



XAPP1026 (v1.0) April 11, 2008

## LightWeight IP (lwIP) Application Examples

Author: Siva Velusamy

### Summary

Lightweight IP (lwIP) is an open source TCP/IP networking stack for embedded systems. Xilinx Embedded Development Kit (EDK) provides lwIP software customized to run on Xilinx Embedded systems containing either a PowerPC® or a MicroBlaze™ processor. This application note describes how to utilize the lwIP library to add networking capability to an embedded system. In particular, lwIP is utilized to develop the following applications: echo server, web server, and a TFTP server.

### Included Systems

Included with this application note are reference systems for the Xilinx ML505, ML403 and Spartan®-3AN FPGA Starter Kit boards:

[www.xilinx.com/support/documentation/application\\_notes/xapp1026.zip](http://www.xilinx.com/support/documentation/application_notes/xapp1026.zip)

### Hardware and Software Requirements

The hardware and software requirements are:

- One of Xilinx [ML505](#), [ML403](#) or [Spartan-3AN Starter](#) Development Board
- Xilinx Platform USB Cable or Parallel IV Cable
- RS232 Cable
- A crossover ethernet cable connecting the board to a Windows or Linux host.
- Serial Communications Utility Program, such as HyperTerminal or Teraterm
- Xilinx Platform Studio 10.1i
- ISE® 10.1 design tools

### Introduction

[lwIP](#) is an open source networking stack designed for embedded systems. It is provided under a BSD style license. The objective of this application note is to describe how to use lwIP shipped along with the Xilinx EDK to add networking capability to an embedded system. In particular, this application note describes the:

- EDK hardware system requirements for running lwIP
- Software applications (echo server, web server and TFTP server) utilizing lwIP.

### Reference System Specifics

The reference design for this application note is structured in the following way. The zip file contains two folders, “raw” and “socket”, both of which have the structure as shown in Table 1.

**Table 1: Structure of raw and socket Folders in the Reference Design**

Folder	Contents
ml505	EDK design for ML505 board: contains MicroBlaze, xps_ll_temac and SDMA.
ml403	EDK design for ML403 board: contains PPC405, xps_ll_temac and xps_ll_fifo.
s3an	EDK design for Spartan-3AN Starter board: contains MicroBlaze and xps_ethernetlite.

The raw folder contains applications that use the lwIP raw API. The socket folder contains software that uses the API of the lwIP socket. Within each design, there is a software folder (apps/src) containing the source code for all the applications. The same source code is used in all three designs.

## Hardware Systems

Three reference hardware systems are present in the reference design. These include designs for the ML505, ML403 and Spartan-3AN starter kit boards. This section describes the hardware designs.

### Minimal Hardware Requirements for lwIP

To run lwIP, the following hardware components are required:

- **A processor:** In a Xilinx EDK based embedded system, this can be a PowerPC 405 processor, or a MicroBlaze soft-processor, depending on the FPGA.
- **An Ethernet MAC:** The Ethernet MAC IP is required to send and receive packets. EDK provides two MAC IPs:
  - ◆ **xps\_ethernetlite:** a small (low area), simple Ethernet MAC that is suitable for applications that require low performance network connectivity
  - ◆ **xps\_ll\_temac:** a high performance ethernet MAC that is suitable for applications that require high bandwidth network connectivity. It can utilize the embedded Hard Trimode Ethernet MACs present in some FPGA families.
- **Interrupt controller:** The Xilinx lwIP adapters work in interrupt mode only, meaning that packet reception or transmission is notified to software via interrupts by the ethernet MAC. An interrupt controller is required to connect multiple sources of interrupt to the processor.
- **Programmable Timer:** lwIP requires a periodic interrupt to update TCP timers. This is typically implemented by programming a timer to generate interrupts at a constant rate. PowerPC 405 contains an internal timer which can be used for this purpose. For MicroBlaze systems however, an external timer is required. EDK provides the xps\_timer IP core that can be used for this purpose.

### Optional Hardware Components

The web server software application demonstrates how to control or monitor the status of an embedded system by interfacing to LEDs or DIP switches. These peripherals are controlled by GPIO (General Purpose I/O) cores. Applications requiring such functionality must add XPS GPIO IP cores to interface to the peripherals.

### Hardware System Block Diagrams

The block diagrams of the three hardware designs present in the reference design are shown below.

#### *ML505 Hardware System*

The ML505 hardware design shown in [Figure 1](#) is comprised of a MicroBlaze soft processor, a Multi-Ported Memory Controller (MPMC) connected to DDR2 memory, and the high performance XPS LL TEMAC core. A timer is present as a source of periodic interrupts. The interrupt controller connects the DMA receive and transmit interrupts, EMAC error interrupt and the timer interrupt to the processor. XPS GPIO cores are used to connect to LEDs and DIP switches. The system frequency is 125 MHz.

#### *ML403 Hardware System*

The ML403 system shown in [Figure 2](#) utilizes the embedded PowerPC processor for running all software applications. The system has a xps\_ll\_temac core with a FIFO interface. The local-link interface of the xps\_ll\_temac is connected to the PLB via a xps\_ll\_fifo adapter. In such a system

with a FIFO, the processor is responsible for all data transfers from the EMAC to the memory controller (MPMC). The PIT (Programmable Interval Timer) within the PowerPC is used as a source of periodic interrupts, so there is no need for an external timer. The xps\_gpio cores connect to the LEDs and DIP switches. In this system, the PowerPC is clocked at 300 MHz and the rest of the peripherals are clocked at 100 MHz.

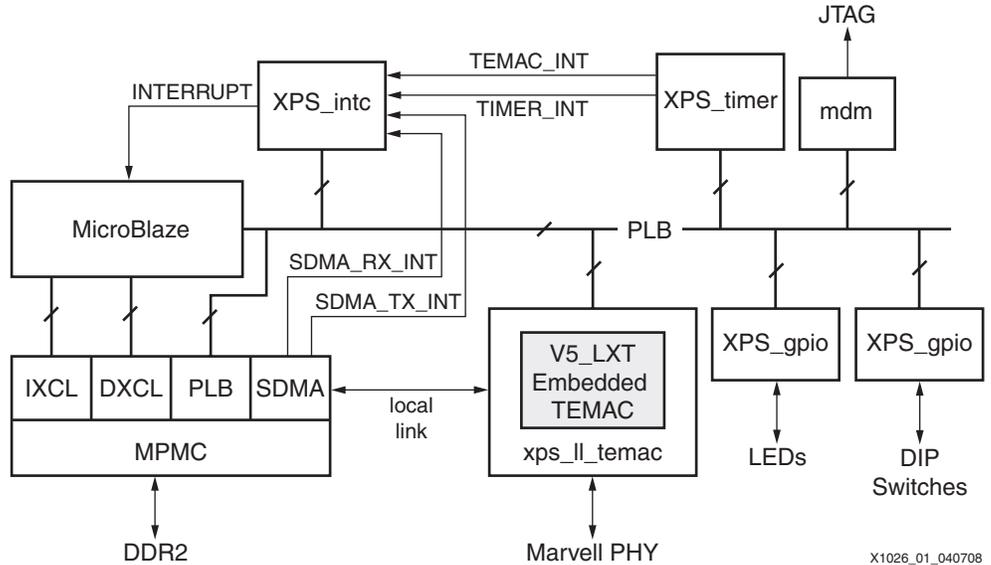


Figure 1: ML505 Hardware System

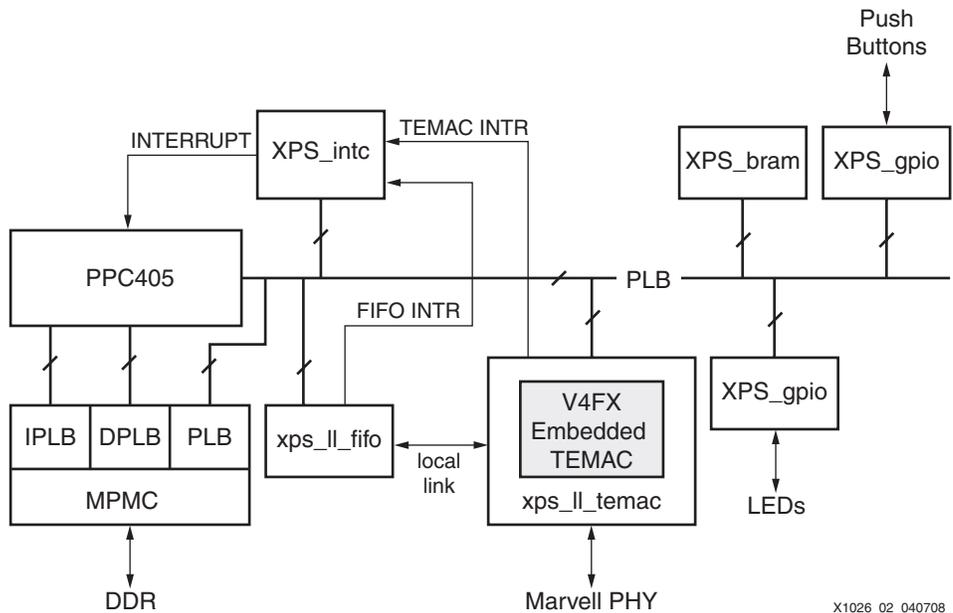


Figure 2: ML403 Hardware System

### Spartan-3AN Starter Kit Hardware System

The Spartan-3AN starter kit system shown in Figure 3 is the simplest of the three systems. It uses the small xps\_ethernetlite EMAC IP to provide ethernet connectivity. The xps\_ethernetlite core supports 10/100 Mbps link speeds only, and can have a maximum of 2 kb of internal buffer to hold packets. The other components are the same as in the ML505 system. The system frequency is 62.5 MHz.

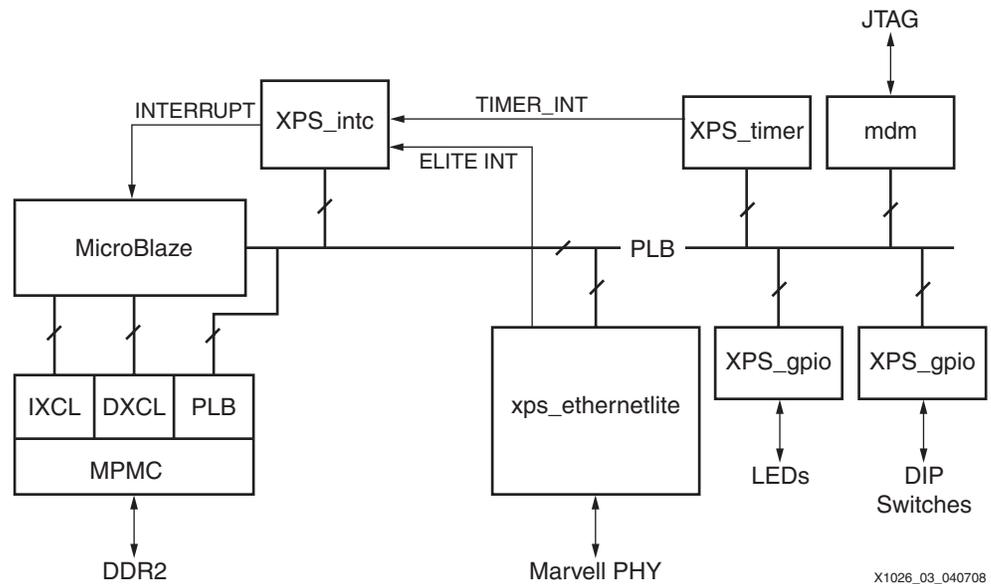


Figure 3: Spartan-3AN Starter Kit Hardware System

## Software Applications

The reference design includes three software applications: echo server, web server and a TFTP server. The applications are available in RAW and Socket modes.

### Echo Server

The echo server is a simple program that echoes whatever input is sent to the program via the network. This application provides a good starting point for investigating how to write lwIP applications.

The socket mode echo server is structured as follows. A main thread listens continually on a specified echo server port. For each connection request, it spawns a separate echo service thread, and then continues listening on the echo port.

```
while (1) {
    new_sd = lwip_accept(sock, (struct sockaddr *)&remote, &size);
    sys_thread_new(process_echo_request, (void*)new_sd,
    DEFAULT_THREAD_PRIO);
}
```

The echo service thread receives a new socket descriptor as its input on which it can read received data. This thread does the actual echoing of the input to the originator.

```
while (1) {
    /* read a max of RECV_BUF_SIZE bytes from socket */
    n = lwip_read(sd, recv_buf, RECV_BUF_SIZE);

    /* handle request */
    nwrote = lwip_write(sd, recv_buf, n);
}
```

**Note:** The above code snippets are not complete and are intended to show the major structure of the code only.

The socket mode provides a simple API that blocks on socket reads and writes until they are complete. However, the socket API requires many pieces to achieve this, chief among them being a simple multithreaded kernel (xilkernel). Because this API contains significant overhead for all operations, it is slow.

The RAW API provides a callback style interface to the application. Applications using the RAW API register callback functions to be called on significant events like accept, read or write. A RAW API based echo server is single threaded, and all the work is done in the callback functions. The main application loop is structured as follows.

```
while (1) {
    xemacif_input(netif);
    transfer_data();
}
```

The function of the application loop is to receive packets constantly (`xemacif_input`), then pass them on to lwIP. Before entering this loop, the echo server sets up certain callbacks:

```
/* create new TCP PCB structure */
pcb = tcp_new();

/* bind to specified @port */
err = tcp_bind(pcb, IP_ADDR_ANY, port);

/* we do not need any arguments to callback functions */
tcp_arg(pcb, NULL);

/* listen for connections */
pcb = tcp_listen(pcb);

/* specify callback to use for incoming connections */
tcp_accept(pcb, accept_callback);
```

This sequence of calls creates a TCP connection and sets up a callback on a connection being accepted. When a connection request is accepted, the function `accept_callback` is called asynchronously. Because an echo server needs to respond only when data is received, the accept callback function sets up the receive callback by performing:

```
/* set the receive callback for this connection */
tcp_recv(newpcb, recv_callback);
```

When a packet is received, the function `recv_callback` is called. The function then echoes the data it receives back to the sender:

```
/* indicate that the packet has been received */
tcp_recved(tpcb, p->len);

/* echo back the payload */
err = tcp_write(tpcb, p->payload, p->len, 1);
```

Although the RAW API is more complex than the SOCKET API, it provides much higher throughput because it does not have a high overhead.

## Web Server

A simple web server implementation is provided as a reference for a TCP based application. The web server implements only a subset of the HTTP 1.1 protocol. Such a web server can be used to control or monitor an embedded platform via a browser. The web server demonstrates the following three features:

- Accessing files residing on a Memory File System via HTTP GET commands
- Controlling the LED lights on the development board using the HTTP POST command
- Obtaining status of DIP switches (push buttons on the ML403) on the development board using the HTTP POST command

The Xilinx Memory File System (xilmfs) is used to store a set of files in the memory of the development board. These files can then be accessed via a HTTP GET command by pointing a web browser to the IP address of the development board and requesting specific files.

Controlling or monitoring the status of components in the board is done by issuing POST commands to a set of URLs that map to devices. When the web server receives a POST command to a URL that it recognizes, it calls a specific function to do the work that has been requested. The output of this function is sent back to the web browser in Javascript Object Notation (JSON) format. The web browser then interprets the data received and updates its display.

The overall structure of the web server is similar to the echo server – there is one main thread which listens on the HTTP port (80) for incoming connections. For every incoming connection, a new thread is spawned which processes the request on that connection.

The http thread first reads the request, identifies if it is a GET or a POST operation, then performs the appropriate operation. For a GET request, the thread looks for a specific file in the memory file system. If this file is present, it is returned to the web browser initiating the request. If it is not available, a HTTP 404 error code is sent back to the browser.

In socket mode, the http thread is structured as follows:

```
/* read in the request */
if ((read_len = read(sd, recv_buf, RECV_BUF_SIZE)) < 0)
    return;

/* respond to request */
generate_response(sd, recv_buf, read_len);
```

Pseudo code for the generate response function is shown below:

```
/* generate and write out an appropriate response for the http request */
int generate_response(int sd, char *http_req, int http_req_len)
{
    enum http_req_type request_type =
        decode_http_request(http_req, http_req_len);

    switch(request_type) {
    case HTTP_GET:
        return do_http_get(sd, http_req, http_req_len);
    case HTTP_POST:
        return do_http_post(sd, http_req, http_req_len);
    default:
        return do_404(sd, http_req, http_req_len);
    }
}
```

The RAW mode webserver does most of its work in callback functions. When a new connection is accepted, the `accept` callback function sets up the send and receive callback functions. These are called when sent data has been acknowledged or when data is received.

```
err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    /* keep a count of connection # */
    tcp_arg(newpcb, (void*)palloc_arg());

    tcp_recv(newpcb, recv_callback);
    tcp_sent(newpcb, sent_callback);

    return ERR_OK;
}
```

When a web page is requested, the `recv_callback` function is called. This function then performs work similar to the socket mode function – decoding the request and sending the appropriate response.

```
/* acknowledge that we've read the payload */
tcp_recved(tpcb, p->len);

/* read and decipher the request */
/* this function takes care of generating a request, sending it,
 * and closing the connection if all data has been sent. If
 * not, then it sets up the appropriate arguments to the sent
 * callback handler.
 */
generate_response(tpcb, p->payload, p->len);

/* free received packet */
pbuf_free(p);
```

The complexity lies in how the data is transmitted. In the socket mode, the application sends data using the `lwip_write` API. This function blocks if the TCP send buffers are full. However in RAW mode, the application assumes the responsibility of checking how much data can be sent and of sending that much data only. Further data can be sent only when space is available in the send buffers. Space becomes available when sent data is acknowledged by the receiver (the client computer). When this event happens, lwIP calls the `sent_callback` function, indicating that data was sent which implies that there is now space in the send buffers for more data. The `sent_callback` is hence structured as follows.

```
err_t sent_callback(void *arg, struct tcp_pcb *tpcb, u16_t len)
{
    int BUFSIZE = 1024, sndbuf, n;
    char buf[BUFSIZE];
    http_arg *a = (http_arg*)arg;

    /* if connection is closed, or there is no data to send */
    if (tpcb->state > ESTABLISHED) {
        return ERR_OK;
    }

    /* read more data out of the file and send it */
    sndbuf = tcp_sndbuf(tpcb);
    if (sndbuf < BUFSIZE)
        return ERR_OK;

    n = mfs_file_read(a->fd, buf, BUFSIZE);
    tcp_write(tpcb, buf, n, 1);

    /* update data structure indicating how many bytes
     * are left to be sent
    */
}
```

```
    */
    a->fsize -= n;
    if (a->fsize == 0) {
        mfs_file_close(a->fd);
        a->fd = 0;
    }

    return ERR_OK;
}
```

Both the sent and the receive callbacks are called with an argument which can be set using `tcp_arg`. For the web server, this argument points to a data structure which maintains a count of how many bytes remain to be sent and what is the file descriptor that can be used to read this file.

## TFTP server

TFTP (Trivial File Transfer Protocol) is a UDP based protocol for sending and receiving files. Because UDP does not guarantee reliable delivery of packets, TFTP implements a protocol to ensure packets are not lost during transfer. Detailed explanation of the TFTP protocol can be found in the [RFC 1350 – The TFTP Protocol](#).

The socket mode TFTP server is very similar to the web server in application structure. A main thread listens on the TFTP port and spawns a new TFTP thread for each incoming connection request. This TFTP thread implements a subset of the TFTP protocol and supports either read or write requests. At the most, only one TFTP Data or Acknowledge packet can be in flight which greatly simplifies the implementation of the TFTP protocol. Because the RAW mode TFTP server is very simplistic and does not handle timeouts, it is usable only as a point to point ethernet link with zero packet loss. It is provided as a demonstration only.

Because TFTP code is very similar to the web server code explained above, is not explained in this application note. Because of the use of UDP, the minor differences can be easily understood by looking at the source code.

## Creating an lwIP Application Using the Socket API

The software applications provide a good starting point to write other applications using lwIP. lwIP socket API is very similar to the Berkeley/BSD sockets, therefore, there should be no issues writing the application itself. The only difference lies in the initialization process which is coupled to `lwip_v3_00_a` library and `xilkernel`.

The three sample applications all utilize a common `main.c` file for initialization and to start processing threads. Perform the following steps for any socket mode application.

1. Configure the `xilkernel` with a static thread. In the sample applications, this thread is given the name `main_thread`.
2. The main thread initializes lwip using the `lwip_init` function call, and then launches the network thread using the `sys_thread_new` function. Note that all threads that use the lwIP socket API must be launched with the `sys_thread_new` function provided by lwIP.
3. The main thread adds a network interface using the `xemac_add` helper function. This function takes in the IP address and the MAC address for the interface, and initializes it.
4. The `xemacif_input_thread` is then started by the network thread. This thread is required for lwIP operation when using the Xilinx adapters. This thread takes care of moving data received from the interrupt handlers to the `tcpip_thread` that is used by lwIP for TCP/IP processing.
5. The lwIP has now been completely initialized and further threads can be started as the application requires.

## Creating an lwIP application using the RAW API

The lwIP RAW mode API is more complicated to use as it requires knowledge of lwIP internals. The typical structure of a RAW mode program is as follows.

1. The first step is to initialize all lwIP structures using `lwip_init`.
2. Once lwIP has been initialized, an EMAC can be added using the `xemac_add` helper function.
3. Because the Xilinx lwIP adapters are interrupt based, enable interrupts in the processor and in the interrupt controller.
4. Set up a timer should to interrupt at a constant interval. Usually, the interval is around 250 ms. Update the tcp timers at every timer interrupt.
5. Once the application is initialized, the main program enters an infinite loop performing packet receive operation, and any other application specific operation it needs to do.
6. The packet receive operation (`xemacif_input`), processes packets received by the interrupt handler, and passes them onto lwIP, which then calls the appropriate callback handlers for each received packet.

## Executing the Reference System

This section describes how to execute the system and the expected results

The designs in the reference system have been cleaned of all temporary files. A copy of the reference bitstream and a set of executables are available in the `ready_for_download` folder.

### Host Network Settings

Connect the FPGA board to an Ethernet port on the host computer via a cross-over cable. Assign an IP address to the ethernet interface on the host computer. The address must be the same subnet as the IP address assigned to the board. The software application assigns a default IP address of 192.168.1.10 to the board. The address can be changed in the file `apps/src/main.c`. For this setting, assign an IP address to the host in the same subnet mask, for example 192.168.1.100.

### Executing the Reference System using the Pre-Built Bitstream and the Compiled Software Applications

To execute the system using files in the `ready_for_download/` directory in the project root directory, follow these steps:

1. Change directories to the `ready_for_download` directory.
2. Use iMPACT to download the bitstream by using the following:
 

```
impact -batch xapp1026.cmd
```
3. Invoke XMD and connect to the processor by using the command:
 

```
xmd -opt xapp1026.opt
```
4. Download the executables by using the command:
 

```
dow executable.elf
```

Note that these executables have been compiled with an IP address of 192.168.1.10, and a MAC address of 00:00:00:01:02:03. The host computer must have an IP address in the same subnet. The name of the executable (`executable.elf`) will change based on the software application that will be downloaded. In socket mode, there is a single executable "`demo.elf`" which contains all three applications. In RAW mode, each application is compiled into a separate ELF file. Hence there will be three ELF files named `echo.elf`, `web.elf`, and `tftp.elf` corresponding to the echo, web and TFTP servers respectively.

## Executing the Reference System from EDK

To execute the system using EDK, follow these steps:

1. Open `system.xmp` inside EDK.
2. Use **Hardware** → **Generate Bitstream** to generate a bitstream for the system.
3. Download the bitstream to the board with **Device Configuration** → **Download Bitstream**.
4. Launch XMD with **Debug** → **Launch XMD...**
5. Download the executables by the following command:

```
dow executable.elf
```

### Running the Echo Server Application

When the echo server application is run, it displays the following output on the RS232 screen.

```
-----lwIP TCP echo server -----
TCP packets sent to port 6001 will be echoed back
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
XLlTEmac detect_phy: No PHY detected. Assuming a PHY at address 0
auto-negotiated link speed: 1000
tcp echo server started @ port 7
```

Any echo client can be used to send data to the board, and examine the received response. The simplest client that can be used is the telnet client that is available in most host machines (Windows or Linux). The following sequence of commands illustrates how to connect to the board and how to send data.

```
$ telnet 192.168.1.10 7
Trying 192.168.1.10...
Connected to 192.168.1.10.
Escape character is '^]'.
hello world
hello world
all messages will be echo'ed back
all messages will be echo'ed back
^]
telnet> quit
Connection closed. $
```

If the echo server works properly, any data sent to the board will be echoed in response as seen above. Note that certain telnet clients will immediately send the character to the server and echo the received data back rather than waiting for the carriage return.

## Running the Web Server Application

Download the memory file system image to connect to the web server application. This file is needed in addition to the web server.

The Memory File system image can be created as follows by launching it from the EDK cygwin shell:

```
$ cd memfs
memfs$ mfsngen -cvbfs ../image.mfs 1500*
mfsngen
Xilinx EDK 9.2.02 EDK_Jm_SP2.3
Copyright (c) 2004 Xilinx, Inc. All rights reserved.

css:
main.css 744
images:
```

```
ml505.jpg 44176
logo.gif 1148
index.html 2914
js:
main.js 7248
yui:
conn.js 11633
yahoo.js 5354
anim.js 12580
event.js 14309
dom.js 10855
MFS block usage (used / free / total) = 227 / 373 / 600
Size of memory is 319200 bytes
Block size is 532
mfsgen done!
```

mfsgen is a utility that is shipped along with EDK. The above command creates a single file system image (`image.mfs`) of all the contents in the `memfs` folder.

Once this file system image is created, it can be downloaded to the board using XMD.

```
XMD% dow -data image.mfs 0x01f00000
System Reset ... DONE
Downloading Data File -- image.mfs at 0x01f00000

XMD% dow apps/web.elf
XMD% con
```

Note that the `image.mfs` file must be loaded into memory at a location which is not used by the ELF file itself. In the snippet above, the MFS file was downloaded to address `0x01f00000`. This address should point to an available location in external memory that does not overlap with the ELF file. In addition, MFS should be configured to look for the image at this address. This is done by configure `xilmfs` library in software platform settings.

Now the web server is up and running, and should show the following status of the RS232 terminal.

```
-----lwIP test WebServer -----
Open up your favorite browser and type:
http://192.168.1.10
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
XLlTemac detect_phy: No PHY detected. Assuming a PHY at address 0
auto-negotiated link speed: 1000
Memory File System initialized
web server started
```

Once the web server is active, it can be connected to using a web browser. The sample web pages use Javascript, so the browser must have javascript enabled. A sample screen shot of the web page that is served is shown in [Figure 4](#).

The Toggle LED button will toggle the state of the LEDs on the board. Clicking the Update Status button will refresh the status of the DIP switches on the web page. These are intended to show simple control and monitoring of the embedded platform via the ubiquitous web browser.

Note that the external links section contains links that point to content that are not served by the development platform.

## Running the TFTP Server Application

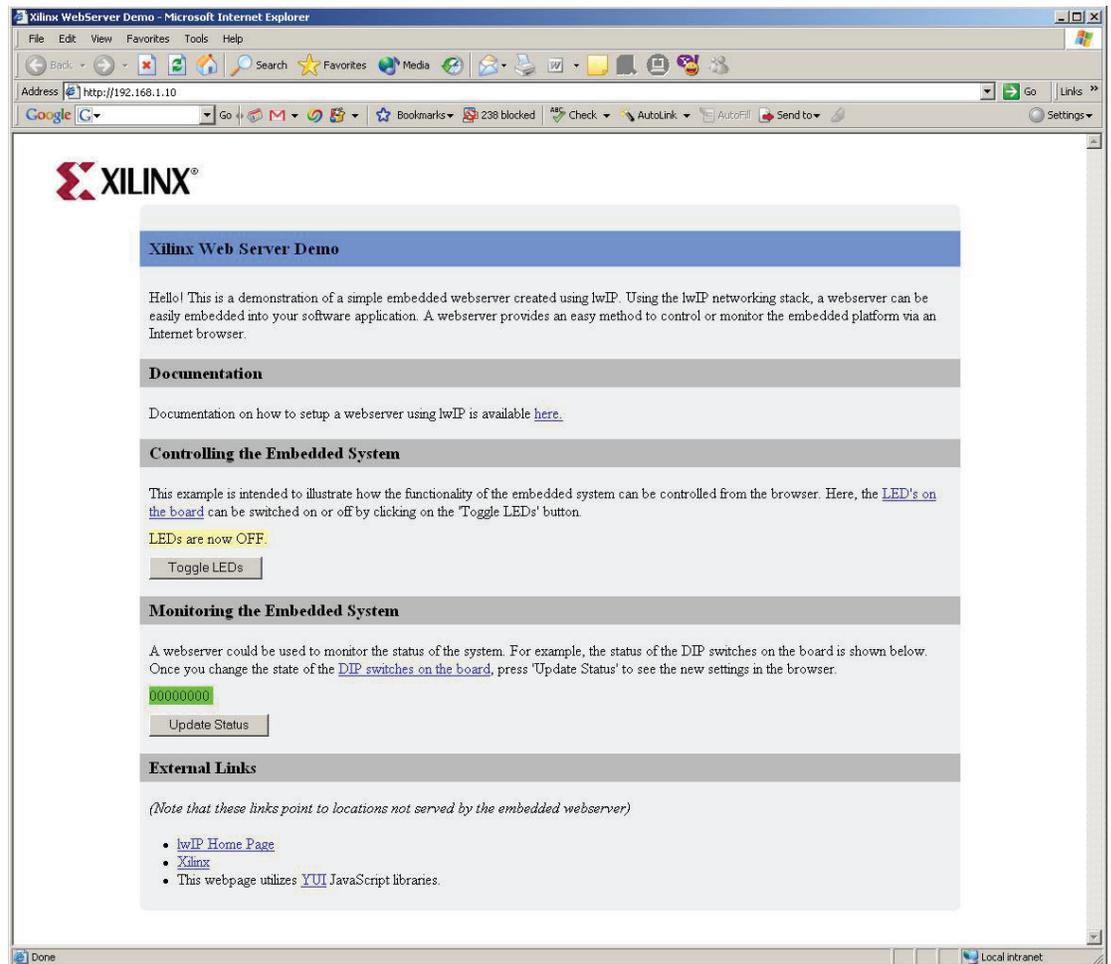
The TFTP server provides simple file transfer capability to and from the memory file system resident on the board. To connect to the TFTP server, first download the image.mfs MFS image created for the web server application, and then download and run the TFTP server application. When the software is run, it should produce the following output on the RS232 terminal:

```
-----lwIP TFTP server -----
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
XLlTEmac detect_phy: No PHY detected. Assuming a PHY at address 0
auto-negotiated link speed: 1000
tftp server started @ port 69
```

Now, a TFTP client can be used to send or receive files from the memory file system. An example interaction is shown below using the TFTP client on Windows.

```
C:\>tftp -i 192.168.1.10 GET index.html
Transfer successful: 2914 bytes in 1 second, 2914 bytes/s
C:\>tftp -i 192.168.1.10 PUT test.txt
Transfer successful: 19 bytes in 1 second, 19 bytes/s
```

The above two examples show how to read or write files on the board's memory file system from the local host.



X1026\_04\_040708

Figure 4: Web Page served by the Reference Web Server

## lwIP Performance

This section provides a brief overview of the expected performance when using lwIP with Xilinx Ethernet MACs. Details regarding the benchmarking process and the systems used will be made available in a separate whitepaper.

Table 2: RAW Mode

FPGA	CPU	EMAC	System Frequency	Max TCP Throughput
Virtex <sup>®</sup> -4	PPC405	xps_ll_temac	100 MHz	100 Mbps
Virtex-5	MicroBlaze	xps_ll_temac	125 MHz	100 Mbps
Spartan-3	MicroBlaze	xps_ll_temac	66 MHz	40 Mbps
Spartan-3	MicroBlaze	xps_ethernetlite	66 MHz	20 Mbps

Applications that require high performance can utilize the RAW mode to achieve performance in excess of 100 Mbps.

Table 3: Socket Mode

FPGA	CPU	EMAC	System Frequency	Max TCP Throughput
Any	Any	Any	Any	1 Mbps

The Xilinx adapters for socket mode are not optimized, and provide only about 1 Mbps across all configurations. The performance in socket mode will be resolved in subsequent releases.

## Debugging Network Issues

If any of the sample applications do not work, there could be a number of potential reasons. This section provides a troubleshooting guide to fix common sources of setup errors.

1. The first step is to ensure that the link lights are active. Most development boards have LEDs that indicate whether an ethernet link is active. If the bitstream downloaded has some ethernet MAC properly configured, and a ethernet cable is attached to the board, the link lights should indicate an established ethernet link.
2. If the board include LEDs indicating the link speed (10/100/1000 Mbps), it should be verified that the link is established at the correct speed. For designs that include xps\_ethernetlite EMAC IP, the link should be established at only 10 or 100 Mbps. xps\_ethernetlite will not be able to transmit or receive data at 1000 Mbps. The xps\_ll\_temac EMAC core supports all three link speeds. The TEMAC must be informed of the correct speed to which the PHY has auto-negotiated. lwIP includes software to detect the PHY speed, however this software works only for Marvell PHYs. Users should confirm that the link speed that lwIP detects matches the link speed as shown in the LEDs.
3. To confirm that the board actually receives packets, the simplest test is to ping the board, and check to make sure that the RX LED goes high for a moment to indicate that the PHY actually received the packet. If the LEDs do not go high, then there are either ARP, IP, or link level issues that prevent the host from sending packets to the board.
4. Assuming that the board receives the packets, but the system does not respond to ping requests, the next step is to ensure that lwIP actually receives these packets. This can be determined by setting breakpoints at `XEmacLite_InterruptHandler` for xps\_ethernetlite systems, and `lldma_recv_handler` for xps\_ll\_temac systems. If packets are received properly, then these breakpoints should be hit for every received packet. If these breakpoints are not hit, then that indicates that the MAC is not receiving the packets. This could mean that the packets are being dropped at the PHY. The most

common reason that the breakpoints are not hit, is that the link was established at a speed that the EMAC does not support.

5. Finally, some hosts have firewalls enabled that could prevent receiving packets back from the network. If the link LEDs indicate that the board is receiving and transmitting packets, yet the packets transmitted by the board are not received in the host, then the host firewall settings should be relaxed.

---

## Conclusion

This application note showcases how lwIP can be used to develop networked applications for embedded systems on Xilinx FPGAs. The echo server provides a simple starting point for networking applications. The web server application shows a more complex TCP based application, and the TFTP server shows a complex UDP based application.

## References

1. [lwIP – A Lightweight TCP/IP Stack– CVS Repositories](#)
2. [RFC 1350 – The TFTP Protocol](#)

---

## Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
4/11/08	1.0	Initial Xilinx release.

---

## Notice of Disclaimer

Xilinx is disclosing this Application Note to you “AS-IS” with no warranty of any kind. This Application Note is one possible implementation of this feature, application, or standard, and is subject to change without further notice from Xilinx. You are responsible for obtaining any rights you may require in connection with your use or implementation of this Application Note. XILINX MAKES NO REPRESENTATIONS OR WARRANTIES, WHETHER EXPRESS OR IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL XILINX BE LIABLE FOR ANY LOSS OF DATA, LOST PROFITS, OR FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR INDIRECT DAMAGES ARISING FROM YOUR USE OF THIS APPLICATION NOTE.