HCC TCP User Guide

Version 2.80

For use with TCP module versions 6.09 and above

Date: 20-Jun-2017 10:33

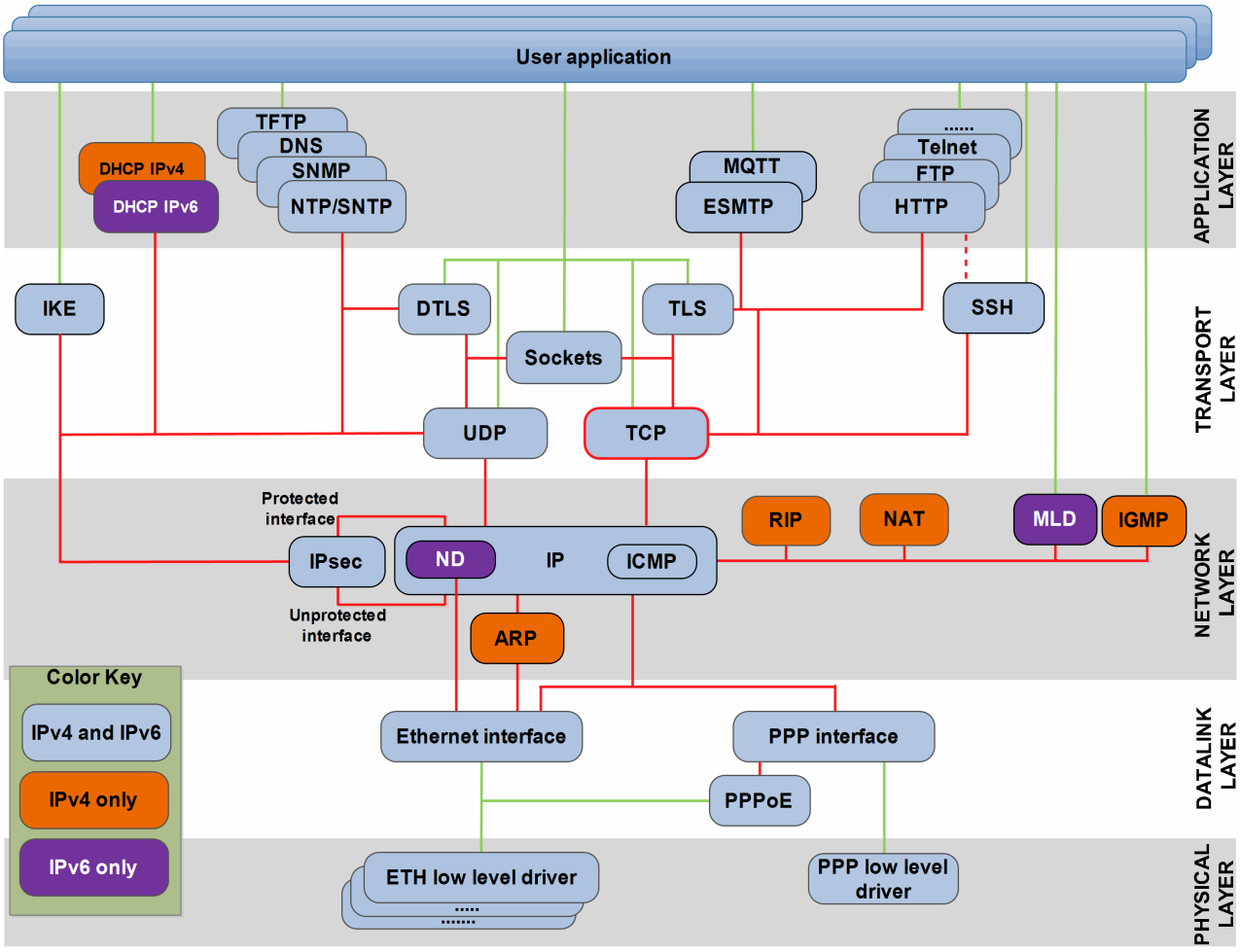
Table of Contents

1 System Overview

1.1 Introduction

This guide is for those who want to implement the optional TCP module as part of HCC's TCP/IP Suite. The TCP module allows an application to send and receive data using TCP connections across a TCP/IP network.

The location of the TCP/IP module within the HCC TCP/IP stack is shown below. (In this diagram green lines show interfaces available to users of the stack, red lines show interfaces internal to the TCP/IP system.)



The TCP module uses the native TCP Application Programming Interface (API). Users of the Sockets API, which is an alternative to the native TCP API, should refer to the *HCC TCP/IP Sockets Interface User Guide* .

1.2 Feature Check

The main features of the system are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Complies with the HCC MISRA-compliant TCP/IP stack.
- Designed for integration with both RTOS and non-RTOS based systems.
- Compliant with RFC 793, the RFC that defines extensions to Sockets for IPv6.
- Supports zero copy send and receive.
- Provides an optional MISRA-compliant native API.
- Provides an optional Sockets API.

1.3 Packages and Documents

Packages

The table below list the packages. For all systems the TCP/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 TCP/IP Dual Stack base package.
ip_base_v4, ip_base_v6	The base packages for IPv4 and IPv6, respectively.
mip_tcp	The TCP module. The HCC TCP package, described in this document, is an optional extension to the HCC TCP/IP suite. No additional packages beyond those included in the base system are required.
ip_socket	The BSD Sockets interface. This is an optional extension to the UDP and TCP packages. It enables a standard Sockets interface to be used for accessing TCP and UDP ports. If this package is not used, you must use the HCC native TCP API, described in this manual, for accessing TCP ports.
ip_tcp_test	A suite of reference code for exercising the TCP interface to the network stack. This provides a test suite for using both the native TCP interface and the Sockets interface.

Documents

For an overview of the HCC 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 User Guide

This is this document.

HCC TCP/IP Dual Stack System User Guide

This is the core document that describes the complete TCP/IP stack. It covers both IPv4 and IPv6 systems.

HCC TCP/IP Sockets Interface User Guide (optional)

This document is the Application Programming Interface (API) guide for the HCC Sockets package.

1.4 Change History

This section describes past changes to this manual.

- To download earlier manuals, see Archive: TCP User Guide.
- For the history of changes made to the package code itself, see History: mip_tcp.

The current version of this manual is 2.80. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
2.80	2017-06-20	6.09	New *Change History* format.
2.70	2017-03-20	6.07	Updated network diagram.
2.60	2017-01-18	6.06	Updated network diagram.
2.50	2016-03-18	6.01	Updated to work with IP base major version 6.
2.40	2015-12-07	5.01	Added new parameters to **tcp_connect()** function.
2.30	2015-09-04	4.01	Modified *Introduction*.
2.20	2015-03-31	4.01	Added software change history to manual.
2.10	2014-08-22	4.01	Reorganized *System Overview* section.
2.00	2014-05-15	3.01	First online version.

2 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 file.

2.1 API Header File

The file **src/api/api_ip_tcp.h** should be included by any application using the system. This is the only file that should be included by an application using this module. For details of these API functions, see Application Programming Interface.

2.2 Configuration File

The file **src/config/config_ip_tcp.h** contains all the configurable parameters of the system. You can configure these as required. For detailed explanation of these options, see Configuration Options.

2.3 TCP System Files

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

File	Description
tcp.c	Implements the TCP protocol.
tcp.h	TCP protocol header file.

2.4 Version File

The file **src/version/ver_ip_tcp.h** 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 Configuration Options

Set the system configuration options in the file **src/config/config_ip_tcp.h**. This section lists the available configuration options and their default values.

TCP_SERVER_PORTS

The maximum number of ports that can be opened for listening by using **tcp_open()**. The default is 4.

TCP_CONNECTIONS

The maximum number of connections that can be accepted/initiated by using **tcp_accept()** and **tcp_connect()**. The default is 8.

TCP_MSS

The Maximum Segment Size (MSS) used on the local node. This defines the maximum size of the data received in one TCP packet. The default is IP_MTU_SIZE.

TCP_MSS_PER_WINDOW

This defines the window size by setting the number of segments in it. The default is 2.

TCP_MAX_UNACK_LEN

The maximum unacknowledged data length. If the total unacknowledged data length exceeds this value, an ACK is sent immediately, otherwise the ACK is sent after the latest TCP_RX_UNACK_TIMEOUT number of milliseconds. The default is 128.

TCP_LOCAL_PORT_BASE, TCP_LOCAL_PORT_COUNT

The base port number used when a connection is initiated by calling **tcp_connect()**. The range of local port numbers used in this case is from TCP_LOCAL_PORT_BASE to TCP_LOCAL_PORT_BASE +TCP_LOCAL_PORT_COUNT - 1.

> **Note:** TCP_LOCAL_PORT_COUNT must be bigger than TCP_CONNECTIONS.

The default for TCP_LOCAL_PORT_BASE is 10500. The default for TCP_LOCAL_PORT_COUNT is 500.

TCP_TIMER_PERIOD

The timer period in milliseconds. The default is 100. These configuration options set the various retry and timeout values for different TCP states. All timeout values are defined in milliseconds.

Note: The minimum expiry unit is TCP_TIMER_PERIOD. For example, if a timeout is defined to be 100ms and the period is set to 500ms, the timeout is automatically set to 500ms.

TCP_CONN_RETRY

The initial connection establishment retry value. The default is 2.

TCP_CONN_TIMEOUT

The initial connection establishment timeout value. The default is 5000.

TCP_CONN_PENDING_TIMEOUT

The pending connection timeout. If the stack has accepted a connection but the user did not call **tcp_accept()** for this period, the connection is aborted. The default is 2000.

TCP_TX_RETRY, TCP_TX_TIMEOUT

The TX retry and timeout define the expiry of unacknowledged transmitted packets. If timeout occurs, unacknowledged packets are retransmitted. If retry expires, the connection is aborted. The default for TCP_TX_RETRY is 5. The default for TCP_TX_TIMEOUT is 1000.

Another use for these options is for sending Zero Window Probe in case the send window size is too small and the remote node does not send a window update for this period.

TCP_RX_UNACK_TIMEOUT

If there are unacknowledged packets, this is the maximum time before an ACK packet is forced. The default is 200.

TCP_RX_OUT_OF_SEQ_TIMEOUT

The period after which all out of sequence packets are dropped. The default is 2000.

TCP_CLOSE_WAIT_TIMEOUT

The close wait timeout. If the remote node tries to close the connection and the user did not call **tcp_disconnect()** for this period, the connection is automatically closed. The default is 2000.

TCP_DCONN_RETRY, TCP_DCONN_TIMEOUT

The disconnect retry count and timeout. After expiry the connection is aborted in case the local node failed to close the connection. The default for TCP_DCONN_RETRY is 3. The default for TCP_DCONN_TIMEOUT is 200.

TCP_TIME_WAIT_TIMEOUT

The TIME-WAIT expiration, the amount of time before releasing a connection after closing/aborting it. This is required to ensure packets sent to a previous connection do not arrive at a new one. The default is 500.

TCP_RX_CHECKSUM_ENABLE

The default of 1 enables checksum verification for RX packets.

You can set the value to 0 to disable this. If you do this, the packet is accepted regardless of the setting. On many closed networks this field is redundant. On open networks, if verification is disabled the network driver should validate the checksum and discard invalid packets. Where the Ethernet controller supports this, HCC network drivers can enable this hardware level checking.

TCP_TX_CHECKSUM_ENABLE

The default of 1 enables checksum verification for RX packets.

You can set the value to 0 to disable this. If you do this, you must provide another mechanism to generate the checksum. This can often be done at an Ethernet controller level, reducing the CPU load. Many HCC network drivers support switching this feature on.

4 Using the API

This section describes HCC's native TCP Application Programming Interface (API) then shows time sequence diagrams of the communication between the server and the client, using TCP.

The native TCP API may be used to access TCP ports. It can be used in parallel with the TCP/IP stack's Sockets APIs, as long as the APIs do not simultaneously use the same ports.

The native API has these advantages over the Sockets API:

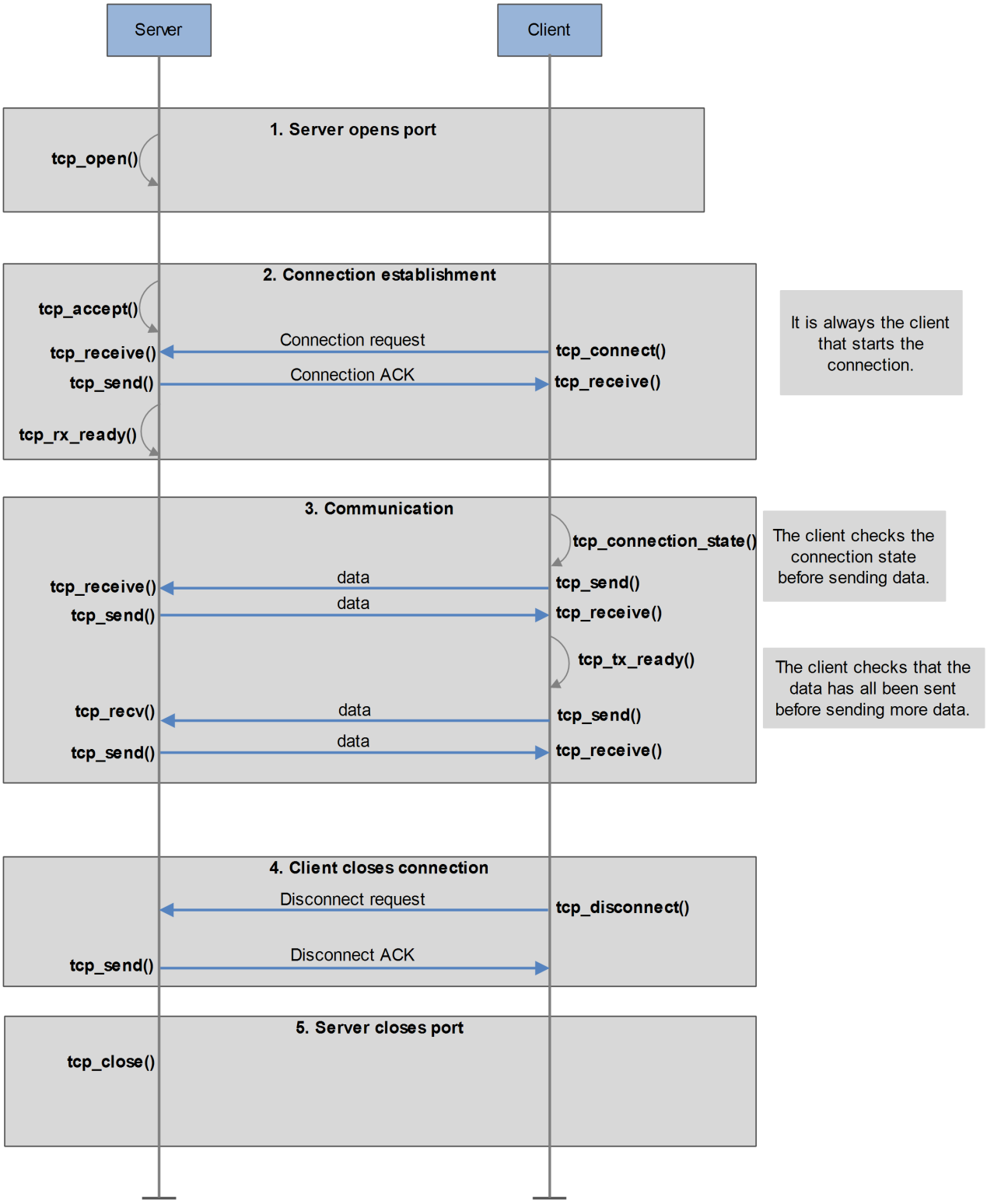
- The native API yields slightly better performance and allows the possibility of zero copy transfers.
- The native API is designed with modern and strict coding standards in mind, and its components are consequently easier to interface to applications requiring a more rigorous implementation.

The main drawbacks of using the native API, compared to the Sockets API, are:

- The Sockets API is an industry standard BSD Sockets interface. This means applications written for other platforms that use BSD sockets can be used unchanged.
- With Sockets, applications can use their own buffers to read and write data to the IP stack, although inevitably this means that there is a copy in the interface.

4.1 TCP Time Sequence

The following time sequence diagram shows the communication between the server and the client, using the TCP port.



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 native TCP API interface interface has the advantage that it is fast and simple, since it is designed specifically for this stack. However, the API is non-standard and requires the user to allocate data buffers from the IP buffer pool.

> **Note:** Certain TCP API functions behave differently, depending whether the system is running in OS mode or in POLL mode.

5.1 Functions

The functions are the following:

Function	Description
tcp_open()	Opens a TCP port to accept connections on.
tcp_close()	Closes an open TCP port.
tcp_accept()	Accepts a connection from a remote node on a TCP port previously opened by using **tcp_open()**.
tcp_connect()	Initiates a connection to a remote TCP port.
tcp_connect()	Disconnects the specified TCP connection.
tcp_connection_state()	Gets the state of the specified TCP connection.
tcp_get_buf()	Gets a TCP protocol buffer from the IP buffer pool.
tcp_release_buf()	Releases a buffer back to the IP buffer pool.
tcp_rx_ready()	Checks whether there is any unread data from the remote node on the specified TCP connection.
tcp_receive()	Receives data from the remote node on the specified TCP connection.
tcp_tx_ready()	Checks whether all data has been transmitted on the specified connection.
tcp_send()	Sends TCP data to a remote node on the specified TCP connection.
tcp_set_ip_opt()	Sets the IP options for the specified TCP connection.
tcp_get_remote_mss()	Returns the Maximum Segment Size (MSS) of the remote node on the specified TCP connection.
tcp_set_ifc_config()	Sets TCP configuration parameters for the specified interface.
tcp_get_stats()	Gets statistics from the TCP module.

tcp_open

Use this function to open a TCP port to accept connections on.

Format

```
t_ip_ret tcp_open (
    const uint16_t           port_num,
    const uint16_t           max_conn_cnt,
    t_ip_ntf * const         p_ntf,
    t_tcp_port_hdl * const   p_tcp_port_hdl )
```

Arguments

Argument	Description	Type
port_num	The port number.	uint16_t
max_conn_cnt	The maximum number of connections.	uint16_t
p_ntf	A pointer to the notification function.	t_ip_ntf *
p_tcp_port_hdl	Where to write the TCP port handle.	t_tcp_port_hdl *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_ERR_PORT_OPEN	The port is already open.
Else	See Error Codes.

tcp_close

Use this function to close an open TCP port.

Format

```
t_ip_ret tcp_close ( const t_tcp_port_hdl tcp_port_hdl )
```

Arguments

Argument	Description	Type
tcp_port_hdl	The TCP port handle.	t_tcp_port_hdl

Return Values

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

tcp_accept

Use this function to accept a connection from a remote node on a TCP port previously opened by using **tcp_open()**.

Format

```
t_ip_ret tcp_accept (
    const t_tcp_port_hdl     tcp_port_hdl,
    const uint32_t           timeout,
    t_ip_ntf * const         p_ntf,
    t_ip_port * const        p_ip_port,
    t_tcp_conn_hdl * const   p_tcp_conn_hdl )
```

Arguments

Argument	Description	Type
tcp_port_hdl	The TCP port handle.	t_tcp_port_hdl
timeout	The maximum time in seconds to wait for a connection. This is ignored in non-OS mode.	uint32_t
p_ntf	A pointer to the notification function.	t_ip_ntf *
p_ip_port	Where to write the remote node IP address and port.	t_ip_port *
p_tcp_conn_hdl	Where to write the connection handle.	t_tcp_conn_hdl *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_CONNECTION	No connection was accepted.
Else	See Error Codes.

tcp_connect

Use this function to initiate a connection to a remote TCP port.

Format

```
t_ip_ret tcp_connect (
    const uint16_t            src_ip_port,
    const t_ip_addr * const   p_dst_ip_addr,
    const uint16_t            dst_ip_port,
    const uint32_t            timeout,
    t_ip_ntf * const          p_ntf,
    t_tcp_conn_hdl * const    p_tcp_conn_hdl )
```

Arguments

Argument	Description	Type
src_ip_port	The source IP port number.	uint16_t
p_dst_ip_addr	A pointer to the destination IP address.	t_ip_addr *
dst_ip_port	The destination IP port number.	uint16_t
timeout	The maximum time in seconds to wait for connection establishment. This is ignored in non-OS mode.	uint32_t
p_ntf	A pointer to the notification function.	t_ip_ntf *
p_tcp_conn_hdl	Where to write the connection handle.	t_tcp_conn_hdl *

Return Values

Return value	Description
IP_SUCCESS	Connection established. (This is only seen in OS mode.)
IP_ERR_NO_CONNECTION	Connection could not be established during the timeout period. (This is only seen in OS mode.)
IP_CONN_PENDING	Connection procedure has started but the connection has not yet been fully established. (This is only seen in POLL mode.)
Else	See Error Codes.

tcp_disconnect

Use this function to disconnect the specified TCP connection.

This function may be called whenever you want to stop a connection. Always call it when another function returns the IP_DISCONNECT_WAIT error code.

Format

```
t_ip_ret tcp_disconnect ( const t_tcp_conn_hdl tcp_conn_hdl )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl

Return Values

Return value	Description
IP_SUCCESS	The connection was disconnected.
IP_ERR_NO_CONNECTION	No connection.
Else	See Error Codes.

tcp_connection_state

Use this function to get the state of the specified TCP connection.

Format

```
t_ip_ret tcp_connection_state ( const t_tcp_conn_hdl tcp_conn_hdl )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl

Return Values

Return value	Description
IP_SUCCESS	Connection is established.
IP_CONN_PENDING	Connection establishment is pending.
IP_ERR_NO_CONNECTION	Connection is not active.
IP_DISCONNECT_WAIT	Waiting for disconnection from the user.
Else	See Error Codes.

tcp_get_buf

Use this function to get a TCP protocol buffer from the IP buffer pool.

Specify the TCP handle of the connection requesting the buffer, the required length of that buffer, and a pointer to the location to place the retrieved buffer structure.

> **Note:** Check the return code of this function to ensure that a buffer was successfully allocated.

In case no buffer is available immediately, you can:

- Specify a timeout to wait for a buffer if none is available. The function waits for this period of time and, if a buffer becomes available during this period, the call returns successfully.
- Specify a notification function to be called if the call fails but a buffer becomes available later. If the timeout expires and you have specified a notification function, the call returns successfully. If a buffer becomes available later, the notification function is called to notify you that you can try requesting a buffer again.

Format

```
t_ip_ret tcp_get_buf (
    const t_tcp_conn_hdl      tcp_conn_hdl,
    const uint16_t            req_len,
    t_ip_get_buf_tn * const   p_get_buf_tn,
    t_ip_get_buf * const      p_get_buf )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
req_len	The requested buffer length.	uint16_t
p_get_buf_tn	A pointer to the structure holding the timeout and/or notification.	t_ip_get_buf_tn *
p_get_buf	Where to write the properties of the buffer structure.	t_ip_get_buf *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_ERR_INVALID_SIZE	Invalid size requested.
IP_ERR_NO_CONNECTION	Connection handle is invalid.
IP_ERR_INVALID_HDL	Invalid connection handle.
Else	See Error Codes.

tcp_release_buf

Use this function to release a buffer back to the IP buffer pool.

Note: This function only handles buffers allocated by **tcp_get_buf()** or buffer pointers returned by **tcp_receive()**.

Format

```
void tcp_release_buf ( uint8_t * const p_buf )
```

Arguments

Argument	Description	Type
p_buf	A pointer to the buffer to be released.	uint8_t *

Return Values

Return value
None.

tcp_rx_ready

Use this function to check whether there is any unread data from the remote node on the specified TCP connection.

Format

```
t_ip_ret tcp_rx_ready (
    const t_tcp_conn_hdl   tcp_conn_hdl,
    uint32_t * const       p_len )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
p_l	Where to write the amount of pending data. This pointer can be set to NULL if the information is not needed by the caller.	uint32_t *

Return Values

Return value	Description
IP_SUCCESS	There is unread data.
Else	See Error Codes.

tcp_receive

Use this function to receive data from the remote node on the specified TCP connection.

> **Note:** After the received data is processed, the buffer must be released using **tcp_release_buf()**, unless the caller re-uses the buffer for a subsequent **tcp_send()**.

Format

```
t_ip_ret tcp_receive (
    const t_tcp_conn_hdl   tcp_conn_hdl,
    const uint32_t         timeout,
    uint8_t * * const      p_buf,
    uint16_t * const       p_buf_len )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
timeout	The maximum time in seconds to wait for data to be available. In non-OS mode this field is ignored and is always non-blocking.	uint32_t
p_buf	Where to write the pointer to the received data.	uint8_t * *
p_buf_len	Where to write the length of the received data.	uint16_t *

Return Values

Return value	Description
IP_SUCCESS	Data is available and it is the last IP fragment.
IP_MORE_DATA	Data is available and it is not the last IP fragment.
Else	See Error Codes.

tcp_tx_ready

Use this function to check whether all data has been transmitted on the specified connection.

In OS mode a timeout can be used to wait for the transmit queues to empty.

Format

```
t_ip_ret tcp_tx_ready (
    const t_tcp_conn_hdl   tcp_conn_hdl,
    const uint32_t         timeout )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
timeout	The maximum time in seconds to wait if transmit is not ready. This is ignored in POLL mode.	uint32_t

Return Values

Return value	Description
IP_SUCCESS	TX queue is empty.
Else	See Error Codes.

tcp_send

Use this function to send TCP data to a remote node on the specified TCP connection.

> **Note:** *p_buf* must point to a buffer previously allocated by using **tcp_get_buf()**. If the function returns an error, release the buffer by using **tcp_release_buf**().

Format

```
t_ip_ret tcp_send (
    const t_tcp_conn_hdl   tcp_conn_hdl,
    uint8_t * const        p_buf,
    const uint16_t         buf_len )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
p_buf	A pointer to the buffer to transmit.	uint8_t *
buf_len	The length of the data to send.	uint16_t

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_DISCONNECT_WAIT	Waiting for disconnection from the user.
IP_ERR_INVALID_SIZE	Buffer length exceeds TCP_MSS.
IP_ERR_NO_CONNECTION	Connection not established.
Else	See Error Codes.

tcp_set_ip_opt

Use this function to set the IP options for the specified TCP connection.

Format

```
t_ip_ret tcp_set_ip_opt (
    const t_tcp_conn_hdl     tcp_conn_hdl,
    const t_ip_opt * const   p_ip_opt )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
p_ip_opt	A pointer to the IP options structure to use.	t_ip_opt *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_CONNECTION	Connection not established.
Else	See Error Codes.

tcp_get_remote_mss

Use this function to return the Maximum Segment Size (MSS) of the remote node on the specified TCP connection.

Format

```
t_ip_ret tcp_get_remote_mss (
    const t_tcp_conn_hdl   tcp_conn_hdl,
    uint16_t *             p_mss )
```

Arguments

Argument	Description	Type
tcp_conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
p_mss	Where to write the remote node's MSS.	uint16_t *

Return Values

Return value	Description
IP_SUCCESS	Successful execution.
IP_ERR_NO_CONNECTION	Connection not established.
Else	See Error Codes.

tcp_set_ifc_config

Use this function to set TCP configuration parameters for the specified interface.

Format

```
t_ip_ret tcp_set_ifc_config (
    const t_ip_ifc_hdl   ifc_hdl,
    const uint16_t       mss,
    const uint16_t       mss_per_window,
    const uint16_t       max_unack_len )
```

Arguments

Argument	Description	Type
ifc_hdl	The interface handle.	t_ip_ifc_hdl
mss	The maximum segment size.	uint16_t
mss_per_window	The number of segments per window.	uint16_t
max_unack_len	The maximum length of unacknowledged data.	uint16_t

Return Values

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

tcp_get_stats

Use this function to get statistics from the TCP module.

Format

```
void tcp_get_stats ( t_tcp_stats * p_stats )
```

Arguments

Argument	Description	Type
p_stats	A pointer to a structure for storing the statistics.	t_tcp_stats *

Return Values

Return value
None.

5.2 Error Codes

The following table shows the TCP-specific error codes.

Return Value	Description
TCP_INVALID_PORT_HDL	Invalid TCP port handle.
TCP_INVALID_CONN_HDL	Invalid TCP connection handle.

The following table shows the meaning of the IP return codes.

Return Value	Value	Description
IP_SUCCESS	0	Successful execution.
IP_MORE_DATA	1	There is more data pending.
IP_CONN_PENDING	2	Connection initiated but not complete.
IP_DISCONNECT_WAIT	3	Waiting for completion of the disconnection process.
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 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.

Return Value	Value	Description
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.
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_ERR_INVALID_BUFFER	22	An interface was started but the required pool buffer queue properties can't 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 wants to be deleted but 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 has been received with a checksum error.
IP_ERR_DUP_FRAGMENT	29	A duplicate fragment was received.
IP_ERR_TASK_NOT_FOUND	30	Task associated with operation does not exist.

Return Value	Value	Description
IP_ERR_TIMEOUT	31	Operation timed out.

5.3 Types and Definitions

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_ntf

t_ip_ntf is the IP notification descriptor structure:

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_port

t_ip_port is the IP port descriptor structure:

Element	Type	Description
ipp_ip_addr	uint32_t	The IP address of the port.
ipp_port	uint16_t	The port number.

t_ip_opt

t_ip_opt is the IP packet options structure:

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

t_ip_get_buf

t_ip_get_buf is the get buffer structure. This contains information on the buffer obtained by using **tcp_get_buf()**. It has two elements:

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_ip_get_buf_tn

t_ip_get_buf_tn is the IP pool get buffer notification structure. It is used to tell **tcp_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()** 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()** function fails and, after returning, a buffer becomes free.

t_tcp_stats

t_tcp_stats is the TCP statistics structure:

Element	Type	Description
ts_transmitted	uint32_t	The number of transmitted packets.
ts_retransmitted	uint32_t	The number of retransmitted packets.
ts_waitack	uint32_t	The number of packets waiting for acknowledgment.

6 Integration

This section describes all aspects of the TCP module that require integration with your target project. This includes porting and configuration of external resources.

6.1 Utilities

The TCP 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.

6.2 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 module makes use of the following standard PSP functions:

Function	Package	Element	Description
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.

The module 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_BE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as big-endian to a memory location.