



lib_xtcp: TCP/IP library

Publication Date: 2025/11/20

Document Number: XM-007217-UG v7.0.0

IN THIS DOCUMENT

1	Introduction	3
1.1	Terms	3
2	Overview	4
2.1	TCP/IP stack	4
2.2	Ethernet MAC/PHY	4
2.3	IP Configuration	5
2.4	Events and Connections	5
2.5	Creating a socket	6
2.6	New Connections	6
2.7	Receiving Data	7
2.8	Sending Data	8
2.9	Closing Connections	10
2.10	Link Status Events	10
2.11	Server Configuration	10
2.12	XTCP Configuration	10
3	Usage	12
3.1	Using lib_xtcp	12
3.2	Example Application	12
3.3	Building the example	13
4	Configuration API	15
4.1	Configuration Defines	15
4.2	Client Callback Function	15
5	Functional API	17
5.1	Data Structures/Types	17
5.2	Event types	19
5.3	Server API	21
5.4	Client API	22



1 Introduction

This document details the XMOS TCP library **lib_xtcp** which allows use of TCP and UDP traffic over Ethernet.

The following sections of the document describe the general usage and behaviour of the library, followed by a detailed usage with an example application and then detailed descriptions of the APIs.

This document assumes familiarity with the XMOS xcore architecture, Ethernet, and TCP/IP along with the XMOS XTC toolchain and the XC language.

lib_xtcp is intended to be used with the [XCommon CMake](#), the XMOS application build and dependency management system.

This library is for use with *xcore-200* series (XS2 architecture) or *xcore.ai* series (XS3 architecture) devices, previous generations of xcore devices (i.e. XS1 architecture) are supported, but all examples and app-notes target newer devices.

1.1 Terms

The terms used in this document can appear confusing as *client* and *server* can both be used in two different ways. Firstly, for an XC interface there are clients and servers, see [Client API](#) as an example. The client being application and the server being the **lib_xtcp** stack. This usage is commonly used throughout this document. Secondly, there are TCP/IP clients and servers, these are referred to as a local host or remote host, or in context such as *DHCP server* or *HTTP server*.



2 Overview

The TCP/IP library provides OSI layer 3 and 4 features. Client applications built on **lib_xtcp** can provide layers 5-7 features as needed, such as the HTTP example provided in the library's example **app_simple_webserver**, see [Example Application](#) section.

Table 1: lib_xtcp and the OSI Layer Model

OSI Layers	Addressing	xcore libraries
Application	e.g. HTTP/URL	(lib_xtcp client) app_simple_webserver
Presentation	application specific	lib_xtcp client application
Session	application specific	lib_xtcp client application
Transport	Port	lib_xtcp
Network	IP	lib_xtcp
Data link	MAC	lib_ethernet (MAC)
Physical	PHY	lib_ethernet (PHY)

2.1 TCP/IP stack

In earlier releases of **lib_xtcp**, the TCP/IP stack used was selectable between LwIP and uIP. From version 7.0.0 onwards, only the LwIP stack is supported with 2 configurations, *standard* and *minimal*. The *standard* configuration is the default and provides better performance by having a larger memory footprint.

The TCP/IP stack runs in a task implemented in the `xtcp_lwip()` function which implements TCP/IP functionality using the [lwIP](#) stack.

2.2 Ethernet MAC/PHY

This task connects to either the RMII/RGMII MAC components or the MII component in the Ethernet library **lib_ethernet**. See the figures [Fig. 1](#) and [Fig. 2](#) and the Ethernet library user guide for details on these components.

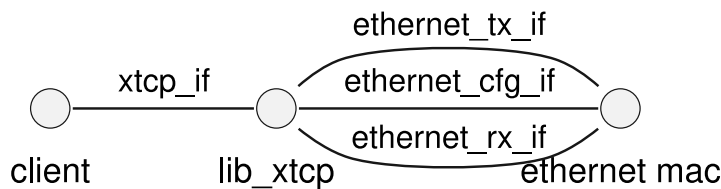


Fig. 1: XTCP task diagram

Or direct to the MII component,

Clients can interact with the TCP/IP stack via the interface of the **lib_xtcp** component using the interface functions described in [Client API](#).

Note



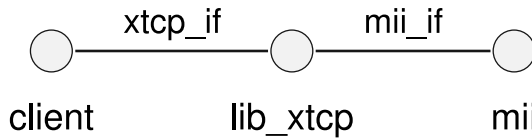


Fig. 2: XTCP task diagram (MII)

The **lib_xtcp** will only build against *real-time* MACs, due to the use of **lib_ethernet** timestamps in the TCP/IP stack.

2.3 IP Configuration

The library server will determine its IP configuration based on the **xtcp_ipconfig_t** configuration passed into the **xtcp_lwip()** task (see section [Server API](#)). If an address is supplied then that address will be used (a static IP address configuration):

```
xtcp_ipconfig_t ipconfig = {
  { 192, 168, 0, 2 }, // ip address
  { 255, 255, 255, 0 }, // netmask
  { 192, 168, 0, 1 } // gateway
};
```

To use dynamic addressing via DHCP, the **xtcp_lwip()** function can be passed a structure with an IP address that is all zeros:

```
xtcp_ipconfig_t ipconfig = {
  { 0, 0, 0, 0 }, // ip address
  { 0, 0, 0, 0 }, // netmask
  { 0, 0, 0, 0 } // gateway
};
```

Note

Note that the DHCP client will retry indefinitely until it obtains an address. This means that if there is no DHCP server on the network then the stack will not be able to send or receive any packets.

2.4 Events and Connections

The TCP/IP client application stack interface (see [Client API](#)) is a low-level event based interface. This is to allow applications to manage buffering and connections in the most efficient way possible for the application.

Each client will receive *event* notifications from the server to indicate that the server has either new data or network notifications for that client. The client then retrieves the event using the **get_event()** call. This returns the event type as a **xtcp_event_type_t**, detailed in [Event types](#), and a connection id as an out parameter. The connection id is used to identify the socket that the event relates to.

A client will typically handle its connection to the XTCP server in the following manner:

```
int32_t conn_id;
char buffer[ETHERNET_MAX_PACKET_SIZE];
unsigned data_len;
```

(continues on next page)



(continued from previous page)

```

select {
case i_xtcp.event_ready():
    xtcp_event_type_t event = i_xtcp.get_event(conn_id);
    // Handle events
    switch (event) {
        ...
        case XTCP_RECV_DATA:
            int32_t length = i_xtcp.recv(conn_id, buffer, ETHERNET_MAX_PACKET_SIZE);
            if (length > 0) {
                // process data in buffer
            } else {
                // handle error
            }
            ...
            break;
        ...
    }
    break;
}
}

```

The client can also call interface functions to initiate new connections, manage the connection and send or receive data.

If the client is handling multiple connections then the server may interleave events for each connection so the client may have to hold persistent state for each connection.

At **lib_xtcp** version 7.0.0, the interface API changed to make the data handling easier for clients. The old **xtcp_connection_t** structure is no longer used and instead the connection *id* is passed as an out parameter from the **get_event()** function and as an in parameter to the other interface functions.

Thus, the connection and event model is now different for TCP connections and UDP connections. Full details of both the possible events and possible commands can be found in [Functional API](#). See the following sections for details of handling new connections, sending and receiving data using UDP and TCP.

2.5 Creating a socket

To create a new socket, the client must call the **socket()** function with the desired protocol (TCP or UDP). This function will return a connection ID that can be used to refer to the socket in future calls.

Note

This was added in v7.0.0 and is the way to create a new socket. In previous versions, the socket was created implicitly by the interface.

2.6 New Connections

New connections are made in two different ways. Either the **connect()** function is used to initiate a connection with a remote host or the **listen()** function is used to listen on a port for remote hosts to connect to.

TCP connections

On a TCP socket, calling **connect()** will begin the process of establishing a connection and when the **XTCP_NEW_CONNECTION** event is received by the client the connection can be used. And respectively, calling **listen()** will allow the client to wait for remote host connections, when each host connects the **XTCP_ACCEPTED** event is received by the client and the connection can be used. After either event the socket can be considered *connected* by the client and send and receive data as needed. Using the functions **send()** and **recv()**.



For example, a client that wishes to listen for HTTP requests over TCP connections on port 80:

```
static const xtcp_ipaddr_t any_addr = { 0, 0, 0, 0 };
int32_t id = i_xtcp.socket(XTCP_PROTOCOL_TCP);
xtcp_error_code_t listen_result = i_xtcp.listen(id, 80, any_addr);
```

A note on handling IDs on TCP connections

When making a new connection with `connect()`, the same connection ID is used for the duration of the connection, and it will be provided in the `XTCP_NEW_CONNECTION` event, so it can be matched with that ID supplied by the original call to `socket()`.

Due to the way TCP sockets work, a single listening socket can be used to accept multiple incoming connections. Each time a new connection is *accepted*, the server will send an `XTCP_ACCEPTED` event to the client with a **new** connection ID. This ID is used in subsequent calls to send and receive data on the connection. So, the client should keep track of the connection IDs for each listening socket and each accepted connection. Calling `close()` on a listening socket will close only the listening socket.

UDP connections

On a UDP socket, after calling `connect()` there is no *connection* event and the socket can be considered *connected* by the client and send and receive data as needed. Using the functions `send()` and `recv()`. And respectively, after calling `listen()` there is no *accepted* event. However, the application simply waits for data to arrive on the socket with the `XTCP_RECV_FROM_DATA` event. Then uses the functions `recvfrom()` and `sendto()` to receive and send data.

Note

From **lib_xtcp** v7.0.0, there is no need to call `close()` after receiving data from the UDP socket. This was required in previous versions of the library. Calling `close()` on a UDP socket will close the socket and it will need to be re-created with `socket()`.

A client could create a new UDP connection to port 15333 on a host at 192.168.0.2 using:

```
xtcp_ipaddr_t addr = { 192, 168, 0, 2 };
int32_t id = i_xtcp.socket(XTCP_PROTOCOL_UDP);
xtcp_error_code_t connect_result = i_xtcp.connect(id, 15333, addr);
```

2.7 Receiving Data

When data is received for a client the server will indicate that there is a packet ready and the `get_event()` call will indicate that the event type is `XTCP_RECV_DATA` or `XTCP_RECV_FROM_DATA` and the packet data can be accessed by a call to `recv()` or `recvfrom()` respectively. For an indication of the sequence of events and calls please see Fig. 3 and Fig. 4.

Data is sent from the XTCP server to client as the UDP or TCP packets arrive from the ethernet MAC. There is buffering in the server so new incoming packets will be handled if there is sufficient memory.

On TCP connections there is a receive *window* that is used to manage the flow of data. When the window is full the TCP protocol will delay further incoming packets until there is space available in the window. The size of this window is determined by the maximum segment size (MSS), and the window is typically 4 times the MSS. In the *standard* config-



uration the MSS is 1460 bytes, so the window size is typically 5840 bytes. In the *minimal* configuration the MSS is 536 bytes, so the window size is typically 2144 bytes.

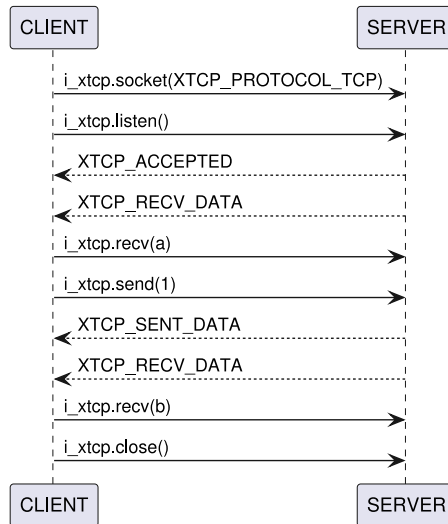


Fig. 3: Example TCP/IP receive sequence

On UDP connections there is no window, and packets are delivered to the client as they arrive. If the remote host sends packets faster than the client can process them, then packets will be dropped.

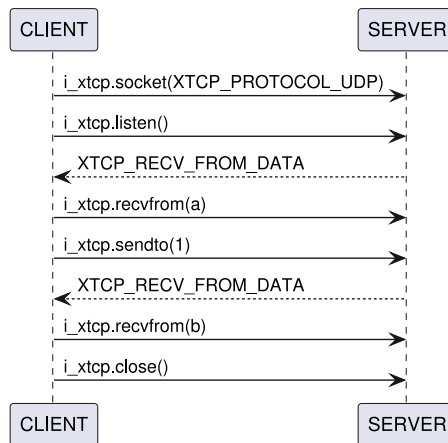


Fig. 4: Example UDP receive sequence

2.8 Sending Data

When sending data, the client is responsible for dividing the data into chunks for the server and re-transmitting the previous chunk if a transmission error occurs. Generally, the client should send data in chunks no larger than the MSS for TCP connections, and MTU (1460 bytes) for UDP connections.



The client sends a packet by calling the `send()` interface function. On TCP connections a *resend* is done by calling `send()` function with the same data buffer as the previous send.

After this data is sent to the server, two things can happen, shown in Fig. 5. Either, the server will respond with an `XTCP_SENT_DATA` event, in which case the next chunk of data can be sent. Or with an `XTCP_RESEND_DATA` event in which case the client must re-transmit the previous chunk of data.

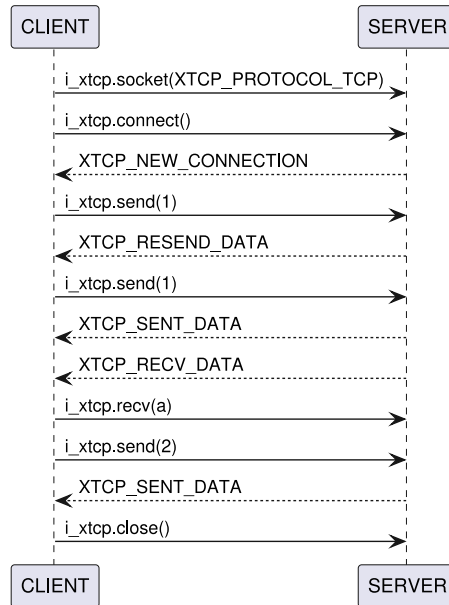


Fig. 5: Example TCP/IP send sequence

For UDP connections, there is no *sent* or *resend* events from the protocol, so the client can simply send data, see Fig. 6, if no response is received in an appropriate amount of time then the data is lost and may be retried. So, the client is responsible for any re-transmission of data if needed.

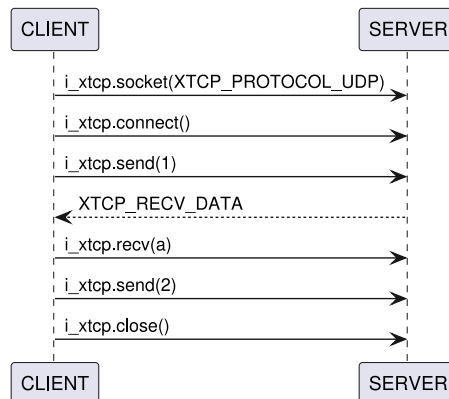


Fig. 6: Example UDP send sequence



When making UDP connections with `connect()` the client should use the `send()` function to send data as the remote host address is already known. When making UDP connections with `listen()` the client should use the `sendto()` function to specify the remote host address. This address is typically supplied by the `recvfrom()` function.

2.9 Closing Connections

When a client has finished with a connection it should call the `close()` function with the connection ID. This will close the connection and free any resources associated with it.

For TCP connections, if the remote host closes the connection then the client will receive an `XTCP_CLOSED` event and the client should then call `close()` with the connection ID.

If the client needs to immediately shutdown the connection it should call the `abort()` function with the connection ID. This will close the connection, sending a *reset* to the remote host if needed, and free any resources associated with it.

If there is a problem with the connection then the client may receive a `XTCP_TIMEOUT` event. This currently happens if there is either a timeout waiting for a response from the remote host or if the remote host resets the connection.

If the remote host aborts the connection or the LwIP stack has to recover from an error then the client will receive an `XTCP_ABORTED` event. The client does not need to call `close()`, as the resources should already be cleaned up. Thus, the client will likely receive events with an ID of -1.

2.10 Link Status Events

As well as events related to connections, the server will also send link status events to the client. The events `XTCP_IFUP` and `XTCP_IFDOWN` indicate to a client when the link goes up or down.

The connection ID should be ignored for these events, as the event relates to the network interface and not a connection.

When the link goes down all existing TCP data connections will be closed by the server, and the client should clean up as needed. Listening sockets may be left open, but any active connections will be closed.

2.11 Server Configuration

The server is configured via arguments passed to server task, see section [Server API](#) (`xtcp_lwip()`) and the defines described in section [Configuration Defines](#).

Note

The `lib_xtcp` will only build against *real-time* MACs, due to the use of `lib_ethernet` timestamps in the TCP/IP stack.

2.12 XTCP Configuration

The underlying stack configuration can be modified by including optional header files in the application. One or both of the following, these will override the LwIP build settings. See [Configuration Defines](#).



- ▶ xtcp_client_conf.h
- ▶ xtcp_conf.h



3 Usage

3.1 Using lib_xtcp

lib_xtcp is intended to be used with the [XCommon CMake](#), the XMOS application build and dependency management system.

To use this library, include **lib_xtcp** in the application's `APP_DEPENDENT_MODULES` list in *CMakeLists.txt*, for example:

```
set(APP_DEPENDENT_MODULES "lib_xtcp")
```

All functions and types can be found in the `xtcp.h` header file:

```
#include <xtcp.h>
```

3.2 Example Application

The `app_simple_webserver` example is provided to show how the library can use TCP traffic for a very simple HTTP server.

The example targets the **XK-ETH-316-DUAL** dev-kit and 100BASE-T ethernet with an RMII PHY.

For an example of using **lib_xtcp** on the **XCORE-200** with Gigabit Ethernet, say the **XCORE-200-EXPLORER** dev-kit, see the example application note **AN02044**.

The **lib_xtcp** uses a third-party TCP/IP stack, the **LwIP** stack. This is built automatically when the library is built. **lib_xtcp** uses one thread to run the TCP/IP stack and uses around 50 kB of code and 40 kB of data, and the application client runs in another thread. The memory usage will vary depending on the configuration of the stack and the application.

By default the IP address for the xcore will be a static IP address, please update the IPv4 address to match your network, in the first row of `ipconfig` and the subnet mask to the second row, the subnet mask is typically `{ 255, 255, 255, 0 }`. Otherwise, to set the IP address automatically assigned via DHCP, fill `xtcp_ipconfig_t ipconfig = { ... }` in `main.xc` with zeros. For details please see section [IP Configuration](#)

The excerpt from the example web server shown below shows how to configure the **lib_xtcp** server with the application client here as `xhttpd`

```
int main(void) {
    xtcp_if i_xtcp[NUM_XTCP_CLIENTS];
    ethernet_cfg_if i_cfg[NUM_CFG_CLIENTS];
    ethernet_rx_if i_rx[NUM_ETH_CLIENTS];
    ethernet_tx_if i_tx[NUM_ETH_CLIENTS];
    smi_if i_smi;

    par {
        on tile[0]: rmi_ethernet_rt_mac( i_cfg, NUM_CFG_CLIENTS,
                                         i_rx, NUM_ETH_CLIENTS,
                                         i_tx, NUM_ETH_CLIENTS,
                                         null, null,
                                         p_phy_clk,
                                         p_phy_rxd,
                                         null,
                                         USE_UPPER_2B,
                                         p_phy_rxdv,
                                         p_phy_txen,
                                         p_phy_txd,
                                         null,
                                         USE_UPPER_2B,
                                         phy_rxcclk,
                                         phy_txclk,
                                         get_port_timings(PHY0_PORT_TIMINGS),
                                         ETH_RX_BUFFER_SIZE_WORDS, ETH_RX_BUFFER_SIZE_WORDS,
                                         ETHERNET_DISABLE_SHAPER);
    }
}
```

(continues on next page)



(continued from previous page)

```

on tile[1]: dual_ethernet_phy_driver(i_smi, i_cfg[CFG_TO_PHY_DRIVER], null);

// SMI/ethernet phy driver
on tile[1]: smi(i_smi, p_smi_mdio, p_smi_mdc);

// TCP component
on tile[0]: xtcp_lwip(i_xtcp, NUM_XTCP_CLIENTS, null,
                    i_cfg[CFG_TO_XTCP], i_rx[ETH_TO_XTCP], i_tx[ETH_TO_XTCP],
                    ipconfig);

// HTTP server application
on tile[0]: xhttpd(i_xtcp[XTCP_TO_HTTP]);
}
return 0;
}

```

The function `xhttpd()`, called from main will listen for a TCP connection on port 80 and shows an example of handling the events and data flowing to and from the TCP stack. For details please see section [Events and Connections](#) and the notifications are defined in [Event types](#).

```

void xhttpd(client xtcp_if i_xtcp)
{
    debug_printf("**WELCOME TO THE SIMPLE WEBSERVER DEMO**\n");

    // Initiate the HTTP state
    httpd_init(i_xtcp);

    // Loop forever processing TCP events
    while(1) {
        char rx_buffer[RX_BUFFER_SIZE];
        unsigned data_len;
        int32_t conn_id;

        select {
            case i_xtcp.event_ready():
                const xtcp_event_type_t event = i_xtcp.get_event(conn_id);
                switch (event) {
                    case XTCP_EVENT_NONE:
                        // No event to process
                        break;

                    case XTCP_IFUP:
                        xtcp_ipconfig_t ipconfig = i_xtcp.get_netif_ipconfig(0);
                        debug_printf("IP Address: %d.%d.%d.%d\n", ipconfig.ipaddr[0], ipconfig.ipaddr[1], ipconfig.ipaddr[2],
↪ ipconfig.ipaddr[3]);
                        break;

                    case XTCP_IFDOWN:
                        debug_printf("IFDOWN\n");
                        break;

                    case XTCP_ACCEPTED:
                        httpd_init_state(i_xtcp, conn_id);

```

3.3 Building the example

This section assumes that the [XMOSES XTC Tools](#) have been downloaded and installed. The required version is specified in the accompanying [README](#).

Installation instructions can be found [here](#).

Special attention should be paid to the section on [Installation of Required Third-Party Tools](#).

The application is built using the [xcommon-cmake](#) build system, which is provided with the XTC tools and is based on [CMake](#).

The `lib_xtcp` software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:



```
cd lib_xtcp
cd examples/app_simple_webserver
cmake -B build -G "Unix Makefiles"
xmake -j -C build
```

Once built run with,

```
xrun --xscope bin/app_simple_webserver.xe
```

Alternatively, the application can be programmed into flash memory for standalone execution:

```
xflash bin/app_simple_webserver.xe
```

When running and with the dev-kit connected to the same network as the computer, open a browser window and enter the address printed on the xrun terminal. The browser will display a short message, "Hello World!".



4 Configuration API

4.1 Configuration Defines

Configuration defines can either be overridden by adding the file `xtcp_conf.h` into the application and then putting `#define` directives into that header file (which will then be read by the library on build).

XTCP_HOSTNAME

Ethernet network interface hostname. By default "lwip-xcore". Option `LWIP_NETIF_HOSTNAME=1` required in `lwipopts.h` to use client host name.

MAX_XTCP_CLIENTS

Maximum number of connected XTCP clients. Used by the interface in allocating resources. Default is 5.

CLIENT_QUEUE_SIZE

Maximum number of events in a client queue. Used for allocating resources for retaining and passing notification events to the clients. Default is 20.

LwIP Configuration

There are 2 predefined `lwipopts.h` header files provided with the library, *standard* and *minimal*, which indicates the memory resource usage of each. The *standard* configuration is the default and provides better performance by having a larger memory footprint. To override the default configuration add the CMake define to the project:

```
set(LWIP_OPTS_PATH <relative-path-to-lwipopts.h>)
```

Path is relative to the `lib_xtcp/lib_xtcp` folder path not the client application. So, the path may need to start with `../../<lwipopts-h-path>` to go up one or more folders.

4.2 Client Callback Function

```
void xtcp_configure_mac(
    unsigned netif_id, uint8_t mac_address[MACADDR_NUM_BYTES],
)
```

Configure the MAC address for a given network interface. Define in the client application to provide a MAC address.

This function is called by `xtcp_lwip()` during initialization to set the MAC address for the network interface.

Note

This is a weak function that must be overridden by the user to provide a custom MAC address configuration.

Warning



This function is called from the `xtcp_lwip()` task and may not be on the same tile as the client application. Do not use shared memory or resources in this function.

Parameters

- ▶ **netif_id** – The network interface ID to configure the MAC address for. Can be ignored for now as only 1 netif is currently supported.
- ▶ **mac_address** – The six-octet MAC address output parameter to set for the given network interface.



5 Functional API

See [Usage](#) section and [Example Application](#) for details on usage of the following.

5.1 Data Structures/Types

typedef uint8_t **xtcp_ipaddr_t**[4]

XTCP IP address.

This data type represents a single ipv4 address in the XTCP stack.

struct **xtcp_host_t**

XTCP host's address.

This data type represents the address of a host in the XTCP, with an IP address and port number.

struct **xtcp_ipconfig_t**

IP configuration information structure.

This structure describes IP configuration for a network interface. With an IP address, netmask and gateway.

enum **xtcp_protocol_t**

XTCP protocol type.

This determines what type a connection is: either UDP or TCP.

Values:

enumerator **XTCP_PROTOCOL_NONE**

No Protocol

enumerator **XTCP_PROTOCOL_TCP**

Transmission Control Protocol

enumerator **XTCP_PROTOCOL_UDP**

User Datagram Protocol

enum **xtcp_error_code_t**

XTCP error codes.

This type represents the error codes that can be returned by various XTCP functions.

Values:

enumerator **XTCP_SUCCESS**

Success

enumerator **XTCP_EINVAL**

Invalid argument

enumerator **XTCP_ENOMEM**

Out of memory



enumerator **XTCP_EAGAIN**

Resource temporarily unavailable

enumerator **XTCP_EPROTONOSUPPORT**

Protocol not supported

enumerator **XTCP_EINUSE**

Address in use



5.2 Event types

enum **xtcp_event_type_t**

XTCP event type.

The event type represents what event is occurring on a particular connection. It is created by calling `socket()` and accessed by `get_event()` and after `event_ready()`.

Values:

enumerator **XTCP_EVENT_NONE**

No event

enumerator **XTCP_NEW_CONNECTION**

This event represents a new connection has been made. For TCP client connections it occurs when a session is set up with the remote host.

enumerator **XTCP_ACCEPTED**

This event occurs when a listening TCP socket has received a connection request from a remote host.

enumerator **XTCP_RECV_DATA**

This event occurs when the connection has received some data. Call `recv()` to access the data.

enumerator **XTCP_RECV_FROM_DATA**

This event occurs when the connection has received some data from a remote host, UDP only. Call `recvfrom()` to access the data and address of remote host.

enumerator **XTCP_SENT_DATA**

This event occurs when the server has successfully sent the previous piece of TCP data that was given to it via a call to `send()`.

enumerator **XTCP_RESEND_DATA**

This event occurs when the local host has failed to send the previous piece of data that was given to it via a call to `send()`. The stack is now requesting for the same data to be sent again.

enumerator **XTCP_TIMED_OUT**

This event occurs when the connection request has timed out or been reset by the remote host (TCP only). This event represents the closing of a connection and is the last event that will occur on an active connection.

enumerator **XTCP_ABORTED**

This event occurs when the connection has been aborted by the local or remote host (TCP only). This event represents the closing of a connection and is the last event that will occur on an active connection.

enumerator **XTCP_CLOSED**

This event occurs when the connection has been closed by the local or remote host, TCP only. This event represents the closing of a connection and is the last event that will occur on an active connection.



enumerator **XTCP_IFUP**

This event occurs when the link goes up (with valid new ip address). This event has no associated connection.

enumerator **XTCP_IFDOWN**

This event occurs when the link goes down. This event has no associated connection.

enumerator **XTCP_DNS_RESULT**

This event occurs when the XTCP connection has a DNS result for a request. There is no connection associated with this event, so the "id" returned by [get_event\(\)](#) is the DNS return code as a `xtcp_error_code_t`. XTCP_SUCCESS for successful resolution. XTCP_EINVAL for invalid argument. XTCP_ENOMEM for DNS request failed.



5.3 Server API

```
void xtcp_lwip(
    SERVER_INTERFACE_ARRAY(xtcp_if, i_xtcp, n_xtcp), static_const_unsigned
    n_xtcp, NULLABLE_CLIENT_INTERFACE(mii_if, i_mii), NULLABLE_CLIENT_INTERFACE(ethernet_cfg_if,
    i_eth_cfg), NULLABLE_CLIENT_INTERFACE(ethernet_rx_if,
    i_eth_rx), NULLABLE_CLIENT_INTERFACE(ethernet_tx_if,
    i_eth_tx), REFERENCE_PARAM(xtcp_ipconfig_t, ipconfig),
)
```

Function implementing the TCP/IP stack using the lwIP stack.

This functions implements a TCP/IP stack that clients can access via an interface.

Parameters

- ▶ **i_xtcp** – The interface array to connect to the clients.
- ▶ **n_xtcp** – The number of clients to the task.
- ▶ **i_mii** – If this component is connected to the mii() component in the Ethernet library then this interface should be used to connect to it. Otherwise it should be set to null.
- ▶ **i_eth_cfg** – If this component is connected to an MAC component in the Ethernet library then this interface should be used to connect to it. Otherwise it should be set to null.
- ▶ **i_eth_rx** – If this component is connected to an MAC component in the Ethernet library then this interface should be used to connect to it. Otherwise it should be set to null.
- ▶ **i_eth_tx** – If this component is connected to an MAC component in the Ethernet library then this interface should be used to connect to it. Otherwise it should be set to null.
- ▶ **ipconfig** – This **xtcp_ipconfig_t** structure is used to determine the IP address configuration of the component.



5.4 Client API

group **Xtcp_if**

The client interface for lib_xtcp

Functions

void **event_ready()**

Notifies the client that there is data/information ready for them.

After this notification is raised a call to [get_event\(\)](#) is needed.

[xtcp_event_type_t](#) **get_event**(REFERENCE_PARAM(int32_t, id))

Receive information/data from the XTCP server.

After the client is notified by [event_ready\(\)](#) it must call this function to receive the event from the server.

Note

When receiving a new connection event on a TCP socket, the id will denote the new connection socket not the listening socket's ID.

Parameters

- **id** – Output parameter for the connection descriptor the event occurred on.

Returns

The event type produced on the given connection.

int32_t **socket**([xtcp_protocol_t](#) protocol)

Create an xtcp socket.

See also

[close\(\)](#)

Parameters

- **protocol** – The protocol for any communication over the returned connection.

Returns

A xtcp connection descriptor that will be used to refer to this socket in future calls.

void **close**(int32_t id)

Close a connection.

May still receive data on a TCP connection after this call, until the close session hand-shake has completed with the remote-host. Use [abort\(\)](#) if you wish to stop all data immediately.

If this TCP socket was a remote-host data connection on a listening socket, it will continue to listen on the assigned port unless closing the listening socket itself.

If this TCP socket was a local-host triggered connection, it will close the connection.



If this is a UDP socket, it will close the connection and stop listening on the assigned port.

See also

[socket\(\)](#) and [abort\(\)](#)

Note

The id will become invalid after this call, so it should not be used.

Parameters

- **id** – The connection descriptor to act on.

void **abort**(int32_t id)

Abort a connection.

For UDP this is the same as closing the connection. For TCP the server will send a RST signal and stop all incoming data, before closing the socket.

Note

id will become invalid after this call, so it should not be used.

Parameters

- **id** – The connection descriptor to act on.

xtcp_error_code_t **listen**(
int32_t id, uint16_t port_number, *xtcp_ipaddr_t* ipaddr,
)

Listen to a particular incoming port.

After this call, when a TCP connection is established by a remote-host an XTCP_ACCEPTED event is signalled.

For a UDP socket, this will bind on the specified port allowing incoming packets. Any data received will be passed as a data received event, XTCP_RECV_FROM_DATA.

See also

[close\(\)](#)

Note

The XTCP_ACCEPTED event will create a new socket ID with the connection details of the remote-host. The original listening socket remains valid and active.



Note

For UDP connections, no XTCP_ACCEPTED event will be generated.

Parameters

- ▶ **id** – The connection descriptor to act on.
- ▶ **port_number** – The local port number to listen to.
- ▶ **ipaddr** – The address of the local host.

Returns

XTCP_SUCCESS if successful, XTCP_EINVAL if invalid parameters are provided. Also, XTCP_EINUSE if the connection is already active, and may require a TCP timeout before using again.

```
xtcp_error_code_t connect(
    int32_t id, uint16_t port_number, xtcp_ipaddr_t ipaddr,
)
```

Attempt to connect to a remote port.

For TCP this will initiate the remote-host handshake. When the handshake is complete an XTCP_NEW_CONNECTION event will be signalled.

For UDP this will assign a local port and bind the remote address of the connection to the host specified. This sends no network traffic and no event is generated. So, it can be considered **connected** immediately.

Parameters

- ▶ **id** – The connection descriptor to act on.
- ▶ **port_number** – The remote port to associate with the connection.
- ▶ **ipaddr** – The address of the remote host.

Returns

XTCP_SUCCESS if successful, XTCP_EINVAL if invalid parameters are provided.

```
int32_t send(int32_t id, const uint8_t buffer[length], uint32_t length)
```

Send data to the connection.

Parameters

- ▶ **id** – The connection descriptor to act on.
- ▶ **buffer** – An array of data to be transmitted on the network.
- ▶ **length** – The length of data to send. If this is 0, no data will be sent and a XTCP_SENT_DATA event will not occur.

Returns

The number of bytes accepted by xtcp or a negative xtcp_error_code_t. XTCP_EINVAL if invalid parameters are provided.

```
int32_t send_timed(
    int32_t id, const uint8_t buffer[length], uint32_t
    length, REFERENCE_PARAM(uint32_t, ts),
)
```

Send data to the connection.

Parameters

- ▶ **id** – The connection descriptor to act on.
- ▶ **buffer** – An array of data to be transmitted on the network.
- ▶ **length** – The length of data to send. If this is 0, no data will be sent and a XTCP_SENT_DATA event will not occur.



- **ts** – The packet transmit timestamp.

Returns

The number of bytes accepted by xtcp or a negative xtcp_error_code_t. XTCP_EINVAL if invalid parameters are provided.

```
int32_t sendto(
    int32_t id, const uint8_t buffer[length], uint32_t length, xtcp_ipaddr_t re-
    mote_addr, uint16_t remote_port,
)
```

Send data to the connection.

Parameters

- **id** – The connection descriptor to act on.
- **buffer** – An array of data to be transmitted on the network.
- **length** – The length of data to send. If this is 0, no data will be sent and a XTCP_SENT_DATA event will not occur.
- **remote_addr** – The address of the remote host.
- **remote_port** – The remote port of the remote host.

Returns

The number of bytes accepted by xtcp or an xtcp_error_code_t. XTCP_EINVAL if invalid parameters are provided.

```
int32_t sendto_timed(
    int32_t id, const uint8_t buffer[length], uint32_t length, xtcp_ipaddr_t re-
    mote_addr, uint16_t remote_port, REFERENCE_PARAM(uint32_t, ts),
)
```

Send timestamped data to the connection.

Parameters

- **id** – The connection descriptor to act on.
- **buffer** – An array of data to be transmitted on the network.
- **length** – The length of data to send. If this is 0, no data will be sent and a XTCP_SENT_DATA event will not occur.
- **remote_addr** – The address of the remote host.
- **remote_port** – The remote port of the remote host.
- **ts** – The packet transmit timestamp.

Returns

The number of bytes accepted by xtcp or an xtcp_error_code_t. XTCP_EINVAL if invalid parameters are provided.

```
int32_t recv(int32_t id, uint8_t buffer[length], uint32_t length)
```

Receive data on a connection.

Copies data from an internal buffer to the given buffer, if data is available.

Parameters

- **id** – The connection descriptor to act on.
- **buffer** – The destination buffer where received data will be stored.
- **length** – The length of the given buffer and the maximum amount of data that will be copied.

Returns

Either the total number of bytes copied to the given buffer or an xtcp_error_code_t. XTCP_EINVAL if invalid parameters are provided. XTCP_EAGAIN if the send buffer is less than the length of the data received.



```
int32_t recv_timed(
    int32_t id, uint8_t buffer[length], uint32_t length, REFERENCE_PARAM(uint32_t,
    ts),
)
```

Receive timestamped data on a connection.

Copies data from an internal buffer to the given buffer, if data is available.

Parameters

- ▶ **id** – The connection descriptor to act on.
- ▶ **buffer** – The destination buffer where received data will be stored.
- ▶ **length** – The length of the given buffer and the maximum amount of data that will be copied.
- ▶ **ts** – The packet receive timestamp.

Returns

Either the total number of bytes copied to the given buffer or an `xtcp_error_code_t`. `XTCP_EINVAL` if invalid parameters are provided. `XTCP_EAGAIN` if the send buffer is less than the length of the data received.

```
int32_t recvfrom(
    int32_t id, uint8_t buffer[length], uint32_t length, REFERENCE_PARAM(xtcp_ipaddr_t,
    ipaddr), REFERENCE_PARAM(uint16_t, port_number),
)
```

Receive data on a connection, remote host and port.

Copies data from an internal buffer to the given buffer, if data is available.

Parameters

- ▶ **id** – The connection descriptor to act on.
- ▶ **buffer** – The destination buffer where received data will be stored.
- ▶ **length** – The length of the given buffer and the maximum amount of data that will be copied.
- ▶ **port_number** – The remote port buffer data was received from.
- ▶ **ipaddr** – The address of the remote host.

Returns

Either the total number of bytes copied to the given buffer or an `xtcp_error_code_t`. `XTCP_EINVAL` if invalid parameters are provided. `XTCP_EAGAIN` if the send buffer is less than the length of the data received.

```
int32_t recvfrom_timed(
    int32_t id, uint8_t buffer[length], uint32_t length, REFERENCE_PARAM(xtcp_ipaddr_t,
    ipaddr), REFERENCE_PARAM(uint16_t, port_number), REFERENCE_PARAM(uint32_t,
    ts),
)
```

Receive timestamped data on a connection, remote host and port.

Copies data from an internal buffer to the given buffer, if data is available.

Parameters

- ▶ **id** – The connection descriptor to act on.
- ▶ **buffer** – The destination buffer where received data will be stored.
- ▶ **length** – The length of the given buffer and the maximum amount of data that will be copied.



- ▶ **port_number** – The remote port buffer data was received from.
- ▶ **ipaddr** – The address of the remote host.
- ▶ **ts** – The packet receive timestamp.

Returns

Either the total number of bytes copied to the given buffer or an `xtcp_error_code_t`. `XTCP_EINVAL` if invalid parameters are provided. `XTCP_EAGAIN` if the send buffer is less than the length of the data received.

xtcp_ipconfig_t **get_netif_ipconfig**(int32_t netif_id)

Fill the provided ipconfig address with the current state of the interface.

Parameters

- ▶ **netif_id** – The network interface ID to get the IP config for.

Returns

The current IP configuration of the interface.

xtcp_host_t **get_ipconfig_remote**(int32_t id)

Fill the provided ipconfig address with the current remote host for the connection.

Note

For UDP connections this will be unset unless `connect()` has been called.

Parameters

- ▶ **id** – The connection descriptor

Returns

The current remote host for the connection.

xtcp_host_t **get_ipconfig_local**(int32_t id)

Fill the provided ipconfig address with the current local host for the connection.

Parameters

- ▶ **id** – The connection descriptor

Returns

The current local host for the connection.

int32_t **set_connection_client_data**(int32_t id, void *data)

Allows the client to record additional data alongside the connection.

Parameters

- ▶ **id** – The connection descriptor to set the state for.
- ▶ **data** – A pointer to the additional data to associate with the connection.

Returns

`XTCP_SUCCESS` on success, or a negative error code on failure.

void ***get_connection_client_data**(int32_t id)

Allows the client to retrieve additional data alongside the connection.

Parameters

- ▶ **id** – The connection descriptor to set the state for.

Returns

Pointer to the additional data associated with the connection, or NULL if not set.



void **join_multicast_group**(*xtcp_ipaddr_t* addr)

Subscribe to a particular IP multicast group address.

Parameters

- **addr** – The address of the multicast group to join. It is assumed that this is a multicast IP address.

void **leave_multicast_group**(*xtcp_ipaddr_t* addr)

Unsubscribe from a particular IP multicast group address.

Parameters

- **addr** – The address of the multicast group to leave. It is assumed that this is a multicast IP address.

```
xtcp_host_t request_host_by_name(
    const uint8_t  hostname[len], static_const_unsigned len, xtcp_ipaddr_t
    dns_server,
)
```

Request a host's IP address from its pretty name.

Note

This is a non-blocking call. The result of the lookup will be indicated by an XTCP_DNS_RESULT event. If the event returns XTCP_SUCCESS then this function should be called a second time and the returned address is that of the host requested.

Parameters

- **hostname** – The human readable host name, e.g. "www.xmos.com"
- **len** – Length of hostname string
- **dns_server** – IP address of DNS server to query

Returns

The remote host for the hostname, or an empty IP address on error if the lookup failed or the request is in progress.

```
xtcp_error_int32_t getsockopt(
    int32_t id, xtcp_socket_level_t level, uint32_t option, uint8_t
    value[length], static_const_unsigned length,
)
```

Get a socket option.

Parameters

- **id** – The connection descriptor to act on.
- **level** – The protocol level to operate on.
- **option** – The socket option.
- **value** – The option value.
- **length** – The option value length.

Returns

XTCP_SUCCESS if successful, XTCP_EINVAL if invalid parameters are provided.

```
xtcp_error_code_t setsockopt(
    int32_t id, xtcp_socket_level_t level, uint32_t option, const uint8_t
    value[length], static_const_unsigned length,
)
```



Set a socket option.

Parameters

- ▶ **id** – The connection descriptor to act on.
- ▶ **level** – The protocol level to operate on.
- ▶ **option** – The socket option.
- ▶ **value** – The option value.
- ▶ **length** – The option value length.

Returns

XTCP_SUCCESS if successful, XTCP_EINVAL if invalid parameters are provided.

int **is_ifup**(void)

Query if the underlying interface is up.

Return values

- ▶ **1** – if the underlying interface is up,
- ▶ **0** – if the underlying interface is down.



Copyright © 2025, All Rights Reserved.

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

