

# TCPIP Sockets Interface User Guide

Version 3.30

For use with TCP/IP Sockets Interface module versions  
3.34 and above

**Date:** 15-Nov-2017 17:26

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

|                                   |    |
|-----------------------------------|----|
| System Overview                   | 4  |
| Introduction                      | 5  |
| Feature Check                     | 6  |
| Packages and Documents            | 7  |
| Packages                          | 7  |
| Documents                         | 7  |
| Change History                    | 8  |
| Source File List                  | 9  |
| API Header File                   | 9  |
| Configuration File                | 9  |
| Sockets System                    | 9  |
| Version File                      | 9  |
| Configuration Options             | 10 |
| Using the API                     | 11 |
| Sockets Usage Summary             | 11 |
| HCC Sockets Function Name Mapping | 12 |
| Data Paths                        | 13 |
| Creating Data Paths               | 13 |
| Managing Data Paths               | 13 |
| Closing Data Paths                | 13 |
| TCP Time Sequence                 | 14 |
| UDP Time Sequence                 | 15 |
| Application Programming Interface | 16 |
| Module Management                 | 16 |
| socket_init                       | 17 |
| socket_start                      | 18 |
| socket_stop                       | 19 |
| socket_delete                     | 20 |
| Sockets Functions                 | 21 |
| socket_accept                     | 22 |
| socket_bind                       | 23 |
| socket_close                      | 24 |
| socket_connect                    | 25 |
| socket_get_errno                  | 26 |
| socket_gethostbyaddr              | 27 |
| socket_gethostbyname              | 28 |
| socket_getopt                     | 29 |
| socket_htonl                      | 30 |
| socket_htons                      | 31 |
| socket_inet_aton                  | 32 |
| socket_inet_ntoa                  | 33 |
| socket_ioctl                      | 34 |

---

|                                |    |
|--------------------------------|----|
| socket_listen                  | 35 |
| socket_ntohl                   | 36 |
| socket_ntohs                   | 37 |
| socket_open                    | 38 |
| socket_poll                    | 39 |
| socket_recv                    | 40 |
| socket_recvfrom                | 41 |
| socket_select                  | 42 |
| socket_send                    | 44 |
| socket_sendto                  | 45 |
| socket_setopt                  | 46 |
| IPv6 Address Checking Macros   | 47 |
| Error Codes                    | 48 |
| Types and Definitions          | 49 |
| Sockets Macros and Definitions | 49 |
| Socket error value             | 49 |
| Socket domain                  | 49 |
| IPv4 address prefix            | 49 |
| Socket types                   | 49 |
| Any address                    | 49 |
| Option levels                  | 50 |
| Socket options                 | 50 |
| Select Macros                  | 50 |
| IP options                     | 51 |
| Sockets ioctl commands         | 51 |
| Poll events                    | 52 |
| Sockets Structure Definitions  | 53 |
| sockaddr                       | 53 |
| in_addr                        | 53 |
| in6_addr                       | 53 |
| sockaddr_in                    | 53 |
| sockaddr_in6                   | 53 |
| ip_mreq                        | 54 |
| hostent                        | 54 |
| t_fd_set                       | 54 |
| t_pollfd                       | 54 |
| linger                         | 54 |
| timeval                        | 55 |
| IPv6 loopback address          | 55 |
| IPv6 wild card address         | 55 |
| Integration                    | 56 |
| OS Abstraction Layer           | 56 |
| PSP Porting                    | 57 |

# 1 System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

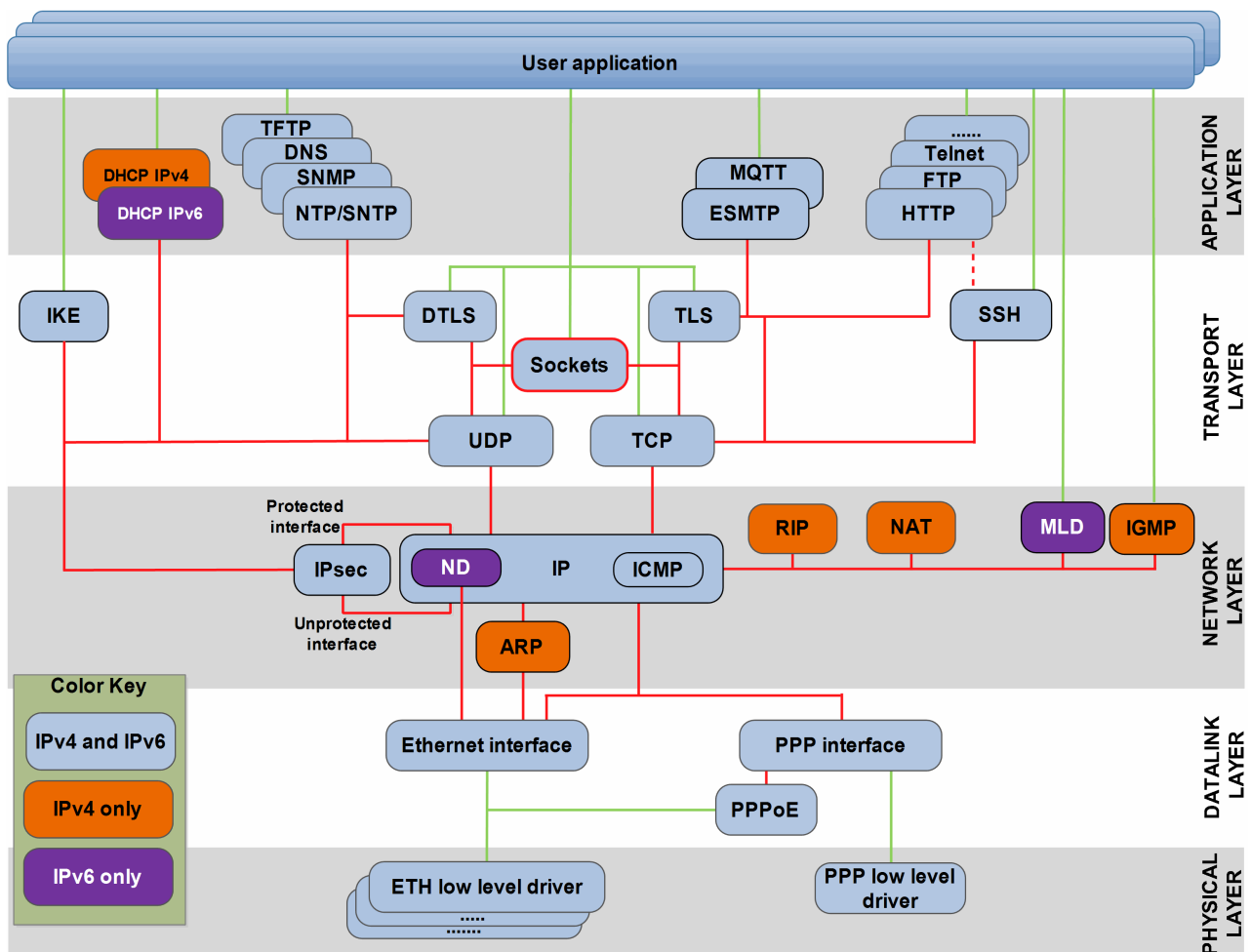
- [Introduction](#) – describes the main elements of the module. This section includes a diagram showing the position of this module within HCC's TCP/IP stack.
- [Feature Check](#) – summarizes the main features of the module as bullet points.
- [Packages and Documents](#) – the *Packages* section lists 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.

# 1.1 Introduction

This guide is for those who want to add the optional IP Sockets module to the HCC TCP/IP Suite. When integrated, the HCC IP Sockets module allows an application to communicate across a TCP/IP network using the industry standard BSD Sockets system calls. This manual describes how to use the HCC IP Sockets module.

User applications can use the native interfaces of the system simultaneously with the Sockets API as long as they operate on different UDP or TCP ports.

The Sockets module is part of the HCC MISRA-compliant TCP/IP stack, as shown below, and is designed specifically for use with it. (In this diagram green lines show interfaces available to users of the stack, red lines show interfaces internal to the TCP/IP system.)



## 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 BSD Sockets.
- IPv6 operation complies with [RFC 2553](#).
- Provides a standard interface for legacy applications to use.
- Allows portability of applications across Sockets-compliant systems.

---

## 1.3 Packages and Documents

---

### Packages

The table below lists the packages that you need in order to use this module:

| Package                   | Description  |
|---------------------------|--|
| <code>hcc_base_doc</code> | This contains the two guides that will help you get started. |
| <code>ip_socket</code>    | The Sockets package.   |
| <code>mip_base</code>     | The TCP/IP Dual Stack base package.                          |

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

#### HCC TCP/IP Sockets Interface User Guide

This is this document.

## 1.4 Change History

This section describes past changes to this manual.

- To download earlier manuals, see [TCP/IP PDFs](#).
- For the history of changes made to the package code itself, see [History: ip\\_socket](#).

The current version of this manual is 3.30. The full list of versions is as follows:

| Manual version | Date       | Software version | Reason for change  |
|----------------|------------|------------------|--|
| 3.30           | 2017-11-15 | 3.34             | Added options SO_KEEPALIVE, SO_BROADCAST, SO_REUSEADDR and SO_NODELAY.                                 |
| 3.20           | 2017-10-11 | 3.32             | Added PSP macros to <i>PSP Porting</i> .   |
| 3.10           | 2017-09-18 | 3.30             | Changed definition of IP_V4_MAP_IP_V6_ADDR_PREFIX_SIZE.<br><br>Added functions to <i>PSP Porting</i> . |
| 3.00           | 2017-09-05 | 3.29             | Corrected <i>Packages</i> list.  |
| 2.90           | 2017-06-20 | 3.29             | New <i>Change History</i> format.  |
| 2.80           | 2017-03-20 | 3.29             | Updated network diagram.   |
| 2.70           | 2017-01-16 | 3.28             | Updated network diagram.   |
| 2.60           | 2016-04-20 | 3.26             | Added function group tables to API section.  |
| 2.50           | 2015-11-12 | 3.21             | Added SO_LINGER option. Updated to work with IP base major version 6.                                  |
| 2.40           | 2015-08-18 | 3.19             | Extended <b>socket_select()</b> .  |
| 2.30           | 2015-04-28 | 3.19             | Added <b>socket_htonl()</b> , <b>socket_htons()</b> , <b>socket_ntohs()</b> , <b>socket ntohl()</b> .  |
| 2.20           | 2015-03-31 | 3.19             | Added software change history to manual.   |
| 2.10           | 2014-08-22 | 3.19             | Reorganized <i>System Overview</i> section.  |
| 2.00           | 2014-05-14 | 3.13             | First online version.  |



## 2 Source File List

This section lists and describes 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_socket.h` is the only file that should be included by an application using this module.

This file defines all API function calls and related structures that are available to the user. For details of these API functions, see [Application Programming Interface](#).

### 2.2 Configuration File

---

The file `src/config/config_ip_socket.h` contains all the configurable parameters of the UDP system. Configure these as required. For details of these options, see [Configuration Options](#).

### 2.3 Sockets System

---

The file `src/ip/stack/socket/socket.c` contains the TCP/IP Sockets source code. **This file should only be modified by HCC.**

### 2.4 Version File

---

The file `src/version/ver_ip_socket.h` contains the software version number of this release of the package. 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_socket.h`. This section lists the available options and their default values.

### SOCK\_TABLE\_SIZE

The maximum number of sockets that the system provides to the user. The default is 8U. Increasing the value increases the amount of RAM allocated by the system.

### SOCK\_DGRAM\_PORT\_BASE

The port base for DGRAM connections. This is used when `socket_sendto()` is called without Bind. The default is 10600.

The port range is from `SOCK_DGRAM_PORT_BASE` to `SOCK_DGRAM_PORT_BASE + SOCK_TABLE_SIZE - 1`.

### SOCK\_INET\_NTOA

Enable this to make the `socket_inet_ntoa()` function available. The default is 0.

**Note:** If the above option is set, the RAM usage is increased by  $16 * IP\_MAX\_TASK$ .

### SOCK\_GETHOSTBYNAME

Enable this to make the `socket_gethostbyaddr()` and `socket_gethostbyname()` functions available. The default is 0.

**Note:** If the above option is set, RAM usage is increased by about  $(36 + IP\_MAX\_FQDN\_SIZE) * IP\_MAX\_TASK$ .

### SOCK\_TASK\_ERRNO

If the socket layer is used with an RTOS, enable this to make error numbers task-specific. The default is 1.

### SOCK\_RCV\_INFO\_ENABLE

Enable this to allow Virtual LAN (VLAN) TCI and destination/source MAC address information requests for the last received packet on a socket. The default is 0.

**Note:** This option is only valid for Ethernet interfaces.

## 4 Using the API

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

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

The Sockets API has these advantages:

- It is an industry standard BSD Sockets interface. This means applications written for other platforms that use BSD sockets can be used unchanged.
- 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.

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

- 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.

### 4.1 Sockets Usage Summary

---

The Sockets interface functions are used as follows:

- In all cases the user application must communicate by using a socket.
- The user application must first allocate a socket for the communication.
- Once allocated, the socket handle is used for all accesses including connection establishment, data transfer, and connection tear-down.

**Note:** HCC supplies reference application code. Using this is recommended to ensure the interface is used correctly.

Blocking behavior is as follows:

- If the system is running without an RTOS then all socket calls are non-blocking.
- If the system is running with an RTOS then calls on open sockets will block unless they have been explicitly switched to non-blocking using the `socket_ioctl()` function.

Broadcasts operate as follows:

- If the IP address field is set to 255.255.255.255, the packet is broadcast on the network interface configured as the default gateway.

- If a specific subnetwork is specified (for example, 192.168.0.255), the broadcast is sent on the interface with that network address.

## 4.2 HCC Sockets Function Name Mapping

The names used by standard Sockets functions can cause conflicts in systems. For example, a function name of "**close**" occurs in many systems. As a consequence HCC has renamed the standard functions as shown in the table below. All function parameters are used exactly as in the Sockets specification. If required, these function names can be easily remapped using a definition file.

| HCC Name        | Standard Sockets Name |
|-----------------|-----------------------|
| socket_open     | socket                |
| socket_close    | close                 |
| socket_bind     | bind                  |
| socket_listen   | listen                |
| socket_accept   | accept                |
| socket_connect  | connect               |
| socket_recv     | recv                  |
| socket_recvfrom | recvfrom              |
| socket_send     | send                  |
| socket_sendto   | sendto                |
| socket_getopt   | getsockopt            |
| socket_setopt   | setsockopt            |
| socket_select   | select                |
| socket_poll     | poll                  |

## 4.3 Data Paths

---

### Creating Data Paths

#### TCP

There are two main ways of establishing TCP socket connections: as a server and as a client.

For a client the typical connection setup sequence is:

- **socket\_open()** to allocate a socket.
- **socket\_connect()** to connect to a specific remote server using the allocated socket.

For a server the typical connection setup sequence is:

- **socket\_open()** to allocate a socket.
- **socket\_bind()** to associate a TCP port with the allocated socket.
- **socket\_listen()** to wait for a remote client to connect to this service.
- **socket\_accept()** to wait for a remote port to connect to this server.

#### UDP

Both server and client use the same means of establishing UDP socket connections. The typical connection setup sequence is:

- **socket\_open()** to allocate a socket.
- **socket\_bind()** to associate a UDP port with the allocated socket.

### Managing Data Paths

For TCP data paths, use **socket\_recv()** and **socket\_send()** to transfer data.

For UDP data paths, use **socket\_recvfrom()** and **socket\_sendto()** to transfer data.

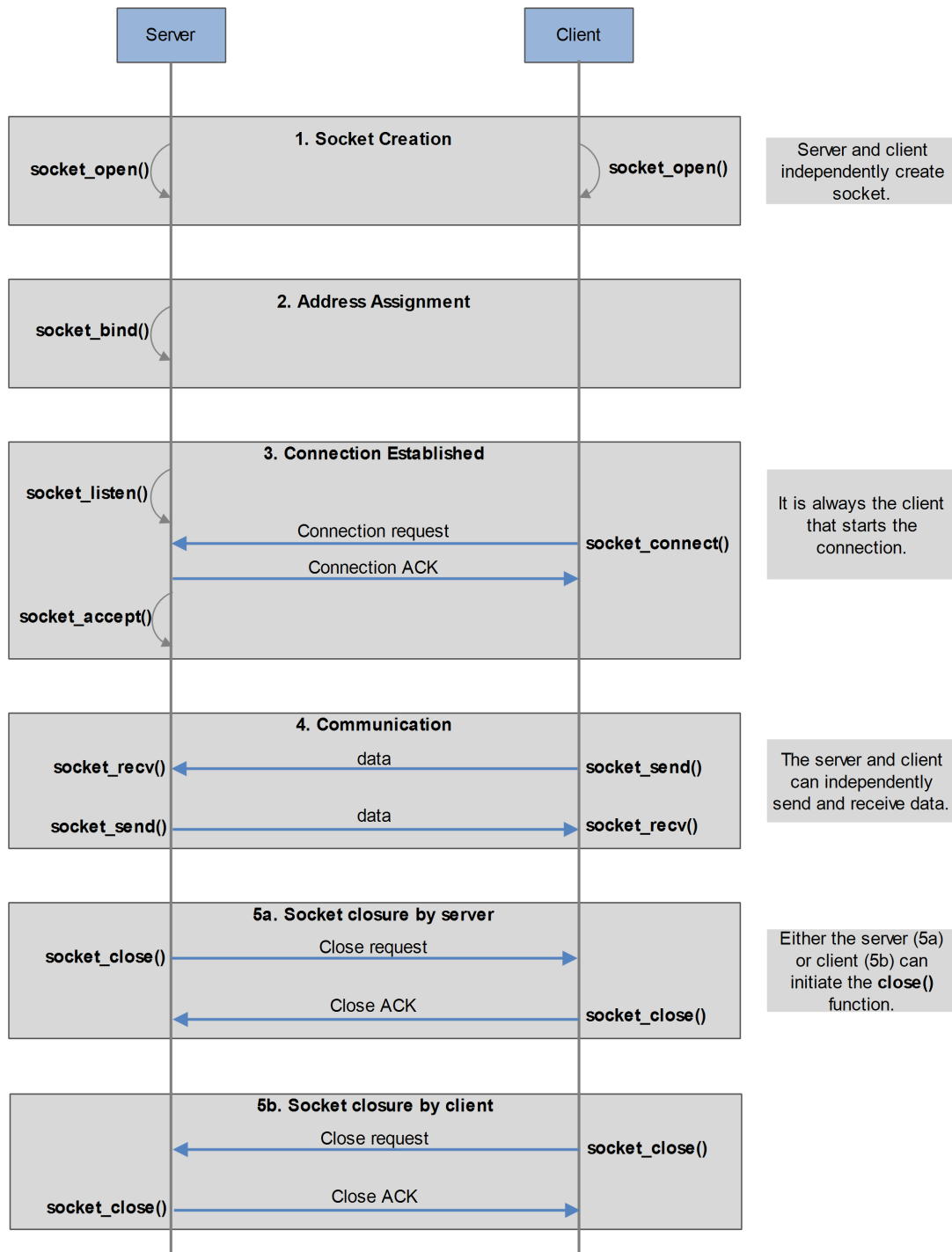
Sockets can be monitored for multiple events by using **socket\_select()** and **socket\_poll()**.

### Closing Data Paths

Terminate TCP and UDP data paths by using **socket\_close()**.

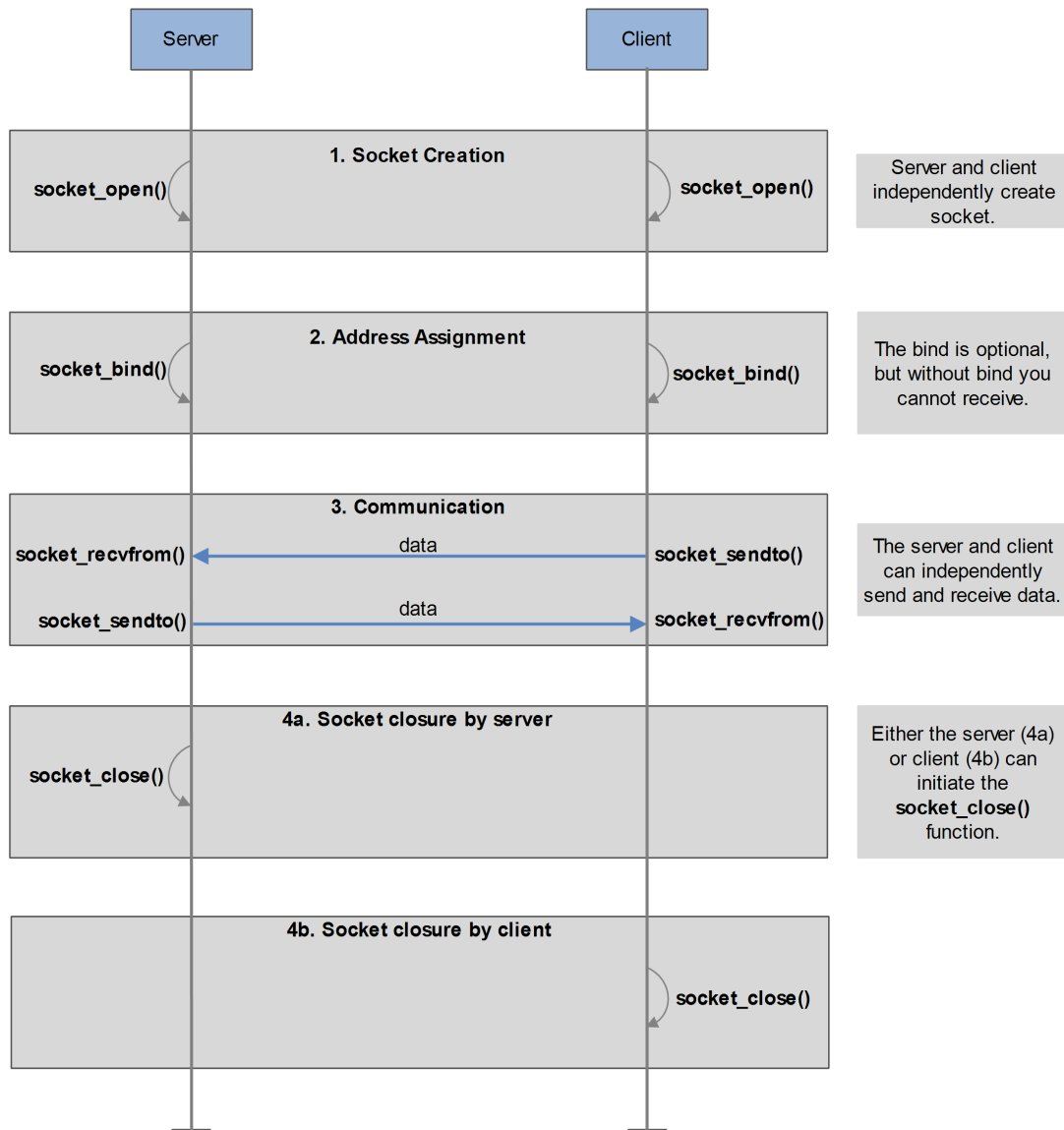
## 4.4 TCP Time Sequence

The following time sequence diagram shows the communication between the server and the client, using the TCP socket. The communication schema is connection-oriented: the two parties use the socket in stream mode.



## 4.5 UDP Time Sequence

The following time sequence diagram shows the communication between the server and the client, using the UDP socket. The communication schema is connectionless: the two parties use the socket in datagram mode.



## 5 Application Programming Interface

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

### 5.1 Module Management

The functions are the following:

| Function               | Description  |
|------------------------|--|
| <b>socket_init()</b>   | Initializes the module and allocates the required resources. |
| <b>socket_start()</b>  | Starts the module.   |
| <b>socket_stop()</b>   | Stops the module.  |
| <b>socket_delete()</b> | Deletes the module and releases the resources it used.       |

You must ensure that:

- **socket\_init()** is called before any other socket function.
- **socket\_start()** is called before the socket API is used.
- No API function is used after **socket\_stop()** is called.
- After **socket\_delete()**, **socket\_init()** is called before any other socket call.

**Note:**

- The module management calls use the TCP/IP stack error codes rather than socket-specific error codes. This allows easier management of error codes.
- All HCC module management functions are designed to be non-reentrant. The system designer must ensure that these calls are made systematically and cannot pre-empt each other. Typically you do this by performing all management of a particular module from a single task.



## socket\_init

Use this function to initialize the Sockets module, allocating the resources it requires.

**Note:** This must be the first Sockets function called.

### Format

```
t_ip_ret socket_init ( void )
```

### Arguments

#### Argument

None.

### Return values

| Return value | Description                   |
|--------------|-------------------------------|
| IP_SUCCESS   | Successful execution.         |
| IP_ERR_OS    | OS resource creation failure. |

## socket\_start

Use this function to start the Sockets module.

This function must be called before any standard socket API call is made.

**Note:** Call `socket_init()` before this function.

### Format

```
t_ip_ret socket_start ( void )
```

### Arguments

#### Argument

None.

### Return values

| Return value | Description                       |
|--------------|-----------------------------------|
| IP_SUCCESS   | Successful execution.             |
| Else         | See <a href="#">Error Codes</a> . |

## socket\_stop

Use this function to stop the Sockets module.

After the Sockets module is stopped, no standard socket API calls may be called until another call to **socket\_start()** is made.

### Format

```
t_ip_ret socket_stop ( void )
```

### Arguments

| Argument |
|----------|
| None.    |

### Return values

| Return value | Description                       |
|--------------|-----------------------------------|
| IP_SUCCESS   | Successful execution.             |
| Else         | See <a href="#">Error Codes</a> . |

## socket\_delete

Use this function to delete the Sockets module.

This function may only be called when the Sockets module is in the stopped state. It frees any resources allocated to the module by **socket\_init()**.

**Note:** After **socket\_delete()**, a new call to **socket\_init()** is required before any other function is called.

### Format

```
t_ip_ret socket_delete ( void )
```

### Arguments

| Argument |
|----------|
| None.    |

### Return values

| Return value | Description               |
|--------------|---------------------------|
| IP_SUCCESS   | Successful execution.     |
| IP_ERR_OS    | OS resource delete error. |

## 5.2 Sockets Functions

| Function                            | Description   |
|-------------------------------------|---|
| <code>socket_accept()</code>        | Starts listening on a TCP port.   |
| <code>socket_bind()</code>          | Opens a TCP/UDP port and associates it with the socket descriptor.              |
| <code>socket_close()</code>         | Closes the port (or, in the TCP case, connection) associated with the socket.   |
| <code>socket_connect()</code>       | Connects to a server using TCP.   |
| <code>socket_get_errno()</code>     | Gets the socket error number for the caller task.                               |
| <code>socket_gethostbyaddr()</code> | Gets a pointer to the <i>hostent</i> structure, based on the IP address.        |
| <code>socket_gethostbyname()</code> | Gets a pointer to the <i>hostent</i> structure, based on the host name.         |
| <code>socket_getopt()</code>        | Gets the value of the requested socket option.                                  |
| <code>socket_htonl()</code>         | Converts a 32 bit integer from machine byte order to network byte order.        |
| <code>socket_htons()</code>         | Converts a 16 bit integer from machine byte order to network byte order.        |
| <code>socket_inet_aton()</code>     | Converts an ASCII string to an IPv4 address.                                    |
| <code>socket_inet_ntoa()</code>     | Converts an IP address to ASCII.  |
| <code>socket_ioctl()</code>         | Accesses the FIONBIO and FIONREAD commands.                                     |
| <code>socket_listen()</code>        | Starts listening on a TCP port.   |
| <code>socket_ntohl()</code>         | Converts a 32 bit integer from network byte order to machine (host) byte order. |
| <code>socket_ntohs()</code>         | Converts a 16 bit integer from network byte order to machine (host) byte order. |
| <code>socket_open()</code>          | Allocates and initializes a new socket descriptor.                              |
| <code>socket_poll()</code>          | Waits until either timeout or a specified event occurs.                         |
| <code>socket_recv()</code>          | Receives data over a TCP connection.  |
| <code>socket_recvfrom()</code>      | Receives a UDP datagram from the port associated with the socket.               |
| <code>socket_select()</code>        | Waits until a specified receive/transmission event or exception occurs.         |
| <code>socket_send()</code>          | Sends data over a TCP connection.   |
| <code>socket_sendto()</code>        | Sends a UDP datagram to a remote host.  |
| <code>socket_setopt()</code>        | Sets the value of the requested socket option.                                  |

## socket\_accept

Use this function to start listening on a TCP port.

The call order should be: socket, bind, listen, accept.

### Format

```
int socket_accept (
    int          sockfd,
    struct sockaddr * p_client_addr,
    socklen_t *   p_addrlen )
```

### Arguments

| Argument      | Description                                      | Type        |
|---------------|--|-------------|
| sockfd        | The socket descriptor.                           | int         |
| p_client_addr | Where to write the client address.               | sockaddr *  |
| p_addrlen     | Where to write the length of the client address. | socklen_t * |

### Return values

| Return value                 | Description                       |
|------------------------------|-----------------------------------|
| Number of socket descriptor. | Successful execution.             |
| Else                         | See <a href="#">Error Codes</a> . |

## socket\_bind

Use this function to open a TCP/UDP port and associate it with the socket descriptor.

### Format

```
int socket_bind (
    int                sockfd,
    const struct sockaddr * p_my_addr,
    socklen_t         addrLen )
```

### Arguments

| Argument  | Description                          | Type       |
|-----------|--------------------------------------|------------|
| sockfd    | The socket descriptor.               | int        |
| p_my_addr | The local host address.              | sockaddr * |
| addrLen   | The length of the address structure. | socklen_t  |

### Return values

| Return value   | Description                       |
|----------------|-----------------------------------|
| SOCKET_SUCCESS | Successful execution.             |
| Else           | See <a href="#">Error Codes</a> . |

## socket\_close

Use this function to close the port (or, in the TCP case, connection) associated with the socket.

### Format

```
int socket_close ( int sockfd )
```

### Arguments

| Argument | Description            | Type |
|----------|------------------------|------|
| sockfd   | The socket descriptor. | int  |

### Return values

| Return value   | Description                       |
|----------------|-----------------------------------|
| SOCKET_SUCCESS | Successful execution.             |
| Else           | See <a href="#">Error Codes</a> . |



## socket\_connect

Use this function to connect to a server using TCP.

### Format

```
int socket_connect (
    int          sockfd,
    const struct sockaddr * p_server_addr,
    socklen_t    addrlen )
```

### Arguments

| Argument      | Description                                       | Type       |
|---------------|---|------------|
| sockfd        | The socket descriptor.                            | int        |
| p_server_addr | The structure holding the server address.         | sockaddr * |
| addrlen       | The length of the <i>p_server_addr</i> structure. | socklen_t  |

### Return values

| Return value   | Description                       |
|----------------|-----------------------------------|
| SOCKET_SUCCESS | Successful execution.             |
| Else           | See <a href="#">Error Codes</a> . |

## socket\_get\_errno

Use this function to get the socket error number for the caller task.

### Format

```
int socket_get_errno ( int * p_errno )
```

### Arguments

| Argument | Description                           | Type  |
|----------|---------------------------------------|-------|
| p_errno  | Where to write the last error number. | int * |

### Return values

| Return value   | Description              |
|----------------|--------------------------|
| SOCKET_SUCCESS | Last error is valid.     |
| SOCKET_ERROR   | No last error available. |

## socket\_gethostbyaddr

Use this function to get a pointer to the *hostent* structure, based on the IP address.

### Format

```
struct hostent * socket_gethostbyaddr (
    const void *   addr,
    int            len,
    int            type )
```

### Arguments

| Argument | Description   | Type   |
|----------|---|--------|
| addr     | A pointer to the IP address (struct <a href="#">in_addr</a> ).          | void * |
| len      | The length of the IP address (size of struct <a href="#">in_addr</a> ). | int    |
| type     | Either <a href="#">AF_INET</a> or <a href="#">AF_INET6</a> .            | int    |

### Return values

| Return value                               | Description                       |
|--|-----------------------------------|
| A pointer to the <i>hostent</i> structure. | Successful execution.             |
| NULL                                       | See <a href="#">Error Codes</a> . |

## socket\_gethostbyname

Use this function to get the pointer to the *hostent* structure, based on the host name.

### Format

```
struct hostent * socket_gethostbyname ( const char * name )
```

### Arguments

| Argument | Description           | Type   |
|----------|-----------------------|--------|
| name     | The name of the host. | char * |

### Return values

| Return value                               | Description                       |
|--|-----------------------------------|
| A pointer to the <i>hostent</i> structure. | Successful execution.             |
| NULL                                       | See <a href="#">Error Codes</a> . |

## socket\_getopt

Use this function to get the value of the requested socket option.

### Format

```
int socket_getopt(  
    int          sockfd,  
    int          level,  
    int          option_name,  
    void *       p_option_value,  
    socklen_t *  p_option_len )
```

### Arguments

| Argument       | Description   | Type        |
|----------------|---|-------------|
| sockfd         | The socket descriptor.                                | int         |
| level          | The socket option level.                              | int         |
| option_name    | The socket option identifier.                         | int         |
| p_option_value | A pointer to the buffer with the socket option value. | void *      |
| p_option_len   | A pointer to the socket option length.                | socklen_t * |

### Return values

| Return value   | Description                       |
|----------------|-----------------------------------|
| SOCKET_SUCCESS | Successful execution.             |
| Else           | See <a href="#">Error Codes</a> . |

## socket\_htonl

Use this function to convert a 32 bit integer from machine (host) byte order to network byte order.

*htons* stands for "host-to-network long".

### Format

```
unsigned int socket_htonl( unsigned int host_addr )
```

### Arguments

| Argument  | Description                    | Type         |
|-----------|--------------------------------|--------------|
| host_addr | An integer in host byte order. | unsigned int |

### Return value

The integer in network byte order.

**Note:** This function is supplied for BSD compatibility reasons. HCC converts this to a NULL macro because internally HCC handles all network address and port assignments natively.

## socket\_htons

Use this function to convert a 16 bit integer from machine (host) byte order to network byte order.

*htons* stands for "host-to-network short".

### Format

```
unsigned short socket_htons( unsigned short host_port )
```

### Arguments

| Argument  | Description                    | Type           |
|-----------|--------------------------------|----------------|
| host_port | An integer in host byte order. | unsigned short |

### Return value

The integer in network byte order.

**Note:** This function is supplied for BSD compatibility reasons. HCC converts this to a NULL macro because internally HCC handles all network address and port assignments natively.

## socket\_inet\_aton

Use this function to convert an ASCII string to an IPv4 address.

**Note:** This function does not support IPv6 addresses.

### Format

```
int socket_inet_aton (
    const char *    p_ascii,
    struct in_addr * p_addr )
```

### Arguments

| Argument | Description                                | Type      |
|----------|--|-----------|
| p_ascii  | A pointer to the string to convert.        | char *    |
| p_addr   | Where to write the resulting IPv4 address. | in_addr * |

### Return values

| Return value   | Description                       |
|----------------|-----------------------------------|
| SOCKET_SUCCESS | Successful execution.             |
| Else           | See <a href="#">Error Codes</a> . |



## socket\_inet\_ntoa

Use this function to convert an IP address to ASCII.

### Format

```
char * socket_inet_ntoa ( struct in_addr addr )
```

### Arguments

| Argument | Description                | Type                    |
|----------|----------------------------|-------------------------|
| addr     | The IP address to convert. | <a href="#">in_addr</a> |

### Return values

| Return value                                   | Description                       |
|--|-----------------------------------|
| A pointer to the ASCII form of the IP address. | Successful execution.             |
| Else   | See <a href="#">Error Codes</a> . |

## socket\_ioctl

Use this function to access the FIONBIO and FIONREAD commands.

- Use FIONBIO to set the blocking mode of the socket. If the argument pointer content is zero, this sets the port to blocking. If it is non-zero then the port is non-blocking. Note that if an RTOS is not used, the socket will always be non-blocking.
- Use FIONREAD to pass a pointer (p\_arg) to a location to store the number of bytes available to be read from the socket.

### Format

```
int socket_ioctl (
    int          sockfd,
    int          cmd,
    unsigned long * p_arg )
```

### Arguments

| Argument | Description  | Type            |
|----------|--|-----------------|
| sockfd   | The socket descriptor.   | int             |
| cmd      | FIONBIO or FIONREAD.   | int             |
| p_arg    | INPUT: a pointer to the input argument (FIONBIO).<br>OUTPUT: a pointer to the input argument (FIONREAD). | unsigned long * |

### Return values

| Return value   | Description                       |
|----------------|-----------------------------------|
| SOCKET_SUCCESS | Successful execution.             |
| Else           | See <a href="#">Error Codes</a> . |

## socket\_listen

Use this function to start listening on a TCP port.

The call order should be: socket, bind, listen.

### Format

```
int socket_listen (  
    int  sockfd,  
    int  backlog )
```

### Arguments

| Argument | Description                                    | Type |
|----------|--|------|
| sockfd   | The socket descriptor.                         | int  |
| backlog  | The maximum number of connections on the port. | int  |

### Return values

| Return value   | Description                       |
|----------------|-----------------------------------|
| SOCKET_SUCCESS | Successful execution.             |
| Else           | See <a href="#">Error Codes</a> . |

## socket\_ntohl

Use this function to convert a 32 bit integer from network byte order to machine (host) byte order.

*ntoh* stands for "network-to-host long".

### Format

```
unsigned int socket_ntohl( unsigned int net_addr )
```

### Arguments

| Argument | Description                       | Type         |
|----------|-----------------------------------|--------------|
| net_addr | An integer in network byte order. | unsigned int |

### Return value

The integer in host byte order.

**Note:** This function is supplied for BSD compatibility reasons. HCC converts this to a NULL macro because internally HCC handles all network address and port assignments natively.

## socket\_ntohs

Use this function to convert a 16 bit integer from network byte order to machine (host) byte order.

*ntohs* stands for "network-to-host short".

### Format

```
unsigned short socket_ntohs( unsigned short net_port )
```

### Arguments

| Argument | Description                       | Type           |
|----------|-----------------------------------|----------------|
| net_port | An integer in network byte order. | unsigned short |

### Return value

The integer in host byte order.

**Note:** This function is supplied for BSD compatibility reasons. HCC converts this to a NULL macro because internally HCC handles all network address and port assignments natively.

## socket\_open

Use this function to allocate and initialize a new socket descriptor.

### Format

```
int socket_open (
    int  domain,
    int  type,
    int  protocol )
```

### Arguments

| Argument | Description  | Type |
|----------|--|------|
| domain   | The socket domain; this should always be <a href="#">AF_INET</a> or <a href="#">AF_INET6</a> . | int  |
| type     | The socket type ( <a href="#">SOCK_DGRAM</a> or <a href="#">SOCK_STREAM</a> ).                 | int  |
| protocol | Not used.  | int  |

### Return values

| Return value              | Description                       |
|---------------------------|-----------------------------------|
| Socket descriptor number. | Successful execution.             |
| SOCKET_ERROR              | See <a href="#">Error Codes</a> . |

## socket\_poll

Use this function to wait until either timeout, or any of the events specified by the *p\_fds*, occurs.

### Format

```
int socket_poll (
    t_pollfd *    p_fds,
    unsigned int  nfds,
    int           timeout )
```

### Arguments

| Argument | Description   | Type         |
|----------|---|--------------|
| p_fds    | A pointer to the array of pollfd structures.                              | t_pollfd *   |
| nfds     | The number of entries in the <i>p_fds</i> array.                          | unsigned int |
| timeout  | The time in milliseconds to wait for a result.<br>Use -1 to wait forever. | int          |

### Return values

| Return value                                   | Description                       |
|--|-----------------------------------|
| The number of sockets where an event occurred. | Successful execution.             |
| SOCKET_ERROR                                   | See <a href="#">Error Codes</a> . |

## socket\_recv

Use this function to receive data over a TCP connection.

### Format

```
int socket_recv (  
    int      sockfd,  
    void *   p_buffer,  
    int      length,  
    int      flags)
```

### Arguments

| Argument | Description                      | Type   |
|----------|----------------------------------|--------|
| sockfd   | The socket descriptor.           | int    |
| p_buffer | A pointer to the data buffer.    | void * |
| length   | The size of the buffer in bytes. | int    |
| flags    | Not used.                        | int    |

### Return values

| Return value                  | Description                       |
|-------------------------------|-----------------------------------|
| The number of bytes received. | Successful execution.             |
| SOCKET_ERROR                  | See <a href="#">Error Codes</a> . |



## socket\_recvfrom

Use this function to receive a UDP datagram from the port associated with the socket.

### Format

```
int socket_recvfrom (
    int          sockfd,
    void *      p_buffer,
    int         length,
    int         flags,
    struct sockaddr * remote_address,
    socklen_t * p_addrlen )
```

### Arguments

| Argument       | Description                                      | Type        |
|----------------|--|-------------|
| sockfd         | The socket descriptor.                           | int         |
| p_buffer       | A pointer to the data buffer.                    | void *      |
| length         | The size of the buffer in bytes.                 | int         |
| flags          | Not used.  | int         |
| remote_address | Where to write the address of the remote node.   | sockaddr *  |
| p_addrlen      | Where to write the length of the remote address. | socklen_t * |

### Return values

| Return value                  | Description                       |
|-------------------------------|-----------------------------------|
| The number of bytes received. | Successful execution.             |
| SOCKET_ERROR                  | See <a href="#">Error Codes</a> . |

## socket\_select

Use this function to wait until one of the following occurs:

- Data is received on any socket in the *p\_readfds* set.
- Data is transmitted on any socket in the *p\_writefds* set.
- An exception event occurs on any socket in the *p\_exceptfds* set.

### Format

```
int socket_select (
    int                nfds,
    t_fd_set *        p_readfds,
    t_fd_set *        p_writefds,
    t_fd_set *        p_exceptfds,
    const struct timeval * p_timeout )
```

### Arguments

| Argument    | Description  | Type       |
|-------------|--|------------|
| nfds        | The number of entries in a set.  | int        |
| p_readfds   | A pointer to the set of socket descriptors to be monitored for incoming data.  | t_fd_set * |
| p_writefds  | A pointer to the set of socket descriptors to be monitored for completion of a send operation.   | t_fd_set * |
| p_exceptfds | A pointer to the set of socket descriptors to be monitored for exception events. Currently socket disconnection is the only exception event supported.   | t_fd_set * |
| p_timeout   | A pointer to a <i>timeval</i> structure. If either of the following applies, this call waits forever: <ul style="list-style-type: none"> <li>• The pointer to the <i>timeval</i> structure is NULL.</li> <li>• Both elements of the structure (seconds and microseconds) are set to -1.</li> </ul> | timeval *  |

**Return values**

| <b>Return value</b>   | <b>Description</b>                |
|---|-----------------------------------|
| The number of socket descriptors ready for input /output.<br>If this is 0 it means the timeout expired. | Successful execution.             |
| SOCKET_ERROR  | See <a href="#">Error codes</a> . |

## socket\_send

Use this function to send data over a TCP connection.

### Format

```
int socket_send (
    int          sockfd,
    const void * p_buffer,
    int          length,
    int          flags )
```

### Arguments

| Argument | Description                      | Type   |
|----------|----------------------------------|--------|
| sockfd   | The socket descriptor.           | int    |
| p_buffer | A pointer to the data buffer.    | void * |
| length   | The size of the buffer in bytes. | int    |
| flags    | Not used.                        | int    |

### Return Values

| Return value              | Description                       |
|---------------------------|-----------------------------------|
| The number of bytes sent. | Successful execution.             |
| SOCKET_ERROR              | See <a href="#">Error Codes</a> . |

## socket\_sendto

Use this function to send a UDP datagram to a remote host.

This function allocates a maximum of one protocol buffer, copies data to it, and sends it.

### Format

```
int socket_sendto (
    int                sockfd,
    const void *      p_buffer,
    int                length,
    int                flags,
    const struct sockaddr * p_dest_addr,
    socklen_t         dest_len )
```

### Arguments

| Argument    | Description   | Type       |
|-------------|---|------------|
| sockfd      | The socket descriptor.                                      | int        |
| p_buffer    | A pointer to the data to send.                              | void *     |
| length      | The number of bytes to send.                                | int        |
| flags       | Not used.   | int        |
| p_dest_addr | A pointer to the structure holding the remote host address. | sockaddr * |
| dest_len    | The length of the <i>p_dest_addr</i> structure.             | socklen_t  |

### Return values

| Return value              | Description                       |
|---------------------------|-----------------------------------|
| The number of bytes sent. | Successful execution.             |
| SOCKET_ERROR              | See <a href="#">Error Codes</a> . |

## socket\_setopt

Use this function to set the value of the requested socket option.

### Format

```
int socket_setopt(  
    int      sockfd,  
    int      level,  
    int      option_name,  
    void *   p_option_value,  
    socklen_t option_len )
```

### Arguments

| Argument       | Description   | Type      |
|----------------|---|-----------|
| sockfd         | The socket descriptor.                                | int       |
| level          | The socket option level.                              | int       |
| option_name    | The socket option identifier.                         | int       |
| p_option_value | A pointer to the buffer with the socket option value. | void *    |
| option_len     | The length of the socket option data.                 | socklen_t |

### Return values

| Return value   | Description                       |
|----------------|-----------------------------------|
| SOCKET_SUCCESS | Successful execution.             |
| Else           | See <a href="#">Error Codes</a> . |

## 5.3 IPv6 Address Checking Macros

These macros are used in IPv6 systems to check the type of an address. They are defined in section 6.7 of: [RFC 2553](#).

| Macro                    | Description  |
|--------------------------|--|
| IN6_IS_ADDR_UNSPECIFIED  | Returns a non-zero value if the address is unspecified.  |
| IN6_IS_ADDR_LOOPBACK     | Returns a non-zero value if the address is a loopback address.                                       |
| IN6_IS_ADDR_MULTICAST    | Returns a non-zero value if the address is multicast.  |
| IN6_IS_ADDR_LINKLOCAL    | Returns a non-zero value if the address is link-local, but not an IPv6 multicast link-local address. |
| IN6_IS_ADDR_SITELOCAL    | Returns a non-zero value if the address is site-local, but not an IPv6 multicast site-local address. |
| IN6_IS_ADDR_V4MAPPED     | Returns a non-zero value if the address is IPv4-mapped.  |
| IN6_IS_ADDR_V4COMPAT     | Returns a non-zero value if the address is IPv4-compatible.  |
| IN6_IS_ADDR_MC_NODELOCAL | Returns a non-zero value if the address is multicast node-local.                                     |
| IN6_IS_ADDR_MC_LINKLOCAL | Returns a non-zero value if the address is multicast link-local.                                     |
| IN6_IS_ADDR_MC_SITELOCAL | Returns a non-zero value if the address is multicast site-local.                                     |
| IN6_IS_ADDR_MC_ORGLOCAL  | Returns a non-zero value if the address is multicast organization-local.                             |
| IN6_IS_ADDR_MC_GLOBAL    | Returns a non-zero value if the address is multicast global.   |

## 5.4 Error Codes

The main return values are the following:

| Error Code     | Value | Meaning               |
|----------------|-------|-----------------------|
| SOCKET_SUCCESS | 0     | Successful execution. |
| SOCKET_ERROR   | -1    | Operation failed.     |

The following table lists all the error codes that the Sockets API calls can set in the global error field. To check an error code after a function has failed, call **socket\_get\_errno()**.

| Error Code     | Meaning                                      |
|----------------|--|
| EBADF          | Bad file descriptor.                         |
| ECONNABORTED * | Software caused connection abort.            |
| ECONNRESET *   | The connection was reset by the peer.        |
| EFAULT *       | General error.                               |
| EINVAL         | Invalid argument.                            |
| ENETDOWN *     | The network is down.                         |
| ENOTCONN *     | The transport endpoint is not connected.     |
| ETIMEDOUT      | The operation timed out.                     |
| EAGAIN         | Try again.                                   |
| ENOMEM         | No memory available.                         |
| EPERM *        | The operation is not permitted.              |
| EALREADY       | A connection request is already in progress. |
| ENOTFOUND      | Not found.                                   |

**Note:** If one of the errors marked \* occurs, the socket cannot be used so close it.



## 5.5 Types and Definitions

This section describes the main elements (structures, typedefs, and macros) that are used. These are defined in the API header files of this package and the **mip\_base** package.

### Sockets Macros and Definitions

#### Socket error value

| Return Value   | Value |
|----------------|-------|
| SOCKET_SUCCESS | 0     |
| SOCKET_ERROR   | -1    |

#### Socket domain

| Return Value | Description |
|--------------|-------------|
| AF_INET      | IPv4        |
| AF_INET6     | IPv6        |

#### IPv4 address prefix

| Return Value                     | Value | Description  |
|----------------------------------|-------|--|
| IP_V4_MAP_IP_V6_ADDR_PREFIX_SIZE | 12    | The size of the prefix in an IP_V4_ENABLE address mapped as an IPv6 address. |

#### Socket types

| Return Value | Description                |
|--------------|----------------------------|
| SOCK_STREAM  | Stream (TCP) connection.   |
| SOCK_DGRAM   | Datagram (UDP) connection. |

#### Any address

| Return Value | Description      |
|--------------|------------------|
| IN_ADDR_ANY  | Zero IP address. |

## Option levels

| Option Level Name | Description                              |
|-------------------|--|
| SOL_SOCKET        | Sets the option level to Socket options. |
| IPPROTO_IP        | Sets the option level to IP options.     |

## Socket options

| Return Value | Description  |
|--------------|--|
| SO_RCVTIMEO  | Used to set a timeout on waiting to receive data on a socket.  |
| SO_SNDTIMEO  | Used to set a timeout on waiting to transmit data on a socket.   |
| SO_PRIORITY  | Used to set the TOS (DS) field for a socket on all outgoing IP packets. This can be used for DGRAM and STREAM sockets. The option is an unsigned char that provides the TOS value to use. The current state of the option can be retrieved by using <b>socket_getopt()</b> . |
| SO_LINGER    | This is only used for SOCK_STREAM (TCP).<br><br>When <b>linger</b> is enabled, <b>socket_close()</b> waits for transmission of TCP packets to complete before disconnecting to prevent data loss.  |

## Select Macros

| Return Value          | Description  |
|-----------------------|--|
| FD_ISSET( fd, fdset ) | Checks whether the specified socket event has occurred on the specified <i>fdset</i> . |
| FD_SET( fd, fdset )   | Sets the specified socket in the <i>fdset</i> .  |
| FD_CLR( fd, fdset )   | Clears the specified socket in the <i>fdset</i> .                                      |
| FD_ZERO( fdset )      | Clears the <i>fdset</i> .  |

## IP options

| Return Value                             | Description  |
|--|--|
| IP_MULTICAST_TTL                         | Sets the TTL value to use for outgoing multicast packets on a socket. By default this is one (according to the standard). The option is an unsigned char that tells the TTL to use. This option can be retrieved by using <b>socket_getopt()</b> .   |
| IP_MULTICAST_IF                          | Sets the interface to use for sending multicast packets on a socket. The option is "struct in_addr" which contains the IP address of the local interface. If address is set to INADDR_ANY, the system automatically chooses the interface (it always uses the interface which has the default gateway). This option can be retrieved by using <b>socket_getopt()</b> .   |
| IP_ADD_MEMBERSHIP,<br>IP_DROP_MEMBERSHIP | Add/drop membership of a multicast group. The option is a "struct ip_mreq", where: <ul style="list-style-type: none"> <li>• <i>imr_multiaddr</i> is the IP address of the multicast group to subscribe.</li> <li>• <i>imr_interface</i> is the local IP address of the interface where membership needs to be added.</li> </ul> <p>If <i>imr_interface</i> is set to INADDR_ANY, the system automatically chooses the interface which is always the one with the default gateway. This option cannot be retrieved by using <b>socket_getopt()</b>.</p> |
| SO_KEEPALIVE                             | For TCP only: enables/disables TCP keep-alive mechanism.   |
| SO_BROADCAST                             | For UDP only: enables/disables broadcast datagrams.  |
| SO_REUSEADDR                             | For TCP and UDP: enables/disables SO_REUSEADDR.  |
| SO_NODELAY                               | For TCP only: enables/disables TCP NO_DELAY.   |

## Sockets ioctl commands

| Return Value | Description                    |
|--------------|--------------------------------|
| FIONBIO      | IOCTL to set blocking mode.    |
| FIONREAD     | IOCTL to request unread bytes. |

**Poll events**

| Event    | Description   |
|----------|---|
| POLLIN   | There is data to read.                                  |
| POLLPRI  | There is urgent data to read (not used).                |
| POLLOUT  | The socket has free buffer space for sending more data. |
| POLLERR  | Error condition.  |
| POLLHUP  | The connection has hung up.                             |
| POLLNVAL | Invalid request: fd not open.                           |

## Sockets Structure Definitions

### sockaddr

| Element     | Type           | Description   |
|-------------|----------------|---|
| sa_family   | unsigned short | This is always <code>AF_INET</code> in this implementation. |
| sa_data[14] | char           | Contains specific socket address data.                      |

### in\_addr

| Element | Type          | Description     |
|---------|---------------|-----------------|
| s_addr  | unsigned long | The IP address. |

### in6\_addr

| Element     | Type    | Description       |
|-------------|---------|-------------------|
| s6_addr[16] | uint8_t | The IPv6 address. |

### sockaddr\_in

| Element     | Type           | Description   |
|-------------|----------------|---|
| sin_family  | short          | This is always <code>AF_INET</code> in this implementation. |
| sin_port    | unsigned short | The port number.  |
| sin_addr    | struct in_addr | The IP address.   |
| sin_zero[8] | uint8_t        | Reserved.   |

### sockaddr\_in6

| Element       | Type            | Description  |
|---------------|-----------------|--|
| sin6_family   | short           | This is always <code>AF_INET6</code> in this implementation. |
| sin6_port     | unsigned short  | The transport layer port number.                             |
| sin6_flowinfo | unsigned short  | IPv6 flow information.                                       |
| sin6_addr     | struct in6_addr | The IPv6 address.  |
| sin6_scope_id | uint32_t        | A set of interfaces used for a scope.                        |

**ip\_mreq**

| Element       | Type           | Description                            |
|---------------|----------------|--|
| imr_multiaddr | struct in_addr | The multicast IP Address of the group. |
| imr_interface | struct in_addr | The local interface's IP address.      |

**hostent**

This is used by the **socket\_gethostbyname()** and **socket\_gethostbyaddr()** functions. See the function headers in the file **api\_ip\_socket.h**.

| Element       | Type   | Description                |
|---------------|--------|----------------------------|
| * h_name      | char   | The host name.             |
| * h_aliases   | char * | Alias list.                |
| h_addrtype    | int    | The host address type.     |
| h_length      | int    | The length of the address. |
| * h_addr_list | char * | List of addresses.         |

**t\_fd\_set**

| Element  | Type     | Description |
|--|----------|-------------|
| fdmask[( ( SOCK_TABLE_SIZE * 2U ) + 31U ) / 32U] | uint32_t |             |

**t\_pollfd**

| Element | Type           | Description            |
|---------|----------------|------------------------|
| fd      | int            | The socket descriptor. |
| events  | unsigned short | Events to wait for.    |
| revents | unsigned short | Received events.       |

**linger**

| Element  | Type | Description   |
|----------|------|---|
| l_onoff  | int  | Set this to 0 to disable lingering. Any other value enables it. |
| l_linger | int  | The number of seconds to linger for; 0 means wait forever.      |

**timeval**

| Element | Type | Description   |
|---------|------|---------------|
| tv_sec  | long | Seconds.      |
| tv_usec | long | Microseconds. |

**Note:** If both values are -1 this means forever.

**IPv6 loopback address**

```
#define IN6ADDR_LOOPBACK_INIT { { { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1 } } }
```

**IPv6 wild card address**

```
#define IN6ADDR_ANY_INIT { { { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 } } }
```

## 6 Integration

This section describes all aspects of the Sockets module that require integration with your target project.

This includes porting and configuration of external resources.

### 6.1 OS Abstraction Layer

---

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

This module uses the following OAL components:

| OAL Resource | Number Required |
|--------------|-----------------|
| Tasks        | 0               |
| Mutexes      | 1               |
| Events       | 1               |



## 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 Sockets module makes use of the following standard PSP functions:

| Function             | Package  | Element    | Description  |
|----------------------|----------|------------|--|
| <b>psp_memcmp()</b>  | psp_base | psp_string | Compares two blocks of memory.                                     |
| <b>psp_memcpy()</b>  | psp_base | psp_string | Copies a block of memory. The result is a binary copy of the data. |
| <b>psp_memset()</b>  | psp_base | psp_string | Sets the specified area of memory to the defined value.            |
| <b>psp_strncpy()</b> | psp_base | psp_string | Copies one string of defined length to another.                    |
| <b>psp_strlen()</b>  | psp_base | psp_string | Gets the length of a string.                                       |

The Sockets module makes use of the following standard PSP macros:

| Macro       | Package  | Element        | Description  |
|-------------|----------|----------------|--|
| PSP_RD_BE32 | psp_base | psp_endianness | Reads a 32 bit value stored as big-endian from 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. |