( tftp_start() == TFTP_SUCCESS )
                    {
                        if ( ftp_user_start() == FTP_SUCCESS )
                        {
                            if ( tftp_user_start() == TFTP_SUCCESS )
                            {
                                ret_val = IP_SUCCESS;
                            }
                        }
                    }
                }
            }
        }
    }

     return ret_val;

 } /* hcc_start_ip */
```

6.3 Stopping the Stack

This example code shows how to stop the stack.

```
t_ip_ret hcc_stop_ip ( void )

{
    t_ip_ret  ret_val = IP_ERR_INIT;  /* For the IP function's return value */

    if ( ftp_stop() == FTP_SUCCESS )

    {
        if ( tftp_stop() == TFTP_SUCCESS )

        {
            if ( ip_eth_ifc_stop( eth0_ifc_hdl ) == IP_SUCCESS )

            {
                if ( ip_stack_stop() == IP_SUCCESS )

                {
                    ret_val = IP_SUCCESS;
                }
            }

        }

    }

    return ret_val;

} /* hcc_stop_ip */
```

7 Integration

This section describes all aspects of the TCP/IP system that require integration with your target project.

This includes porting and configuration of external resources.

7.1 Utilities

The code creates and uses a single timer in the **hcc_timer** module.

The **hcc_timer** module is included in your system when you install the base TCP/IP modules.

7.2 OS Abstraction Layer: OAL

All HCC modules use the OS Abstraction Layer (OAL). This allows modules to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The stack can run in one of two environments:

- An RTOS environment.
- A non-RTOS environment, managed by using the HCC OAL. Here you only need to configure the OAL to make certain resources available for the stack.

The TCP/IP stack base system uses the following OAL components:

OAL Resource	Number Required
Tasks	1
Mutexes	1
Events	(IP_MAX_TASK + 1) events.

Note: For each module added to the system, you may need to add additional OAL resources. Check this in the manuals for individual modules.

7.3 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer. For full details of its functions and macros, see the *HCC Base Platform Support Package User Guide*.

The TCP/IP dual stack base system makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_get_tick_count()	psp_base	psp_tick	Counts the number of ticks.
psp_memcmp()	psp_base	psp_string	Compares two blocks of memory.
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.
psp_strncmp()	psp_base	psp_string	Compares two strings of defined length.
psp_strnlen()	psp_base	psp_string	Gets the length of a string.

The TCP/IP dual stack base system makes use of the following standard PSP macros:

Macro	Package	Element	Description
PSP_RD_BE16	psp_base	psp_endianness	Reads a 16 bit value stored as big-endian from a memory location.
PSP_RD_BE32	psp_base	psp_endianness	Reads a 32 bit value stored as big-endian from a memory location.
PSP_WR_BE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as big-endian to a memory location.
PSP_WR_BE24	psp_base	psp_endianness	Writes a 24 bit value to be stored as big-endian to a memory location.
PSP_WR_BE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as big-endian to a memory location.

8 Memory Management

Note: You only need to read this section if you want to configure memory management on your system.

8.1 Network Drivers

api_nwdriver.h

nwp_rxbuf_count is added to the network driver property structure to tell the higher layer how many buffers the driver requires. This is only required if the higher layer needs to feed buffers to the network driver by using **add_buf()**.

8.2 Buffer Pools

The pool module is part of the base IP stack. It allows free association of memory pools with network drivers. It can be used to share the same memory pool between different network drivers, or used in the traditional way where the network driver defines its memory area. The following pool association methods can be used:

- The memory pool is defined by the network driver. In this case the network driver properties *p_nwp_buf* and *nwp_buf_size* define the memory area to be used for the frames.
- You create a memory pool and specify that it is to be used for an initialized interface.

Memory area overlapping is allowed; this means that the same memory, or part of the same memory, can be used by multiple network drivers.

Note: If one network driver uses part of another network driver's memory area, the smallest common part of the area is used by the pool for the frames. This situation is only allowed if none of the interfaces were started (by using **ip_..._ifc_start()**) when the partially overlapping area was configured.

You can configure the requested buffer sizes for a pool and the minimum number of buffers; see Network Driver Memory Pool Options.

There are two ways that network drivers can obtain a receive buffer in which received packets from the network can be written:

- The traditional RX buffer feeding mechanism where the IP stack feeds network driver property *nwp_rxbuf_count* number of buffers at startup by using **add_buf()**. After this it tries to add buffers to keep *nwp_rxbuf_count* buffers available for the network driver.

- The network driver requests a buffer from a pool when required. For this, use **ip_nwd_get_buf()** from protected network driver functions and **ip_nwd_get_buf_ext()** from unprotected functions. These are listed in the following section.

8.3 Protected and Unprotected Network Driver Functions

The following network driver functions are protected:

- **p_nwfn_get_state()**
- **p_nwfn_receive()**
- **p_nwfn_send()**
- **p_nwfn_addbuf** – calling **ip_nwd_get_buf()** or **ip_nwd_get_buf_ext()** from this function has no meaning.
- **p_nwfn_get_hw_addr()**
- **p_nwfn_set_hw_addr()**
- **p_nwfn_set_filter()**
- **p_nwfn_set_multicast_table()**
- **p_nwfn_get_link_speed()** – the notification called when **ip_nwd_get_buf()** failed to get the requested buffer and a notification is requested in case a buffer becomes available in that pool.

The following network driver functions are unprotected:

- **p_nwfn_init()**
- **p_nwfn_start()**
- **p_nwfn_stop()**
- **p_nwfn_delete()**

9 Checking the Status of the Physical Ethernet Cable

You can check the link state by using a notification callback function. You can register this function by calling the **ip_register_config_ntf()** function. Call that function with *ip_config_ntf_fn* as the argument, the function name to register. The following example code shows how to implement the function.

Note: This is only an example; modify this to suit your requirements.

```
void ip_config_ntf ( const t_ip_ifc_hdl ifc_hdl
                    , const uint32_t ntf
                    , const t_ip_config * p_ip_config
                    , const char_t * const p_fqdn )
{
    char_t str[40];
    t_ip_addr  ip_addr;

    if ( ifc_hdl == eth0_ifc_hdl )
    {
        printf( "ETH 0:" );

        if ( ntf == IP_NTF_CONFIG_CHANGE )
        {
            printf( " Configuration changed." );
        }
        else
        {
            printf( " Configuration state has changed." );
        }

    switch ( p_ip_config->ipc_state & IP_STATE_LINK_MASK )
    {
        case IP_STATE_STOPPED:
            printf( "\r\nINTERFACE IS STOPPED" );
            break;

        case IP_STATE_LINK_DOWN:
            printf( "\r\nINTERFACE IS LINKED DOWN" );
            break;

        case IP_STATE_LINK_UP:
            printf( "\r\nINTERFACE IS LINKED UP ");
            break;

        default:
            printf( "\r\nUNKNOW STATE" );
            break;
    } /* switch */
#if IP_V4_ENABLE
    if ( ( p_ip_config->ipc_state & IP_V4_STATE_CONFIGURED ) != 0U )
    {
        ip_addr_to_str( &(p_ip_config->ipc_ipaddr), str, 40);
        printf( "\r\nIP address : ");
        printf((char *)str);

        ip_addr.ipa_version = IPV_IP_V4;I would
        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_netmask, 4U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( "\r\nNetmask     : ");
        printf((char *)str);

        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_gateway, 4U );
        ip_addr_to_str( &(ip_addr), str, 40);
```

```
        printf( "\r\nGateway     : ");
        printf((char *)str);

        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_dns1_addr, 4U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( "\r\nDNS1        : ");
        printf((char *)str);

        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_dns2_addr, 4U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( "\r\nDNS2        : ");
        printf((char *)str);

        printf( "\r\nFQDN        :  ");
        printf((char *)p_fqdn );
    }
#endif
#if IP_V6_ENABLE
    if ( ( p_ip_config->ipc_state & IP_V6_STATE_CONFIGURED ) != 0U )
    {
        printf( "\r\nIP6 address : " );
        ip_addr_to_str( &(p_ip_config->ipc_ip_v6_addr), str, 40);
        printf( str );

        printf( "\r\nIP6 DNS1 address : " );
        ip_addr.ipa_version = IPV_IP_V6;
        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_ip_v6_dns1, 16U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( str );

        printf( "\r\nIP6 DNS2 address : " );
        (void)psp_memcpy( ip_addr.ipa_address, p_ip_config->ipc_ip_v6_dns2, 16U );
        ip_addr_to_str( &(ip_addr), str, 40);
        printf( str );
    }
#endif
    printf( "\r\n" );
    }
} /* ip_config_ntf */
```