# Xilinx Standalone Library Documentation

## *LwIP 2.1.1 Library*

**UG650 (v2021.2) October 13, 2021**

XILINX®

# Table of Contents

XILINX

Chapter 1

# Introduction

The lwIP is an open source TCP/IP protocol suite available under the BSD license. The lwIP is a standalone stack; there are no operating systems dependencies, although it can be used along with operating systems. The lwIP provides two A05PIs for use by applications:

- RAW API: Provides access to the core lwIP stack.
- Socket API: Provides a BSD sockets style interface to the stack.

The lwip211_v1.6 is built on the open source lwIP library version 2.1.1. The lwip211 library provides adapters for the Ethernetlite (axi_ethernetlite), the TEMAC (axi_ethernet), and the Gigabit Ethernet controller and MAC (GigE) cores. The library can run on MicroBlaze™, Arm® Cortex-A9, Arm Cortex®-A53, Arm Cortex-A72, and Arm Cortex-R5F processors. The Ethernetlite and TEMAC cores apply for MicroBlaze systems. The Gigabit Ethernet controller and MAC (GigE) core is applicable only for Arm Cortex-A9 system (Zynq®-7000 processor devices) and Arm Cortex-A53 & Arm Cortex-R5F system (Zynq® UltraScale+™ MPSoC), and Arm Cortex-A72 and Arm Cortex-R5F system (Versal™ ACAP).

## Features

The lwIP provides support for the following protocols:

- Internet Protocol (IP)
- Internet Control Message Protocol (ICMP)
- User Datagram Protocol (UDP)
- TCP (Transmission Control Protocol (TCP)
- Address Resolution Protocol (ARP)
- Dynamic Host Configuration Protocol (DHCP)
- Internet Group Message Protocol (IGMP)

# References

- FreeRTOS: http://www.freertos.org/
- LwIP Wiki
- Xilinx LwIP designs and application examples: *LightWeight IP Application Examples* (XAPP1026)
- LwIP examples using RAW and Socket APIs: http://savannah.nongnu.org/projects/lwip/
- FreeRTOS port for Zynq®-7000 devices is available for download from the FreeRTOS website.

XILINX

Chapter 2

# Using lwIP

## Overview

The following are the key steps to use lwIP for networking:

1. Creating a hardware system containing the processor, ethernet core, and a timer. The timer and ethernet interrupts must be connected to the processor using an interrupt controller.
2. Configuring lwip211_v1.6 to be a part of the software platform. For operating with lwIP socket API, the Xilkernel library or FreeRTOS BSP is a prerequisite. See the Note below.

***Note*:** The Xilkernel library is available only for MicroBlaze systems. For Cortex-A9 based systems (Zynq devices), Cortex-A53 or Cortex-R5F based systems (Zynq UltraScale+ MPSoC), and Arm Cortex-A72 and Arm Cortex-R5F system (Versal ACAP). There is no support for Xilkernel. Instead, use FreeRTOS. A FreeRTOS BSP is available for Zynq, Zynq UltraScale+ MPSoC, and Versal systems and must be included for using lwIP socket API. The FreeRTOS BSP for Zynq devices, Zynq UltraScale+ MPSoC, and Versal ACAP is available for download from the FreeRTOS website.

## Setting up the Hardware System

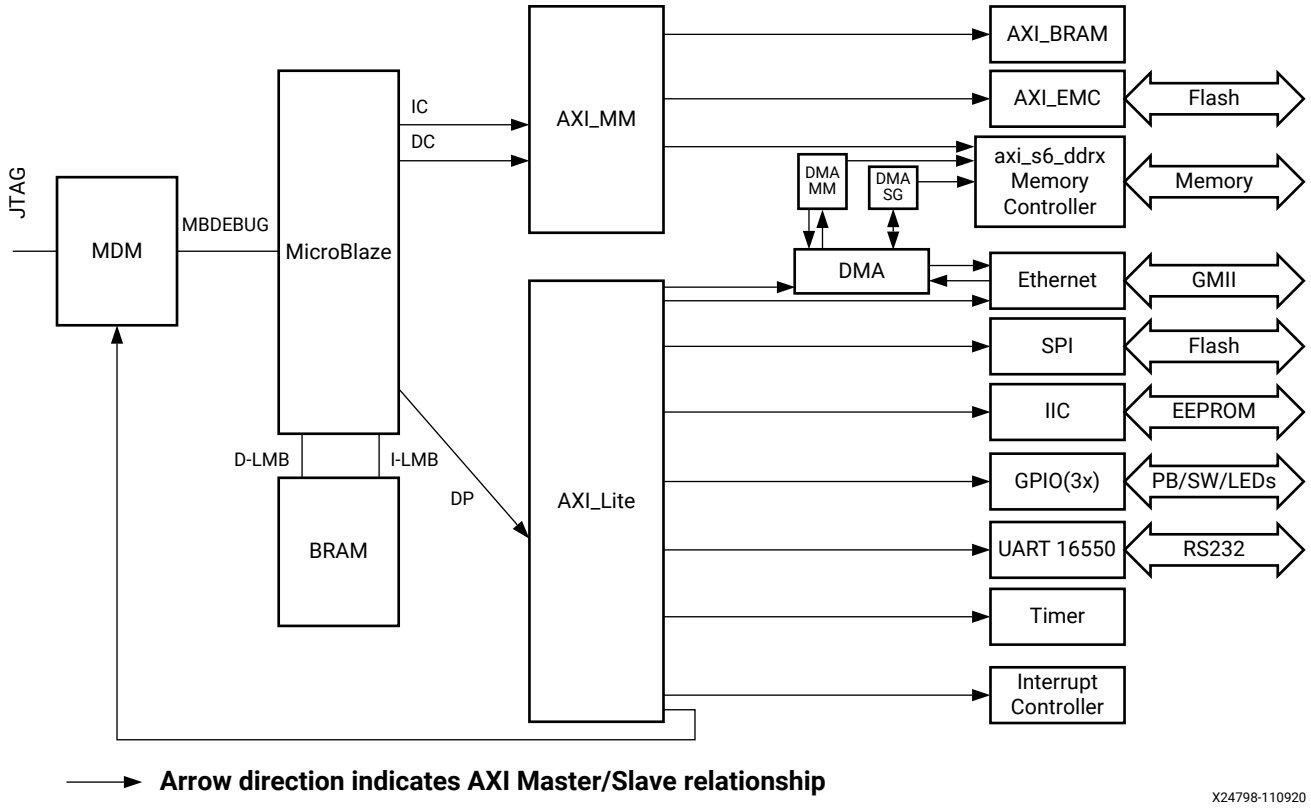
This section describes the hardware configurations supported by lwIP. The key components of the hardware system include:

- **Processor:** Either a MicroBlaze or a Cortex-A9 or a Cortex-A53 or a Cortex-R5F processor or a Cortex-A72. The Cortex-A9 processor applies to Zynq systems. The Cortex-A53 and Cortex-R5F processors apply to Zynq UltraScale+ MPSoC systems. The Cortex-A72 and Cortex-R5F processors apply to Versal ACAP systems.
- **MAC:** LwIP supports axi_ethernetlite, axi_ethernet, and Gigabit Ethernet controller and MAC (GigE) cores.
- **Timer:** To maintain TCP timers, lwIP raw API based applications require that certain functions are called at periodic intervals by the application. An application can do this by registering an interrupt handler with a timer.

- **DMA:** For axi_ethernet based systems, the axi_ethernet cores can be configured with a soft DMA engine (AXI DMA and MCDMA) or a FIFO interface. For GigE-based Zynq devices, Zynq UltraScale+ MPSoC, and Versal ACAP systems, there is a built-in DMA and so no extra configuration is needed. Same applies to axi_ethernetlite based systems, which have their built-in buffer management provisions.

The following figure shows a sample system architecture with a Kintex-6 device using the axi_ethernet core with DMA.

*Figure 1:* **System Architecture using axi_ethernet core with DMA**




# Setting up the Software System

To use lwIP in a software application, you must first compile the lwIP library as a part of the software application.

1. Click File > New > Platform Project.
2. Click Specify to create a new Hardware Platform Specification.
3. Provide a new name for the domain in the Project name field if you wish to override the default value.

4. Select the location for the board support project files. To use the default location, as displayed in the Location field, leave the Use default location check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.
5. From the Hardware Platform drop-down choose the appropriate platform for your application or click the New button to browse to an existing Hardware Platform.
6. Select the target CPU from the drop-down list.
7. From the Board Support Package OS list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click Finish. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select Project > Build Automatically to automatically build the board support package. The Board Support Package Settings dialog box opens. Here you can customize the settings for the domain.
10. Click OK to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click platform.spr file and select the appropriate domain/board support package. The overview page opens.
12. In the overview page, click Modify BSP Settings.
13. Using the Board Support Package Settings page, you can select the OS Version and which of the Supported Libraries are to be enabled in this domain/BSP.
14. Select the lwip211 library from the list of Supported Libraries.
15. Expand the Overview tree and select lwip211. The configuration options for the lwip211 library are listed.
16. Configure the lwip211 library and click OK.

# Configuring lwIP Options

The lwIP library provides configurable parameters. There are two major categories of configurable options:

- **Xilinx adapter to lwIP options:** These control the settings used by Xilinx adapters for the ethernet cores.
- **Base lwIP options:** These options are part of lwIP library itself, and include parameters for TCP, UDP, IP, and other protocols supported by lwIP. The following sections describe the available lwIP configurable options.

# Customizing lwIP API Mode

The lwip211_v1.6 supports both raw API and socket API:

- The raw API is customized for high performance and lower memory overhead. The limitation of raw API is that it is callback-based, and consequently does not provide portability to other TCP stacks.
- The socket API provides a BSD socket-style interface and is very portable; however, this mode is not as efficient as the raw API mode in performance and memory requirements. The lwip211_v1.6 also provides the ability to set the priority on TCP/IP and other lwIP application threads.

The following table describes the lwIP library API mode options.

| Attribute | Description | Type | Default |
|---|---|---|---|
| api_mode {RAW_API \| SOCKET_API} | The lwIP library mode of operation | enum | RAW_API |
| socket_mode_thread_prio | Priority of lwIP TCP/IP thread and all lwIP application threads. This setting applies only when Xilkernel is used in priority mode. It is recommended that all threads using lwIP run at the same priority level. For GigE based Zynq-7000, Zynq UltraScale+ MPSoC, and Versal systems using FreeRTOS, appropriate priority should be set. The default priority of 1 will not give the expected behavior. For FreeRTOS (Zynq-7000, Zynq UltraScale+ MPSoC, and Versal systems), all internal lwIP tasks (except the main TCP/IP task) are created with the priority level set for this attribute. The TCP/IP task is given a higher priority than other tasks for improved performance. The typical TCP/IP task priority is 1 more than the priority set for this attribute for FreeRTOS. | integer | 1 |
| use_axieth_on_zynq | In the event that the AxiEthernet soft IP is used on a Zynq-7000 device or a Zynq UltraScale+ MPSoC. This option ensures that the GigE on the Zynq-7000 PS (EmacPs) is not enabled and the device uses the AxiEthernet soft IP for Ethernet traffic. The existing Xilinx-provided lwIP adapters are not tested for multiple MACs. Multiple Axi Ethernet's are not supported on Zynq UltraScale+ MPSoCs. | integer | 0 = Use Zynq-7000 PS-based or Zynq UltraScale+ MPSoC PS-based GigE controller 1= User AxiEthernet |

# Configuring Xilinx Adapter Options

The Xilinx adapters for EMAC/GigE cores are configurable.

### Ethernetlite Adapter Options

The following table describes the configuration parameters for the axi_ethernetlite adapter.

| Attribute | Description | Type | Default |
|---|---|---|---|
| sw_rx_fifo_size | Software Buffer Size in bytes of the receive data between EMAC and processor | integer | 8192 |
| sw_tx_fifo_size | Software Buffer Size in bytes of the transmit data between processor and EMAC | integer | 8192 |

### TEMAC Adapter Options

The following table describes the configuration parameters for the axi_ethernet and GigE adapters.

| Attribute | Type | Description |
|---|---|---|
| n_tx_descriptors | integer | Number of TX descriptors to be used. For high performance systems there might be a need to use a higher value. Default is 64. |
| n_rx_descriptors | integer | Number of RX descriptors to be used. For high performance systems there might be a need to use a higher value. Typical values are 128 and 256. Default is 64. |
| n_tx_coalesce | integer | Setting for TX interrupt coalescing. Default is 1. |
| n_rx_coalesce | integer | Setting for RX interrupt coalescing. Default is 1. |
| tcp_rx_checksum_offload | boolean | Offload TCP Receive checksum calculation (hardware support required). For GigE in Zynq devices, Zynq UltraScale+ MPSoC, and Versal ACAP, the TCP receive checksum offloading is always present, so this attribute does not apply. Default is false. |
| tcp_tx_checksum_offload | boolean | Offload TCP Transmit checksum calculation (hardware support required). For GigE cores (Zynq devices, Zynq UltraScale+ MPSoC, and Versal ACAP), the TCP transmit checksum offloading is always present, so this attribute does not apply. Default is false. |

| Attribute | Type | Description |
|---|---|---|
| tcp_ip_rx_checksum_ofload | boolean | Offload TCP and IP Receive checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq devices, Zynq UltraScale+ MPSoC, and Versal ACAP, the TCP and IP receive checksum offloading is always present, so this attribute does not apply. Default is false. |
| tcp_ip_tx_checksum_ofload | boolean | Offload TCP and IP Transmit checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq, Zynq UltraScale+ MPSoC, and Versal ACAP, the TCP and IP transmit checksum offloading is always present, so this attribute does not apply. Default is false. |
| phy_link_speed | CONFIG_LINKSPEED_ AUTODETECT | Link speed as auto-negotiated by the PHY. lwIP configures the TEMAC/GigE for this speed setting. This setting must be correct for the TEMAC/GigE to transmit or receive packets. The CONFIG_LINKSPEED_ AUTODETECT setting attempts to detect the correct linkspeed by reading the PHY registers; however, this is PHY dependent, and has been tested with the Marvell and TI PHYs present on Xilinx development boards. For other PHYs, select the correct speed. Default is enum. |
| temac_use_jumbo_ frames_experimental | boolean | Use TEMAC jumbo frames (with a size up to 9k bytes). If this option is selected, jumbo frames are allowed to be transmitted and received by the TEMAC. For GigE in Zynq devices, there is no support for jumbo frames, so this attribute does not apply. Default is FALSE. |

## Configuring Memory Options

The lwIP stack provides different kinds of memories. Similarly, when the application uses socket mode, different memory options are used. All the configurable memory options are provided as a separate category. Default values work well unless application tuning is required. The following table describes the memory parameter options.

| Attribute | Default | Type | Description |
|---|---|---|---|
| mem_size | 131072 | Integer | Total size of the heap memory available, measured in bytes. For applications which use a lot of memory from heap (using C library malloc or lwIP routine mem_malloc or pbuf_alloc with PBUF_RAM option), this number should be made higher as per the requirements. |
| memp_n_pbuf | 16 | Integer | The number of memp struct pbufs. If the application sends a lot of data out of ROM (or other static memory), this should be set high. |
| memp_n_udp_pcb | 4 | Integer | The number of UDP protocol control blocks. One per active UDP connection. |
| memp_n_tcp_pcb | 32 | Integer | The number of simultaneously active TCP connections. |
| memp_n_tcp_pcb _listen | 8 | Integer | The number of listening TC connections. |
| memp_n_tcp_seg | 256 | Integer | The number of simultaneously queued TCP segments. |
| memp_n_sys_timeout | 8 | Integer | Number of simultaneously active timeouts. |
| memp_num_netbuf | 8 | Integer | Number of allowed structure instances of type netbufs. Applicable only in socket mode. |
| memp_num_netconn | 16 | Integer | Number of allowed structure instances of type netconns. Applicable only in socket mode. |
| memp_num_api_msg | 16 | Integer | Number of allowed structure instances of type api_msg. Applicable only in socket mode. |
| memp_num_tcpip_msg | 64 | Integer | Number of TCPIP msg structures (socket mode only). |

***Note*:** Because Sockets Mode support uses Xilkernel services, the number of semaphores chosen in the Xilkernel configuration must take the value set for the memp_num_netbuf parameter into account. For FreeRTOS BSP there is no setting for the maximum number of semaphores. For FreeRTOS, you can create semaphores as long as memory is available.

# Configuring Packet Buffer (Pbuf) Memory Options

Packet buffers (Pbufs) carry packets across various layers of the TCP/IP stack. The following are the pbuf memory options provided by the lwIP stack. Default values work well unless application tuning is required. The following table describes the parameters for the Pbuf memory options.

| Attribute | Default | Type | Description |
|---|---|---|---|
| pbuf_pool_size | 256 | Integer | Number of buffers in pbuf pool. For high performance systems, you might consider increasing the pbuf pool size to a higher value, such as 512. |
| pbuf_pool_bufsize | 1700 | Integer | Size of each pbuf in pbuf pool. For systems that support jumbo frames, you might consider using a pbuf pool buffer size that is more than the maximum jumbo frame size. |
| pbuf_link_hlen | 16 | Integer | Number of bytes that should be allocated for a link level header. |

# Configuring ARP Options

The following table describes the parameters for the ARP options. Default values work well unless application tuning is required.

| Attribute | Default | Type | Description |
|---|---|---|---|
| arp_table_size | 10 | Integer | Number of active hardware address IP address pairs cached. |
| arp_queueing | 1 | Integer | If enabled outgoing packets are queued during hardware address resolution. This attribute can have two values: 0 or 1. |

# Configuring IP Options

The following table describes the IP parameter options. Default values work well unless application tuning is required.

| Attribute | Default | Type | Description |
|---|---|---|---|
| ip_forward | 0 | Integer | Set to 1 for enabling ability to forward IP packets across network interfaces. If running lwIP on a single network interface, set to 0. This attribute can have two values: 0 or 1. |
| ip_options | 0 | Integer | When set to 1, IP options are allowed (but not parsed). When set to 0, all packets with IP options are dropped. This attribute can have two values: 0 or 1. |
| ip_reassembly | 1 | Integer | Reassemble incoming fragmented IP packets. |
| ip_frag | 1 | Integer | Fragment outgoing IP packets if their size exceeds MTU. |
| ip_reass_max_pbufs | 128 | Integer | Reassembly pbuf queue length. |
| ip_frag_max_mtu | 1500 | Integer | Assumed max MTU on any interface for IP fragmented buffer. |
| ip_default_ttl | 255 | Integer | Global default TTL used by transport layers. |

## Configuring ICMP Options

The following table describes the parameter for ICMP protocol option. Default values work well unless application tuning is required.

For GigE cores (for Zynq devices and Zynq UltraScale+ MPSoCs), there is no support for ICMP in the hardware.

| Attribute | Default | Type | Description |
|---|---|---|---|
| icmp_ttl | 255 | Integer | ICMP TTL value. |

## Configuring IGMP Options

The IGMP protocol is supported by lwIP stack. When set true, the following option enables the IGMP protocol.

| Attribute | Default | Type | Description |
|---|---|---|---|
| imgp_options | FALSE | Boolean | Specify whether IGMP is required. |

# Configuring UDP Options

The following table describes UDP protocol options. Default values work well unless application tuning is required.

| Attribute | Default | Type | Description |
|---|---|---|---|
| lwip_udp | TRUE | Boolean | Specify whether UDP is required. |
| udp_ttl | 255 | Integer | UDP TTL value. |
| udp_tx_blocking | FALSE | Boolean | When enabled, the application sends UDP packet blocks till the packet is transmitted.<br>Useful for zero-copy UDP transmission use cases where the application might be using the same buffer to send information out for consecutive UDP packets. This option, when turned on, ensures that the application gets to use the TX buffer only after the buffer is used by the adapter and MAC. |

# Configuring TCP Options

The following table describes the TCP protocol options. Default values work well unless application tuning is required.

| Attribute | Default | Type | Description |
|---|---|---|---|
| lwip_tcp | TRUE | Boolean | Require TCP. |
| tcp_ttl | 255 | Integer | TCP TTL value. |
| tcp_wnd | 2048 | Integer | TCP Window size in bytes. |
| tcp_maxrtx | 12 | Integer | TCP Maximum retransmission value. |
| tcp_synmaxrtx | 4 | Integer | TCP Maximum SYN retransmission value. |
| tcp_queue_ooseq | 1 | Integer | Accept TCP queue segments out of order. Set to 0 if your device is low on memory. |
| tcp_mss | 1460 | Integer | TCP Maximum segment size. |
| tcp_snd_buf | 8192 | Integer | TCP sender buffer space in bytes. |

# Configuring DHCP Options

The DHCP protocol is supported by lwIP stack. The following table describes DHCP protocol options. Default values work well unless application tuning is required.

| Attribute | Default | Type | Description |
|---|---|---|---|
| lwip_dhcp | FALSE | Boolean | Specify whether DHCP is required. |
| dhcp_does_arp_check | FALSE | Boolean | Specify whether ARP checks on offered addresses. |

## Configuring the Stats Option

lwIP stack has been written to collect statistics, such as the number of connections used; amount of memory used; and number of semaphores used, for the application. The library provides the stats_display() API to dump out the statistics relevant to the context in which the call is used. The stats option can be turned on to enable the statistics information to be collected and displayed when the stats_display API is called from user code. Use the following option to enable collecting the stats information for the application.

| Attribute | Description | Type | Default |
|---|---|---|---|
| lwip_stats | Turn on lwIP Statistics | int | 0 |

## Configuring the Debug Option

lwIP provides debug information. The following table lists all the available options.

| Attribute | Default | Type | Description |
|---|---|---|---|
| lwip_debug | FALSE | Boolean | Turn on/off lwIP debugging. |
| ip_debug | FALSE | Boolean | Turn on/off IP layer debugging. |
| tcp_debug | FALSE | Boolean | Turn on/off TCP layer debugging. |
| udp_debug | FALSE | Boolean | Turn on/off UDP layer debugging. |
| icmp_debug | FALSE | Boolean | Turn on/off ICMP protocol debugging. |
| igmp_debug | FALSE | Boolean | Turn on/off IGMP protocol debugging. |
| netif_debug | FALSE | Boolean | Turn on/off network interface layer debugging. |
| sys_debug | FALSE | Boolean | Turn on/off sys arch layer debugging. |
| pbuf_debug | FALSE | Boolean | Turn on/off pbuf layer debugging |

# LwIP Library APIs

The lwIP library provides two different APIs: RAW API and Socket API.

## Raw API

The Raw API is callback based. Applications obtain access directly into the TCP stack and vice-versa. As a result, there is no extra socket layer, and using the Raw API provides excellent performance at the price of compatibility with other TCP stacks.

### Xilinx Adapter Requirements when using the RAW API

In addition to the lwIP RAW API, the Xilinx adapters provide the `xemacif_input` utility function for receiving packets. This function must be called at frequent intervals to move the received packets from the interrupt handlers to the lwIP stack. Depending on the type of packet received, lwIP then calls registered application callbacks. The `<Vitis_install_path>/sw/ThirdParty/sw_services/lwip211/src/lwip-2.1.1/doc/rawapi.txt` file describes the lwIP Raw API.

### LwIP Performance

The following table provides the maximum TCP throughput achievable by FPGA, CPU, EMAC, and system frequency in RAW modes. Applications requiring high performance should use the RAW API.

| FPGA | CPU | EMAC | System Frequency | Max TCP Throughput in RAW Mode (Mbps) |
|---|---|---|---|---|
| Virtex | MicroBlaze | axi-ethernet | 100 MHz | RX Side: 182 TX Side: 100 |
| Virtex | MicroBlaze | xps-ll-temac | 100 MHz | RX Side: 178 TX Side: 100 |
| Virtex | MicroBlaze | xps-ethernetlite | 100 MHz | RX Side: 50 TX Side: 38 |

### RAW API Example

Applications using the RAW API are single threaded. The following pseudo-code illustrates a typical RAW mode program structure.

```
int main()
{
        struct netif *netif, server_netif;
        ip_addr_t ipaddr, netmask, gw;


        unsigned char mac_ethernet_address[] =
                {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

        lwip_init();


        if (!xemac_add(netif, &ipaddr, &netmask,
                &gw, mac_ethernet_address,
                EMAC_BASEADDR)) {
                printf("Error adding N/W interface\n\r");
                return -1;
        }
        netif_set_default(netif);


        platform_enable_interrupts();


        netif_set_up(netif);


        start_application();


        while (1) {
                xemacif_input(netif);

                transfer_data();
        }
}
```

# Socket API

The lwIP socket API provides a BSD socket-style API to programs. This API provides an execution model that is a blocking, open-read-write-close paradigm.

### Xilinx Adapter Requirements when using the Socket API

Applications using the Socket API with Xilinx adapters need to spawn a separate thread called `xemacif_input_thread`. This thread takes care of moving received packets from the interrupt handlers to the `tcpip_thread` of the lwIP. Application threads that use lwIP must be created using the lwIP `sys_thread_new` API. Internally, this function makes use of the appropriate thread or task creation routines provided by XilKernel or FreeRTOS.

### Xilkernel/FreeRTOS scheduling policy when using the Socket API

lwIP in socket mode requires the use of the Xilkernel or FreeRTOS, which provides two policies for thread scheduling: round-robin and priority based. There are no special requirements when round-robin scheduling policy is used because all threads or tasks with same priority receive the same time quanta. This quanta is fixed by the RTOS (Xilkernel or FreeRTOS) being used. With priority scheduling, care must be taken to ensure that lwIP threads or tasks are not starved. For Xilkernel, lwIP internally launches all threads at the priority level specified in `socket_mode_thread_prio`. For FreeRTOS, lwIP internally launches all tasks except the main TCP/IP task at the priority specified in `socket_mode_thread_prio`. The TCP/IP task in FreeRTOS is launched with a higher priority (one more than priority set in `socket_mode_thread_prio`). In addition, application threads must launch `xemacif_input_thread`. The priorities of both `xemacif_input_thread`, and the lwIP internal threads (`socket_mode_thread_prio`) must be high enough in relation to the other application threads so that they are not starved.

### Socket API Example

XilKernel-based applications in socket mode can specify a static list of threads that Xilkernel spawns on startup in the Xilkernel Software Platform Settings dialog box. Assuming that `main_thread()` is a thread specified to be launched by XIlkernel, control reaches this first thread from application `main` after the Xilkernel schedule is started. In `main_thread`, one more thread (network_thread) is created to initialize the MAC layer. For FreeRTOS (Zynq, Zynq Ultrascale+, and Versal processor systems) based applications, once the control reaches application `main` routine, a task (can be termed as main_thread) with an entry point function as main_thread() is created before starting the scheduler. After the FreeRTOS scheduler starts, the control reaches `main_thread()`, where the lwIP internal initialization happens. The application then creates one more thread (network_thread) to initialize the MAC layer. The following pseudo-code illustrates a typical socket mode program structure.

```
void network_thread(void *p)
{
        struct netif *netif;
        ip_addr_t ipaddr, netmask, gw;


        unsigned char mac_ethernet_address[] =
                {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

        netif = &server_netif;
```

```
        IP4_ADDR(&ipaddr,192,168,1,10);
        IP4_ADDR(&netmask,255,255,255,0);
        IP4_ADDR(&gw,192,168,1,1);


        if (!xemac_add(netif, &ipaddr, &netmask,
                        &gw, mac_ethernet_address,
                        EMAC_BASEADDR)) {
                printf("Error adding N/W interface\n\r");
                return;
        }
        netif_set_default(netif);


        netif_set_up(netif);


        sys_thread_new("xemacif_input_thread", xemacif_input_thread,
                netif,
                THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);


        sys_thread_new("httpd" web_application_thread, 0,
                        THREAD_STACKSIZE DEFAULT_THREAD_PRIO);
}

int main_thread()
{
        lwip_init();


        sys_thread_new("network_thread" network_thread, NULL,
                        THREAD_STACKSIZE DEFAULT_THREAD_PRIO);

        return 0;
}
```

# Using the Xilinx Adapter Helper Functions

The Xilinx adapters provide the following helper functions to simplify the use of the lwIP APIs.

*Table 1:* **Quick Function Reference**

| Type | Name | Arguments |
|---|---|---|
| void | xemacif_input_thread | void |
| struct netif * | xemac_add | void |
| void | lwip_init | void |

*Table 1:* **Quick Function Reference** *(cont'd)*

| Type | Name | Arguments |
|---|---|---|
| int | xemacif_input | void |
| void | xemacpsif_resetrx_on_no_rxdata | void |

# Functions

## *xemacif_input_thread*

In the socket mode, the application thread must launch a separate thread to receive the input packets. This performs the same work as the RAW mode function, `xemacif_input()` , except that it resides in its own separate thread; consequently, any lwIP socket mode application is required to have code similar to the following in its main thread:

***Note*:** For Socket mode only.

```
sys_thread_new("xemacif_input_thread",

               xemacif_input_thread
            , netif, THREAD_STACK_SIZE, DEFAULT_THREAD_PRIO);
```

The application can then continue launching separate threads for doing application specific tasks. The `xemacif_input_thread()` receives data processed by the interrupt handlers, and passes them to the lwIP tcpip_thread.

### Prototype

```
void xemacif_input_thread(struct netif *netif);
```

### Returns

## *xemac_add*

The `xemac_add()` function provides a unified interface to add any Xilinx EMAC IP as well as GigE core. This function is a wrapper around the lwIP netif_add function that initializes the network interface 'netif' given its IP address ipaddr, netmask, the IP address of the gateway, gw, the 6 byte ethernet address mac_ethernet_address, and the base address, mac_baseaddr, of the axi_ethernetlite or axi_ethernet MAC core.

### Prototype

```
struct netif * xemac_add(struct netif *netif, ip_addr_t *ipaddr, ip_addr_t
*netmask, ip_addr_t *gw, unsigned char *mac_ethernet_address, unsigned
mac_baseaddr);
```

## *lwip_init*

Initialize all modules. Use this in NO_SYS mode. Use tcpip_init() otherwise.

This function provides a single initialization function for the lwIP data structures. This replaces specific calls to initialize stats, system, memory, pbufs, ARP, IP, UDP, and TCP layers.

### Prototype

```
void lwip_init(void);
```

## *xemacif_input*

The Xilinx lwIP adapters work in interrupt mode. The receive interrupt handlers move the packet data from the EMAC/GigE and store them in a queue. The `xemacif_input()` function takes those packets from the queue, and passes them to lwIP; consequently, this function is required for lwIP operation in RAW mode. The following is a sample lwIP application in RAW mode.

***Note*:** For RAW mode only.

```
            while (1) {


               xemacif_input
            (netif);


}
```

***Note*:** The program is notified of the received data through callbacks.

### Prototype

```
int xemacif_input(struct netif *netif);
```

### Returns

## *xemacpsif_resetrx_on_no_rxdata*

There is an errata on the GigE controller that is related to the Rx path. The errata describes conditions whereby the Rx path of GigE becomes completely unresponsive with heavy Rx traffic of small sized packets. The condition occurrence is rare; however a software reset of the Rx logic in the controller is required when such a condition occurs. This API must be called periodically (approximately every 100 milliseconds using a timer or thread) from user applications to ensure that the Rx path never becomes unresponsive for more than 100 milliseconds.

***Note*:** Used in both Raw and Socket mode and applicable only for the Zynq-7000 devices, Zynq UltraScale+ MPSoC, and Versal processors and the GigE controller

### Prototype

```
void xemacpsif_resetrx_on_no_rxdata(struct netif *netif);
```

### Returns

XILINX

*Appendix A*

# Additional Resources and Legal Notices

## Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

## Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note:*** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

# Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

### AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

### Copyright