UG650 October 5, 2010

lwIP 1.3.0 Library (v3.00.a)

Summary

This document describes the Embedded Development Kit (EDK) port of the open source lightweight IP (lwIP) TCP/IP stack. The lwIP provides an easy way to add TCP/IP-based networking capability to an embedded system.

The `lwip130_v3_00_a` provides adapters for the xps_ethernetlite, xps_ll_temac, axi_ethernetlite, and axi_ethernet Xilinx® Ethernet MAC cores, and is based on the lwIP stack version 1.3.0. This document describes how to use `lwip130_v3_00_a` to add networking capability to embedded software. It contains the following sections:

- "Overview"
- "Features"
- "Additional Resources"
- "Using lwIP"
- "Setting up the Hardware System"
- "Setting up the Software System"
- "lwIP Performance"
- "Known Issues and Restrictions"
- "Migrating from lwip_v3_00_a to lwip130_v3_00_a"
- "API Examples"

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 APIs 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 `lwip130_v3_00_a` is an EDK library that is built on the open source lwIP library version 1.3.0. The `lwip130_v3_00_a` library provides adapters for the Ethernetlite (xps_ethernetlite, axi_ethernetlite) and the TEMAC (xps_ll_temac, axi_ethernet) Xilinx EMAC cores. The library can run on MicroBlaze™, PowerPC® 405, or PowerPC 440 processors.

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)

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/
- *Multi-Port Memory Controller (MPMC) Data Sheet:* available in the following directory of your software installation: `EDK\hw\XilinxProcessorIPLib\pcores\mpmc_v*_00_a`

Using lwIP

The following sections detail the hardware and software steps for using lwIP for networking in an EDK system. 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 `lwip130_v3_00_a` to be a part of the software platform. For lwIP socket API, the Xilkernel library is a pre-requisite.

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 PowerPC (405 or 440 processor) or a MicroBlaze processor.
- EMAC: lwIP supports xps_ethernetlite, axi_ethernetlite, xps_ll_temac, and axi_ethernet EMAC cores
- Timer: to maintain TCP timers, lwIP requires 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: The xps_ll_temac and the axi_ethernet cores can be configured with an optional soft Direct Memory Access (DMA) engine.

The following figure shows a system architecture in which the system is using an `xps_ethernetlite` core.

The system has a processor connected to a Multi-Port Memory Controller (MPMC) with the other required peripherals (timer and ethernetlite) on the PLB v4.6 bus. Interrupts from both the timer and the ethernetlite are required, so interrupts are connected to the interrupt controller.

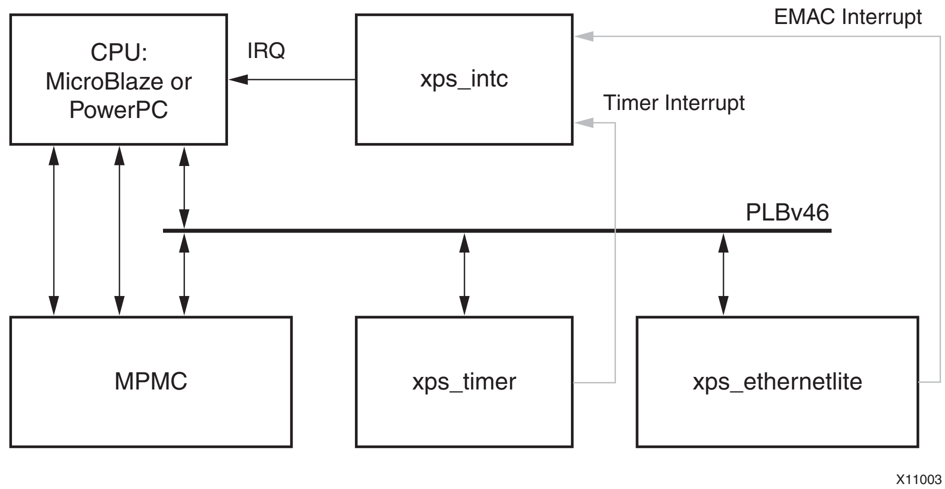

Figure 1: **System Architecture using xps_ethernetlite Core**

When using TEMAC, the system architecture changes depending on whether DMA is required. If DMA is required, a fourth port (of type SDMA), which provides direct connection between the TEMAC (`xps_ll_temac`) and the memory controller (MPMC), is added to the memory controller. The following figure shows this system architecture

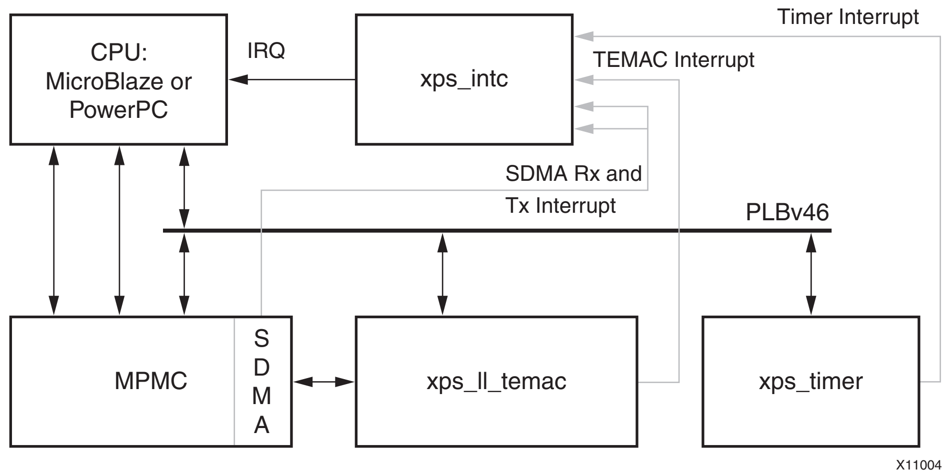

Figure 2: **System Architecture using xps_ll_temac Core (with DMA)**

Note: There are four interrupts that are necessary in this case: a timer interrupt, a TEMAC interrupt, and the SDMA RX and TX interrupts. The SDMA interrupts are from the Multi-Port Memory Controller (MPMC) SDMA Personality Interface Module (PIM). Refer to the *Multi-Port Memory Controller (MPMC) Data Sheet* for more information.

If the TEMAC is used without DMA, a FIFO (`xps_ll_fifo`) is used to interface to the TEMAC. The system architecture in this case is shown in the following figure.

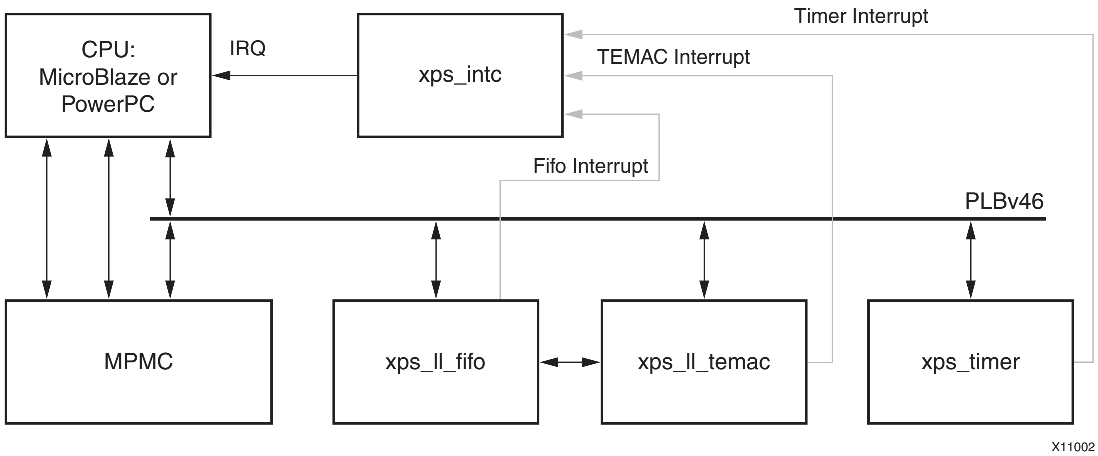

Figure 3: **System Architecture using TEMAC with xps_ll_fifo (without DMA)**

The following figure shows a sample system architecture with Spartan 6 utilizing the axi_ethernet core with DMA.

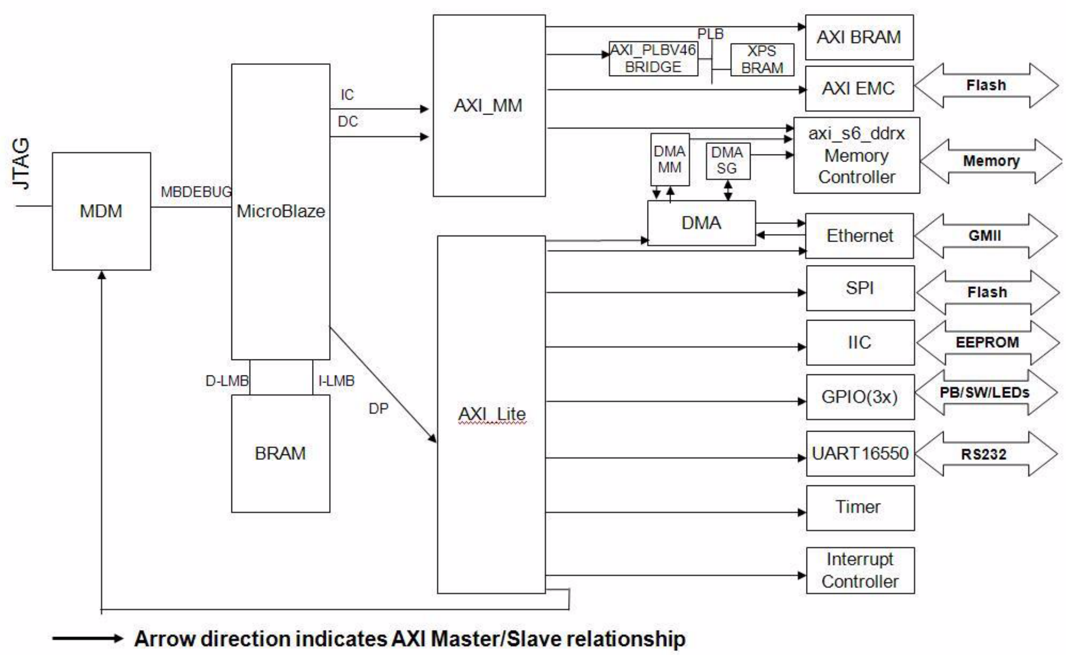

Figure 4: **System Architecture using axi_ethernet core with DMA**

Setting up the Software System

To use lwIP in a software application, you must first compile the library as part of your application software platform. To set up the lwIP library in XPS:

1. Open the **Software Platform Settings** dialog box.
2. Enable lwIP in the **Library/OS Settings** tab. (For Socket API, Xilkernel must be the OS, configured with semaphores, mutexes, and yield functionality available).
3. Select **Generate Libraries and BSPs** to regenerate the library.
4. Link the application with the **`-l lwip4`** linker flag. (For socket API, add **`-l xilkernel`**.)

Configuring lwIP Options

The lwIP provides configurable parameters. The values for these parameters can be changed using the **Software Platform Settings** dialog box. 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 `lwip130_v3_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 inefficient both in performance and memory requirements.

The `lwip130_v3_00_a` also provides the ability to set the priority on TCP/IP and other lwIP application threads. The following table 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 *`integer`*	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.	integer	1

Configuring Xilinx Adapter Options

The Xilinx adapters for EMAC cores are configurable.

Ethernetlite Adapter Options

The following table provides the configuration parameters for the `xps_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

The following table provides the configuration parameters for the xps_ll_temac and axi_ethernet adapters.

Table 3: **xps_ll_temac Adapter**

Attribute	Description	Type	Default
`phy_link_speed {CONFIG_LINKSPEED10 | CONFIG_LINKSPEED100 | CONFIG_LINKSPEED1000 | CONFIG_LINKSPEED_AUTODETECT}`	Link speed as auto-negotiated by the PHY. lwIP configures the TEMAC for this speed setting. This setting must be correct for the TEMAC to transmit or receive packets. ***Note:*** The setting, `CONFIG_LINKSPEED_AUTODETECT`, 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, the correct speed should be chosen.	int	`CONFIG_LINKSPEED_AUTODETECT`
`n_tx_descriptors`	Number of TX buffer descriptors used in SDMA mode	int	32
`n_rx_descriptors`	Number of RX buffer descriptors used in SDMA mode	int	32
`n_tx_coalesce`	TX interrupt coalescing setting for the TEMAC	int	1
`n_rx_coalesce`	RX interrupt coalescing setting for the TEMAC	int	1
`tcp_tx_csum_offload`	TX enable checksum offload	int	1
`tcp_rx_csum_offload`	RX enable checksum offload	int	1

Configuring Memory Options

lwIP stack provides different kinds of memories. The configurable memory options are provided as a separate category. Default values work well unless application tuning is required. The various memory parameter options are provided in the following table:

Table 4: **Memory Configuration Parameter Options**

Attribute	Description	Type	Default
`mem_size`	Size of the heap memory in bytes. Set this value high if application sends out large data.	int	8192
`mem_num_pbuf`	Number of memp struct pbufs. Set this value high if application sends lot of data out of ROM or static memory.	int	16
`mem_num_udp_pcb`	Number of active UDP protocol control blocks. One per active UDP connection.	int	5
`mem_num_tcp_pcb`	Number of active TCP protocol control blocks. One per active TCP connections.	int	5
`mem_num_tcp_pcb_listen`	Number of listening TCP connections.	int	5
`mem_num_tcp_seg`	Number of simultaneously queued TCP segments.	int	255
`mem_num_sys_timeout`	Number of simultaneously active time-outs.	int	3

Configuring Socket Memory Options

Sockets API mode has memory options. The configurable socket memory options are provided as a separate category. Default values work well unless application tuning is required. The following table provides the parameters for the socket memory options.

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

Attribute	Description	Type	Default
`memp_num_netbuf`	Number of struct `netbufs`. This translates to one per socket.	int	5
`memp_num_netconn`	Number of struct `netconns`. This translates to one per socket.	int	5
`memp_num_api_msg`	Number of struct `api_msg`. Used for communication between TCP/IP stack and application.	int	8
`memp_num_tcpip_msg`	Number of struct `tcpip_msg`. Used for sequential API communication and incoming packets.	int	8

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.

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 provides the parameters for the Pbuf memory option:

Table 6: **Pbuf Memory Options Configuration Parameters**

Attribute	Description	Type	Defaults
`pbuf_pool_size`	Number of buffers in `pbuf` pool.	int	512
`pbuf_pool_bufsize`	Size in bytes of each pbuf in `pbuf` pool.	int	1536

Configuring ARP Options

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

Table 7: **ARP Options Configuration Parameters**

Attribute	Description	Type	Default
`arp_table_size`	Number of active hardware addresses, IP address pairs cached.	int	10
`arp_queueing`	When enabled, (default (1)), outgoing packets are queued during hardware address resolution.	int	1
`arp_queue_first`	When enabled, first packet queued is not overwritten by later packets. The default (0), disabled, is recommended.	int	0
`etharp_always_insert`	When set to 1, cache entries are updated or added for every ARP traffic. This option is recommended for routers. When set to 0, only existing cache entries are updated. Entries are added when lwIP is sending to them. Recommended for embedded devices.	int	0

Configuring IP Options

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

Table 8: **IP Configuration Parameter Options**

Attribute	Description	Type	Default
`ip_forward`	Set to 1 for enabling ability to forward IP packets across network interfaces. If running lwIP on a single network interface, set o 0.	int	0
`ip_reassembly`	Reassemble incoming fragmented IP packets.	int	1
`ip_frag`	Fragment outgoing IP packets if their size exceeds MTU.	int	1
`ip_options`	When set to 1, IP options are allowed (but not parsed). When set to 0, all packets with IP options are dropped.	int	0

Configuring ICMP Options

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

Table 9: **ICMP Configuration Parameter Option**

Attribute	Description	Type	Default
icmp_ttl	ICMP TTL value.	int	255

Configuring UDP Options

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

Table 10: **UDP Configuration Parameter Options**

Attribute	Description	Type	Defaults
lwip_udp	Specify if UDP is required.	bool	true
udp_ttl	UDP TTL value.	int	255

Configuring TCP Options

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

Table 11: **TCP Options Configuration Parameters**

Attribute	Description	Type	Defaults
lwip_tcp	Require TCP.	bool	true
tcp_ttl	TCP TTL value.	int	255
tcp_wnd	TCP Window size in bytes.	int	16384
tcp_maxrtx	TCP Maximum retransmission value.	int	12
tcp_synmaxrtx	TCP Maximum SYN retransmission value.	int	4
tcp_queue_ooseq	Accept TCP queue segments out of order. Set to 0 if your device is low on memory.	int	1
tcp_mss	TCP Maximum segment size.	int	1476
tcp_snd_buf	TCP sender buffer space in bytes.	int	32768

Configuring Debug Options

lwIP stack has debug information. The debug mode can be turned on to dump the debug messages onto STDOUT. The following option, when set to true, prints the debug messages.

Table 12: **Debug Options Configuration Parameters**

Attribute	Description	Type	Default
lwip_debug	Turn on lwIP Debug	bool	false

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 Options Configuration Parameters**

Attribute	Description	Type	Default
`lwip_stats`	Turn on lwIP Statistics	int	0

Software APIs

lwIP 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 unnecessary copying of data, 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_EDK/sw/ThirdParty/sw_services/lwip130_v3_00_a/src/lwip-1.3.0/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 `pthread_create()` function in Xilkernel to create a new thread. It also initializes specific per-thread timeout structures necessary for lwIP operation.

Xilkernel scheduling policy when using Socket API

lwIP in socket mode requires the use of the Xilkernel, 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 receive the same time quanta.

With priority scheduling, care must be taken to ensure that lwIP threads are not starved. lwIP internally launches all threads at the priority level specified 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. 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 `xps_ethernetlite` or `xps_ll_temac` 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 and store them in a queue. The `xemacif_input` function takes those received packets from the queue, and passes them to lwIP; consequently, this function is required for lwIP operation in RAW mode. An lwIP application in RAW mode should have a structure like:

```
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`.

lwIP Performance

This section provides a brief overview of the expected performance when using lwIP with Xilinx Ethernet MACs.

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

Table 14: **Library Performance**

FPGA	CPU	EMAC	System Frequency	Max TCP Throughput	
				RAW Mode	Socket Mode
Virtex®	PowerPC 405	`xps_ll_temac`	100 MHz	140 Mbps	40 Mbps
Virtex	MicroBlaze	`xps_ll_temac`	125 MHz	100 Mpbs	30 Mbps
Spartan®	MicroBlaze	`xps_ll_temac`	66 MHz	35 Mpbs	10 Mbps
Spartan	MicroBlaze	`xps_ethernetlite`	66 MHz	15 Mbps	7 Mbps

Known Issues and Restrictions

The lwip130_v3_00_a does not support more than one TEMAC within a single xps_ll_temac instance. For example, lwip130_v3_00_a does not support the TEMAC enabled by setting `C_TEMAC1_ENABLED` = 1 in `xps_ll_temac`.

Migrating from lwip_v3_00_a to lwip130_v3_00_a

You must make the following changes to applications written to work with lwip_v3_00_a in order for them to work with lwip130_v3_00_a:

- The API for function `sys_thread_new` has changed from lwIP 1.2.0 to lwIP 1.3.0. Use the new API as follows:

  ```
  sys_thread_t sys_thread_new(char *name, void (*thread)(void *arg), void
  *arg, int stacksize, int prio);
  ```

- Configure Xilkernel to include yield functionality.
- UDP RAW mode callback functions receive a pointer to the IP address of the sender as one of the parameters. Do not pass this parameter back to any other UDP function as an argument. Instead, make a copy and pass a pointer to the copy.

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

In socket mode, applications specify a static list of threads that Xilkernel spawns on startup in the Xilkernel software platform settings. Assuming that `main_thread()` is a thread specified to be launched by Xilkernel, then 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;
}
```