lwIP 1.4.0 Library (v2.1)

UG650 June 4, 2014

Overview

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 140_v2_00_a is an SDK library that is built on the open source lwIP library version 1.4.0. The lwip140_v2_00_a 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™ and ARM Cortex-A9 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).

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)

Additional Resources

- lwIP wiki: http://lwip.scribblewiki.com
- Xilinx® lwIP designs and application examples: http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf
- lwIP examples using RAW and Socket APIs: http://savannah.nongnu.org/projects/lwip/
- FreeRTOS Port for Zynq is available for download from the FreeRTOS website: http://www.freertos.org/Interactive_Frames/Open_Frames.html?http://interactive.freertos.org/forums

Using lwIP

The following sections detail the hardware and software steps for using lwIP for networking. The key steps are:

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 140_v2_00_a 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), there is no support for Xilkernel. Instead, use FreeRTOS. A FreeRTOS BSP is available for Zynq systems and must be included for using lwIP socket API. The FreeRTOS BSP for Zynq is available for download from: http://www.freertos.org/Interactive_Frames/Open_Frames.html?http://interactive.freertos.org/forums

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 processor. The Cortex-A9 processor applies to Zynq 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 or a fifo interface. For GigE-based Zynq 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.

Figure 1 shows a sample system architecture with a Kintex®-6 device utilizing the axi_ethernet core with DMA.

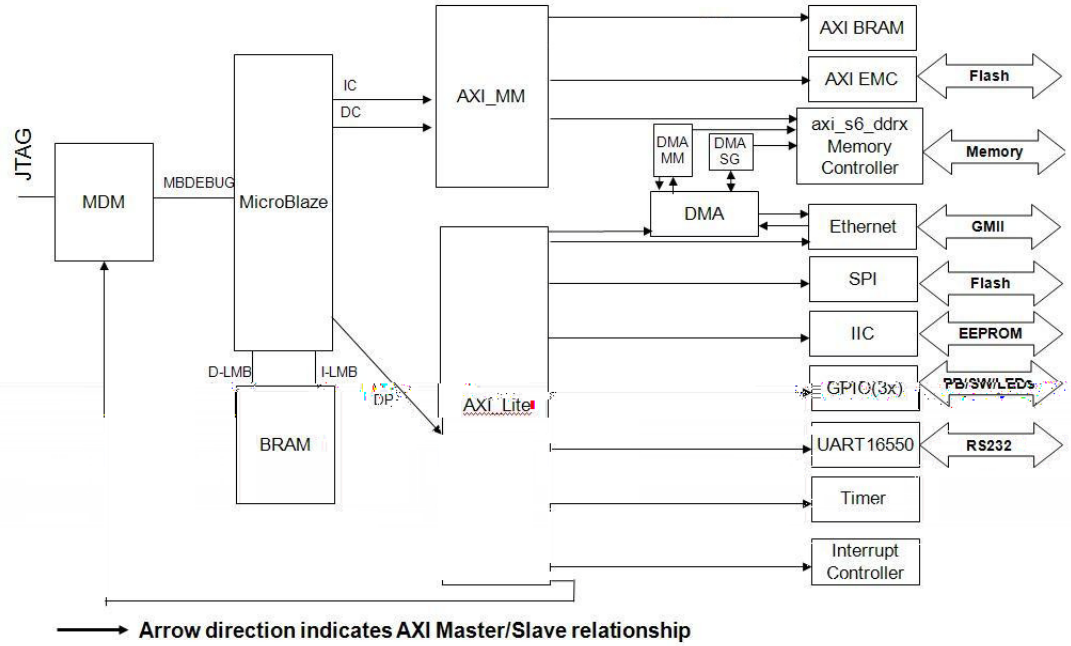

Figure 1: **System Architecture using axi_ethernet core with DMA**

Setting up the Software System

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

To move the hardware design to SDK, you must first export it from the Hardware Tools.

1. Select **Project > Export Hardware Design to SDK**.
2. On the Export to SDK dialog box that opens, click **Export & Launch SDK**.

   Vivado exports the design to SDK. SDK opens and prompts you to create a workspace.

After SDK opens with hw_platform already present in the Project Explorer, compile the lwIP library:

1. Select **File > New > Xilinx Board Support Package**.

   The New Board Support Package window opens.
2. Give the project a name and select a location for it. Select XilKernel, Standalone, or FreeRTOS, and click **Finish**.

   Note: For Zynq there is no option for XilKernel. FreeRTOS must be used for Zynq. The FreeRTOS BSP for Zynq is available for download from: http://www.freertos.org/Interactive_Frames/Open_Frames.html?http://interactive.freertos.org/forums

   Follow the steps provided in the pdf document provided with the port to use the FreeRTOS BSP.

   The Board Support Package Settings window opens.
3. Select the lwip140 library with version 2.1.

   On the left side of the SDK window, lwip140 appears in the list of libraries to be compiled.
4. Select lwip140 in the Project Explorer tab. The configuration options for lwIP are listed. Configure the lwIP and click **OK**.

   The board support package automatically builds with lwIP included in it.

Configuring lwIP Options

The lwIP provides configurable parameters. The values for these parameters can be changed in SDK. 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 140_v2_00_a 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 raw API mode in performance and memory requirements.

The 140_v2_00_aalso provides the ability to set the priority on TCP/IP and other lwIP application threads. Table 1 provides lwIP library API modes.

Table 1: **API Mode Options and Descriptions**

Attribute/Options	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. ***Note:*** For GigE based Zynq-7000 systems using FreeRTOS, appropriate priority should be set. The default priority of 1 will not give the expected behavior. For FreeRTOS (Zynq-700 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. 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. ***Note:*** The existing Xilinx-provided lwIP adapters are not tested for multiple MACs.	integer	0 = Use Zynq-7000 PS-based GigE controller 1= User AxiEthernet.

Configuring Xilinx Adapter Options

The Xilinx adapters for EMAC/GigE cores are configurable.

Ethernetlite Adapter Options

Table 2 provides the configuration parameters for the `axi_ethernetlite` adapter.

Table 2: **xps_ethernetlite Adapter Options**

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

Table 3 provides the configuration parameters for the axi_ethernet and GigE adapters.

Table 3: **axi_Ethernet/GigE Adapter**

Attribute	Default	Type	Description
`n_tx_descriptors`	64	integer	Number of Tx descriptors to be used. For high performance systems there might be a need to use a higher value for this.
`n_rx_descriptors`	64	integer	Number of Rx descriptors to be used. For high performance systems there might be a need to use a higher value for this. Typical values are 128 and 256.
`n_tx_coalesce`	1	integer	Setting for Tx interrupt coalescing[1]
`n_rx_coalesce`	1	integer	Setting for Rx interrupt coalescing[1]
`tcp_rx_checksum_offload`	false	boolean	Offload TCP Receive checksum calculation (hardware support required). For GigE in Zynq, the TCP receive checksum offloading is always present, so this attribute does not apply.
`tcp_tx_checksum_offload`	false	boolean	Offload TCP Transmit checksum calculation (hardware support required). For GigE cores (for Zynq) the TCP transmit checksum offloading is always present, so this attribute does not apply.
`tcp_ip_rx_checksum_ofload`	false	boolean	Offload TCP and IP Receive checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq the TCP and IP receive checksum offloading is always present, so this attribute does not apply.
`tcp_ip_tx_checksum_ofload`	false	boolean	Offload TCP and IP Transmit checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq the TCP and IP transmit checksum offloading is always present, so this attribute does not apply.

Table 3: **axi_Ethernet/GigE Adapter *(Cont'd)***

phy_link_speed	enum	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. ***Note:*** 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 PHYs present on Xilinx development boards. For other PHYs, select the correct speed.
temac_use_jumbo_frames_experimental	false	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 there is no support for jumbo frames, so this attribute does not apply.

1. This setting is not applicable for GigE in Zynq.

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 memory parameter options are provided in Table 4:

Table 4: **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.

Table 5 provides the parameters for the Pbuf memory options:

Table 5: **Pbuf Memory Options Configuration Parameters**

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

Table 6 provides the parameters for the ARP options. Default values work well unless application tuning is required.

Table 6: **ARP Options Configuration Parameters**

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

Table 7 provides the IP parameter options. Default values work well unless application tuning is required.

Table 7: **IP Configuration Parameter Options**

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.

Table 7: **IP Configuration Parameter Options *(Cont'd)***

Attribute	Default	Type	Description
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

Table 8 provides the parameter for ICMP protocol option. Default values work well unless application tuning is required.

Table 8: **ICMP Configuration Parameter Option**

Attribute	Default	Type	Description
icmp_ttl	255	Integer	ICMP TTL value. For GigE cores (for Zynq) there is no support for ICMP in the hardware.

Configuring IGMP Options

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

Table 9: **IGMP Configuration Parameter Option**

Attribute	Default	Type	Description
imgp_options	false	Boolean	Specify whether IGMP is required.

Configuring UDP Options

Table 10 provides UDP protocol options. Default values work well unless application tuning is required.

Table 10: **UDP Configuration Parameter Options**

Attribute	Default	Type	Description
lwip_udp	true	Boolean	Specify whether UDP is required.
udp_ttl	255	Integer	UDP TTL value.

Configuring TCP Options

Table 11 provides the TCP protocol options. Default values work well unless application tuning is required.

Table 11: **TCP Options Configuration Parameters**

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.

Table 11: **TCP Options Configuration Parameters *(Cont'd)***

Attribute	Default	Type	Description
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. Table 12 provides DHCP protocol options. Default values work well unless application tuning is required.

Table 12: **DHCP Options Configuration Parameters**

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.

Table 13: **Statistics Option Configuration Parameters**

Attribute	Description	Type	Default
lwip_stats	Turn on lwIP Statistics	int	0

Configuring the Debug Option

lwIP provides debug information. Table 14 lists all available options.

Table 14: **Debug Option Configuration Parameters**

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

Software APIs

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

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 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.

Raw API File

The `$XILINX_SDK/sw/ThirdParty/sw_services/140_v2_00_a/src/lwip-1.4.0/doc/rawapi.txt` file describes the lwIP Raw API.

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 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 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.

Using Xilinx Adapter Helper Functions

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

`void` **`lwip_init`**`()`

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.

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

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.

`void` **`xemacif_input`**`(struct netif *netif)`

(***RAW mode only***)

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.

```
while (1) {
            /* receive packets */
            xemacif_input(netif);

            /* do application specific processing */
      }
```

The program is notified of the received data through callbacks.

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

(***Socket mode only***)

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:

```
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`.

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

(Used in both Raw and Socket mode and applicable only for the Zynq-7000 processor and GigE controller)

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.

lwIP Performance

Table 15 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.

Table 15: **Library Performance**

FPGA	CPU	EMAC	System Frequency	Max TCP Throughput in RAW Mode	
				Rx Side	Tx Side
Virtex®	MicroBlaze	`axi-ethernet`	100 MHz	182 Mbps	100 Mbps
Virtex	MicroBlaze	`xps-ll-temac`	100 MHz	178 Mbps	100 Mbps
Virtex	MicroBlaze	`xps-ethernetlite`	100 MHz	50 Mbps	38 Mbps

API Examples

Sample applications using the RAW API and Socket API are available on the Xilinx website. This section provides pseudo code that illustrates the typical code structure.

RAW API

Applications using the RAW API are single threaded, and have the following broad structure:

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

        /* the MAC address of the board.
        * This should be unique per board/PHY */
        unsigned char mac_ethernet_address[] =
            {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

        lwip_init();

        /* Add network interface to the netif_list,
         * and set it as default */
        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);

        /* now enable interrupts */
```

```
        platform_enable_interrupts();

        /* specify that the network if is up */
        netif_set_up(netif);

        /* start the application, setup callbacks */
        start_application();

        /* receive and process packets */
        while (1) {
            xemacif_input(netif);
            /* application specific functionality */
            transfer_data();
        }
}
```

RAW API works primarily using asynchronously called Send and Receive callbacks.

Socket API

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-7000 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;
        struct ip_addr ipaddr, netmask, gw;

        /* the MAC address of the board.
         * This should be unique per board/PHY */
        unsigned char mac_ethernet_address[] =
            {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

        netif = &server_netif;

        /* initialize IP addresses to be used */
        IP4_ADDR(&ipaddr,192,168,1,10);
        IP4_ADDR(&netmask,255,255,255,0);
        IP4_ADDR(&gw,192,168,1,1);

        /* Add network interface to the netif_list,
         * and set it as default */
        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);

        /* specify that the network if is up */
```

```
        netif_set_up(netif);

        /* start packet receive thread
         - required for lwIP operation */
        sys_thread_new("xemacif_input_thread", xemacif_input_thread,
            netif,
            THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);

        /* now we can start application threads */
        /* start webserver thread (e.g.) */
        sys_thread_new("httpd" web_application_thread, 0,
              THREAD_STACKSIZE DEFAULT_THREAD_PRIO);
}

int main_thread()
{
        /* initialize lwIP before calling sys_thread_new */
        lwip_init();

        /* any thread using lwIP should be created using
         * sys_thread_new() */
        sys_thread_new("network_thread" network_thread, NULL,
              THREAD_STACKSIZE DEFAULT_THREAD_PRIO);

       return 0;
}
```