XAPP1026 (v3.2)
October 28, 2012

# LightWeight IP (lwIP) Application Examples

Author: Anirudha Sarangi and Stephen MacMahon

## Summary

Lightweight IP (lwIP) is an open source TCP/IP networking stack for embedded systems. The Xilinx Software Development Kit (SDK) provides lwIP software customized to run on various Xilinx embedded systems that can be MicroBlaze™ or PowerPC® processor based. The Xilinx SDK provided lwIP software can also be run on ARM® based Xilinx Zynq™ SoC systems.The information in this application notes applies to MicroBlaze processors and ARM-based Zynq SoC systems. 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 these applications: echo server, Web server, TFTP server and receive and transmit throughput tests.

## Included Systems

Included with this application note are AXI4-based reference systems for the Xilinx ML605 and SP605 FPGA Starter Kit boards. Also included are Zynq SoC reference systems for the Xilinx ZC-702 boards. To access these reference systems, click the following link.

https://secure.xilinx.com/webreg/clickthrough.do?cid=107743.zip

## Hardware and Software Requirements

The hardware and software requirements are:

- One of the Xilinx ML605, or SP605 for MicroBlaze processor based systems and a ZC-702 board for Zynq SoC-based systems.
- Xilinx Platform USB cable for MicroBlaze processor-based systems and Xilinx JTAG for Zynq SoC based systems
- USB cable for RS232 UART communication on the board
- An ethernet cable connecting the board to a Windows or Linux host
- Serial Communications Utility Program, such as HyperTerminal or Teraterm
- Xilinx Platform Studio 14.3 for making hardware modifications
- Xilinx SDK 14.3 for running or making modifications to the software

## 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 SDK to add networking capability to an embedded system. In particular, this application note describes how applications such as an echo server or a Web server can be written using lwIP.

The lwIP library released as part of 14.3 and used in the reference designs use the open source lwIP version lwip 1.4.0.

## Reference System Specifics

The reference design for this application note is structured as follows:

- ml605_AxiEth_8Kb_Cache, ml605_AxiEth_32kb_Cache, sp605_AxiEth_8kb_Cache, sp605_AxiEth_32kb_Cache, sp605_EthernetLite_8kb_Cache, sp605_EthernetLite_32kb_Cache, and zc702_GigE folders correspond to the various supported boards and hardware designs.
- In each of these folders, the hw subdirectory contains the XPS hardware design, and the sw subdirectory contains the application software and software platforms that need to be imported into SDK.
- There is a folder with the name 13_1, which contains the older designs released in the previous version of xapp1026 and which were tested with the 13.1 release. These designs are provided for easy reference. This document does not discuss these older designs anywhere in the subsequent sections.
- The ready_for_download folder contains these relevant files for getting started with the applications:
  - download.bit: bitstream to be downloaded to the board. Applicable only for MicroBlaze processor based systems.
  - system_bd.bmm: for downloading the bitstream. Applicable only for MicroBlaze processor-based systems.
  - image.mfs: memory file system used for tftp, Web server applications
  - image_BIG.mfs: provided only in the ready_for_download folder for the zc702_GigE hardware system. This is a large mfs file that can be used for Zynq device web server applications and shows the high-speed performance of Zynq devices.
  - raw_apps.elf: application ELF image to be downloaded to test the RAW API
  - socket_apps.elf: application ELF image to be downloaded to test the socket API
  - fsbl.elf: first stage boot loader applicable only for Zynq SoC systems to initialize the MIOs/Clocks.
- The memfs folder contains the contents of the memory file system (MFS) image.
- The image itself is also present as the image.mfs file in the respective ready_for_download folders. For Zynq devices, two mfs files are provided. The standard image.mfs (common for all boards) is present along with a large mfs file (image_BIG.mfs) that contains huge web page contents. However, memfs contents for this image_BIG.mfs is not provided.
- A repo folder is provided for Zynq SoC systems (zc702_GigE folder) that contains the FreeRTOS BSP based out of latest standalone BSP v3_07_a.

## Hardware Systems

The hardware systems for the available boards were built using Base System Builder (BSB), with minor modifications in XPS. For more information on hardware requirements for lwIP, see the lwIP documentation available as part of the SDK installation. Table 1 provides a summary of the hardware designs for the available boards.

*Table 1:* **Hardware Design Details**

| Hardware Design Name | Processor | Processor Frequency | Ethernet Controller | DMA |
|---|---|---|---|---|
| ml605_AxiEth_8Kb_Cache | MICROBLAZE | 100 MHz | axi_ethernet | axi_dma |
| ml605_AxiEth_32kb_Cache | MICROBLAZE | 100 MHz | axi_ethernet | axi_dma |
| sp605_AxiEth_8kb_Cache | MICROBLAZE | 100 MHz | axi_ethernet | axi_dma |
| sp605_AxiEth_32kb_Cache | MICROBLAZE | 100 MHz | axi_ethernet | axi_dma |
| sp605_EthernetLite_8kb_Cache | MICROBLAZE | 75 MHz | axi_ethernetlite | NONE |
| sp605_EthernetLite_32kb_Cache | MICROBLAZE | 75 MHz | axi_ethernetlite | NONE |
| zc702_GigE | Cortex-A9 | 666.667 MHz | GigE | In-built |

**Notes:**

1. All axi_ethernet based systems are built with full checksum (both TCP and IP checksums) offload feature.
2. The AXI Ethernetlite based systems are built with PING-PONG buffers (one more buffer for both TX and RX paths to improve performance).
3. The GigE based systems (Zynq devices) has in-built TCP/IP checksum offload support.

# Software Applications

The reference design includes these software applications:

- Echo server
- Web server
- TFTP server
- TCP RX throughput test
- TCP TX throughput test

All of these applications are available in both RAW and socket modes.

## Echo Server

The echo server is a simple program that echoes input that 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:

1. A main thread listens continually on a specified echo server port.
2. For each connection request, it spawns a separate echo service thread.
3. It 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:*** These 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, including primarily a simple multi-threaded kernel (xilkernel for MicroBlaze processor based systems and FreeRTOS for Zynq systems). 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) {
  if (TcpFastTmrFlag) {
    tcp_fasttmr();
    TcpFastTmrFlag = 0;
  }
  if (TcpSlowTmrFlag) {
    tcp_slowtmr();
    TcpSlowTmrFlag = 0;
  }
  xemacif_input(netif);
  transfer_data();
}
```

The *TcpFastTmrFlag* and *TcpSlowTmrFlag* are required for TCP TX handling and are set in the Timer handler for every 250 milliseconds and 500 milliseconds respectively.

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 these 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 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 an 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 that 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 as follows:

```
/* 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 Web server primarily uses callback functions to perform its tasks. 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 tasks similar to the socket mode function – decoding the request and sending the appropriate response.

```
/* acknowledge that we have 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 data transmission is complex. 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 determines how much data can be sent and sends only that much data. 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 occurs, lwIP calls the `sent_callback` function, indicating that data was sent and there is now space in the send buffers for more data. The `sent_callback` is 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 that can be set using `tcp_arg`. For the Web server, this argument points to a data structure that 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. See the RFC 1350 – The TFTP Protocol for a detailed explanation of 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 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 previously, it is not explained in this application note. The use of UDP allows the minor differences to be understood by examining the source code.

## TCP RX Throughput Test and TCP TX Throughput Test

The TCP transmit and receive throughput test applications are very simple applications that determine the maximum TCP transmit and receive throughputs achievable using lwIP and the Xilinx EMAC adapters. These tests communicate with an open source software called iperf ((http://sourceforge.net/projects/iperf/)).

The transmit test measures the transmission throughput from the board running lwIP to the host. In this test, the lwIP application connects to an iperf server running on a host, and then keeps sending a constant piece of data to the host. Iperf running on the host determines the rate at which data is transmitted and prints it out on the host terminal.

The receive test measures the maximum receive transmission throughput of data at the board. The lwIP application acts as a server. This server accepts connections from any host at a certain port. It receives data sent to it, and silently drops the received data. Iperf (client mode) on the host connects to this server and transmits data to it for as long as needed. At frequent intervals, it computes how much data is transmitted at what throughput and prints this information on the console.

## 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. Consequently, there should be no issues writing the application itself. The only difference is in the initialization process that is coupled to the lwip140 library and xilkernel (or FreeRTOS).

The sample application utilizes a common `main.c` file for initialization and to start processing threads. Perform these steps for any socket mode application.

1. For MicroBlaze processor based systems that use Xilkernel, configure the Xilkernel with a static thread. In the sample applications, this thread is named `main_thread`. In addition, make sure Xilkernel is properly configured by specifying the system interrupt controller. For Zynq SoC based systems that use FreeRTOS, create the first task with the name `main_thread` before starting the FreeRTOS scheduler. See `main.c` for the socket application to know the details for task/thread initializations for Xilkernel/FreeRTOS.
2. The main thread initializes lwip using the `lwip_init` function call, and then launches the network thread using the `sys_thread_new` function. 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 ethernet 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 handles moving data received from the interrupt handlers to the `tcpip_thread` that is used by lwIP for TCP/IP processing.
5. The lwIP library 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 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. After 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 to interrupt at a constant interval. Usually, the interval is around 250 ms. In the timer interrupt, update necessary flags to invoke the lwIP TCP APIs tcp_fasttmr and tcp_slowtmr from the main application loop explained previously.
5. After 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 reference design and the expected results.

***Note:*** This section provides details specifically for the ML605_AXI design. The steps are the same for the other designs, except for the address at which the memory file system (MFS) is loaded. The correct address for loading the MFS image is determined by looking at the corresponding software platform settings for xilmfs library. This section assumes that the relevant systems present in the `xapp1026_14_3` folder are copied into C:\`xapp1026_14_3` folder.

### Host Network Settings

1. Connect the relevant board to an Ethernet port on the host computer via an Ethernet cable.
2. 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 respective `main.c` files. For this setting, assign an IP address to the host in the same subnet mask, for example `192.168.1.100`.

### Compiling and Running the Software

The reference applications can be compiled and run using SDK with these steps.

1. Open SDK in a new workspace by providing a suitable name and location.
2. Create a local repo for the FreeRTOS BSP. This is applicable only for Zynq devices.
3. Import the software platform and software applications to automatically compile both the software platform and the applications.
4. Download the bitstream. Applicable only for MicroBlaze processor based systems.
5. Download the MFS image.
6. Create a run configuration and run the application.

Follow the same steps to import and run any application using SDK. For more details regarding SDK concepts and tasks, see the online help in SDK.

These five steps are explained in detail in the following paragraphs.

#### Step 1: Specify the Workspace

Eclipse organizes projects within a folder called workspace. In SDK, a workspace can only contain projects for one specific hardware platform. When SDK starts up, specify a folder to contain software projects for a particular hardware design.

#### Step 2: Create a local repository for FreeRTOS BSP

This is applicable only for a Zynq SoC project. The FreeRTOS BSP is not delivered as part of the ISE® tools 14.3 release. It is available through FreeRTOS website as a community supported port.

Following are the steps to download the latest available FreeRTOS port for Zynq devices.

1. Go to www.freertos.org.

2. Select **FreeRTOS Interactive** from the index items available on the left.
3. Select **Upload/Download Contributions**.
4. Scroll down and select Xilinx.
5. Select **Updated Xilinx FreeRTOS port for Zynq to SDK 14.2 release**.
6. Download Xilinx_Zynq.zip.
7. Unzip it to get the FreeRTOS BSP ported for Zynq devices.

***Note:*** At the time of preparing this application note, the FreeRTOS BSP available at the FreeRTOS website is based on standalone BSP version 3_06_a.

The repo folder available for Zynq SoC systems (zc702_GigE folder) contains the latest FreeRTOS BSP which is based out of the latest standalone BSP in 14.3 (3_07_a). To build and execute the socket applications for Zynq devices, you need to create a local repository that contains the FreeRTOS BSP. This BSP will then be used by SDK to build the socket applications. To create a local repo in SDK, do the following.

1. Select **Xilinx Tools > Repositories** in SDK. A new window pops up.
2. Click the **New** tab for local repositories.
3. Assuming that the folders are downloaded and put at a Windows location C:\xapp1026_14_3, then the FreeRTOS BSP will be present C:\xapp1026_14_3\zc702_GigE\repo\bsp\freertos_zynq_v1_01_a. You need to browse until C:\xapp1026_14_3\zc702_GigE\repo. This is the path for the local repository.
4. Click the **Rescan Repositories** tab.
5. Click the **Apply** tab.
6. Click the **OK** tab.

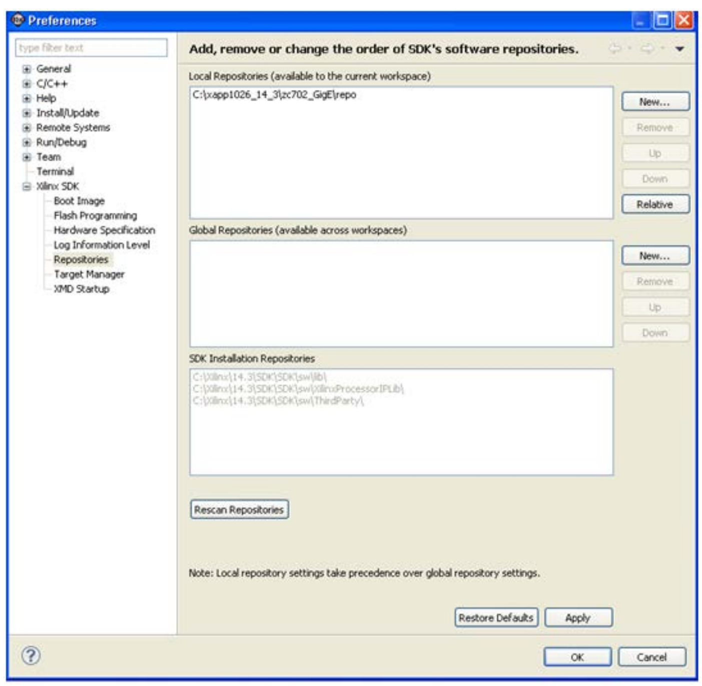



*Figure 1:* **Creating a local repo for FreeRTOS BSP (applicable only for Zynq)**

### Step 3: Import Software Projects

Software platforms and applications can be created in SDK after the hardware platform is specified. Instead of creating a new software platform/application, import the existing software platforms and example applications provided with this reference design using these steps:

1. Select **File** > **Import** to open an import wizard.
2. Select **General** > **Existing SDK Projects into Workspace** in the import wizard.
3. To select the root directory from which the projects need to be imported, click **Browse** and specify the location where the software applications are stored. Assuming that the projects are downloaded at Windows C:\xapp1026_14_3, for the zc702_GigE design, this Windows location is C:\xapp1026_14_3\zc702_GigE\sw folder.
4. The import wizard displays a list of projects that are available to import. This list should include: fsbl (applicable for Zynq devices only and for other systems it will not show up) hw_platform_0, raw_apps, raw_bsp, sock_apps, and sock_bsp. Select all these projects to be imported. Select the option **Copy projects into workspace**, if you want local copies of these projects in the workspace. Finally select **Finish**.

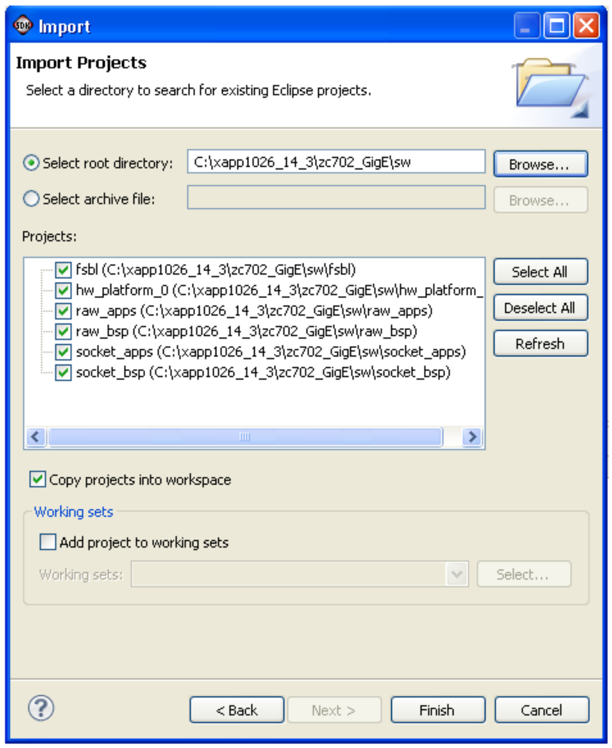



*Figure 2:* **Select Folder to Import Projects**

5. As soon as the import is done, SDK by default starts building all available software applications. Normally there should be no errors. In case of any build errors, you can clean all the projects from SDK and then rebuild them.

### Step 3: Download the Bitstream (Applicable only for MicroBlaze Processor Systems)

To download the bitstream, select **Xilinx Tools** > **Program FPGA** to display the Program FPGA GUI dialog box. Ensure that both the `system.bit` file and the `system_bd.bmm` file locations are proper, and then click **Program**.

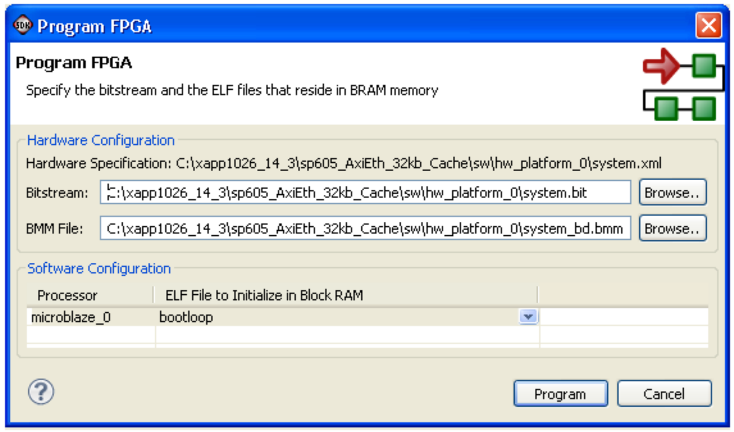



*Figure 3:* **Programming the FPGA**

### Step 4: Download the MFS Image

The memory file system image contains the files required for the Web server to serve files from, and for the TFTP server to store and retrieve files. The image must be downloaded to the onboard DDR memory before the executable can run properly. To download the MFS image for MicroBlaze-based systems, select **Xilinx Tools > XMD Console**. From within the XMD, navigate to the location where the `image.mfs` file has been placed (in `ready_for_download` folder).

From this location, download the image with the applicable command:

- For the ml605_AXI system, use:
  - XMD% connect mb mdm
  - XMD% dow -data image.mfs 0xDF000000
- For the sp605_AXI system, use:
  - XMD% connect mb mdm
  - XMD% dow -data image.mfs 0xAF000000

The preceding procedure does not apply to Zynq devices. For downloading `image.mfs` for Zynq devices, See the following section (step 5). See Appendix A: Creating an MFS Image for instructions on how to create the MFS `image.mfs`.

### Step 5: Create a Run Configuration and Run the Application

To run the application, use these steps:

1. Create a run configuration specifying the ELF that needs to be run:
   a. Select the application that needs to be run.
   b. To create a Run configuration, select **Run** > **Run Configuration**.
   c. Create a new run configuration by right clicking the **Xilinx C/C++ ELF** tab on the left pane.
   d. Browse and specify the Project details.

e. Ensure that the specified ELF is appropriate. If `raw_apps` is selected (in step a), the ELF should be `raw_apps.elf` and if socket_apps is selected (in step a), the ELF should be `socket_apps.elf`.

f. This step applies only for the Zynq SoC project. Click the **Device Initialization** tab. Ensure that the path to the initialization TCL file is correct. Assuming that the projects are downloaded to the folder C:\xapp1026_14_3 folder and the user workspace name is "myws_GigE" to which all the imported folders are copied, the tcl path should be C:\xapp1026_14_3\myws_GigE\hw_platform_0\ps7_init.tcl.

   Similarly the `image.mfs` file path and address should be specified properly in the section for **Specify data files to be downloaded before launch**. You must ensure that the path specified for this case is C:\xapp1026_14_3\zc702_GigE\ready_for_download\image.mfs and the address specified is 0x7200000. See Figure 4 (Run Configurations settings for Zynq Devices).

2. Select **Apply** and **Run** to run the executable.

   It might take a while (and a long time for Zynq SoC systems depending on the size of the `image.mfs` being used) before the ELF (and `image.mfs` for Zynq devices) is downloaded to hardware and starts running.

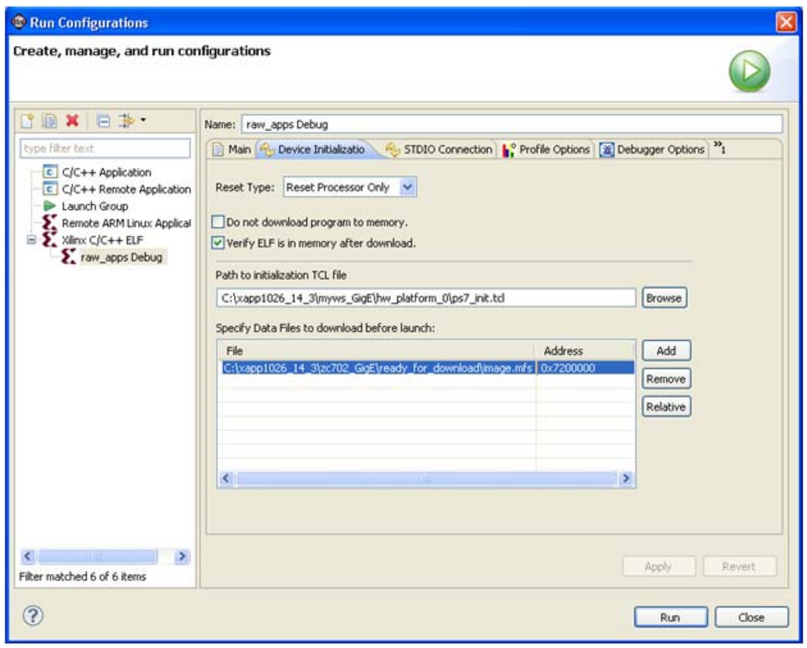



*Figure 4:* **Run Configuration Settings For Zynq**

**Important Notes**:

- The contents for memfs (and hence `image.mfs`) are for demonstration purposes and are not updated for ml605/sp605/zc702 boards. They are provided to show that the web server demonstration example works fine and that you can browse a web page through this example.
- For ml605/sp605 systems, the board pictures and hence the location of LEDs and DIP switches are for older Xilinx FPGA boards. However, the LEDs can still be toggled from the host (using the web page). Similarly the DIP switch status show up properly for ml605/sp605 boards.
- For the zc702 board, the contents for displayed web page are just for demonstration purposes. The pictures of boards are not for the zc702 board. The LEDs cannot be toggled using the web page. Similarly the zc702 board does not contain any DIP switches whose status can be shown up on the web page.
- For Zynq devices, one more mfs file with the name `image_BIG.mfs` (of 8 MB size) is also provided. You can rename it to `image.mfs` and download it at the same location and use the same settings explained previously. This is provided to display the greater speed for Zynq devices. The web pages downloaded through this `image.mfs` by the host PC are large in size. However, this web page is not suitable for running the tftp example (as the corresponding memfs file is not part of this application note release).

## Running the .ELF from XMD

You can directly use the files provided in the `ready_for_download` folder and run them on a relevant board without using SDK to download and run them. You need to do the following:

1. Go to the XMD prompt.
2. Go to the `ready_for_download folder` that contains the download.bit for the relevant board.
3. Execute **`fpga -f download.bit`**. This is applicable only for MicroBlaze-based systems and is used to program the bitstream.
4. Execute **`connect mb mdm`** (for MicroBlaze processors) or **`connect arm hw`** (for Zynq devices).
5. For Zynq SoC systems, fsbl needs to be downloaded to configure the MIO and clocks. Execute **`dow fsbl.elf`** to download the elf. Execute **`con`** to run the fsbl. Execute **`stop`** to stop running the fsbl.
6. Download the `image.mfs` to the appropriate addresses by executing the command **`dow -data image.mfs`** *appropriate_address*.

   The addresses to which the `image.mfs` is to be downloaded for MicroBlaze processor based systems are already specified in Step 4 under the section Executing the Reference System. For Zynq devices, the address to which the `image.mfs` is to be downloaded is 0x7200000.
7. Download the applications by executing **`dow raw_apps.elf`** or **`dow socket_apps.elf`**.
8. Run the downloaded applications by executing the command **`con`**.

## Interacting with the Running Software

The socket mode and the RAW mode applications bundle the following examples into a single executable: echo server, Web server, TFTP server, receive and transmit throughput tests.

### Output From the Application

After the executable is run, this output appears on the serial port:

```
-----lwIP RAW Mode Demo Application ------
Board IP:       192.168.1.10
Netmask :       255.255.255.0
Gateway :       192.168.1.1
auto-negotiated link speed: 1000

              Server   Port Connect With..
-------------------- ------ --------------------
         echo server      7 $ telnet <board_ip> 7
       rxperf server   5001 $ iperf -c <board ip> -i 5 -t 100
       txperf client    N/A $ iperf -s -i 5 -t 100 (on host with IP
192.168.1.100)
         tftp server     69 $ tftp -i 192.168.1.10 PUT <source-file>
         http server     80 Point your web browser to http://192.168.1.10
```

For the socket mode application, only the first line changes to indicate that it is the socket mode demonstration application. Now you can interact with the application running on the board from the host machine.

### Interacting With the Echo Server

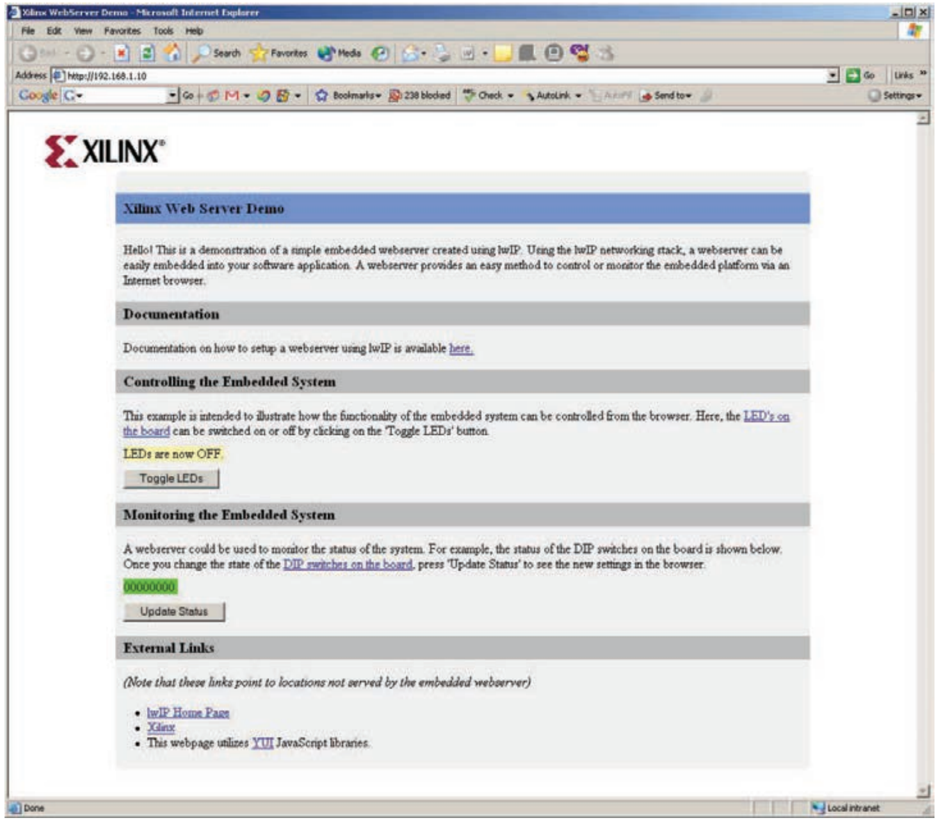
To connect to the echo server, use the telnet utility program.

```
$ telnet 192.168.1.10 7
Trying 192.168.1.10...
Connected to 192.168.1.10.
Escape character is '^]'.
hello
hello
world world ^]
telnet> quit
Connection closed.
```

If the echo server works properly, any data sent to the board is echoed in response. Some telnet clients immediately send the character to the server and echo the received data back instead of waiting for the carriage return.

### Interacting With the Web Server

After 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 Web page that is served is shown in Figure 5. The Toggle LED button toggles the state of the LEDs on the board. Clicking the **Update Status** button refreshes the status of the DIP switches on the Web page. These show simple control and monitoring of the embedded platform via the Web browser. The external links section contains links that point to content that are not served by the development platform.




X1026_03_060409

*Figure 5:* **Web Page Served by the Reference Web Server**

## Interacting With the TFTP Server

The TFTP server provides simple file transfer capability to and from the memory file system resident on the board. The following examples use the TFTP client on Windows, and show how to read or write files on the board memory file system from the local host.

```
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
```

### Interacting With the Receive Throughput Test

To measure receive throughput, connect to the receive iperf application using the iperf client by issuing the **iperf -c** command with relevant options. A sample session (with zc702_GigE as reference) is as follows:

```
C:\>iperf -c 192.168.1.10 -i 5 -t 50 -w 64k
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 64.0 KByte
------------------------------------------------------------
[  3] local 192.168.1.100 port 51644 connected with 192.168.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 5.0 sec   562 MBytes   944 Mbits/sec
[  3]  5.0-10.0 sec   563 MBytes   945 Mbits/sec
[  3] 10.0-15.0 sec   562 MBytes   942 Mbits/sec
[  3] 15.0-20.0 sec   563 MBytes   945 Mbits/sec
[  3] 20.0-25.0 sec   561 MBytes   941 Mbits/sec
[  3] 25.0-30.0 sec   564 MBytes   945 Mbits/sec
[  3] 30.0-35.0 sec   558 MBytes   935 Mbits/sec
[  3] 35.0-40.0 sec   562 MBytes   942 Mbits/sec
[  3] 40.0-45.0 sec   561 MBytes   941 Mbits/sec
[  3] 45.0-50.0 sec   563 MBytes   945 Mbits/sec
[  3]  0.0-50.0 sec  5.49 GBytes   943 Mbits/sec
```

***Note:*** To achieve maximum throughput numbers, ensure that the executable has been compiled for "-O2" optimization level rather than "-O0" optimization. The EthernetLite systems, use a TCP window size of 8k instead of 64k to get maximum numbers.

### Interacting With the Transmit Throughput Test

To measure the transmit throughput, start the iperf server on the host, and then run the executable on the board. When the executable is run, it attempts to connect to a server at host 192.168.1.100. This address can be changed in the `txperf.c` file. A sample session (with zc702_GigEas reference) is as follows:

```
C:\>iperf -s -i 5 -w 64k
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 64.0 KByte
------------------------------------------------------------
[  4] local 192.168.1.100 port 5001 connected with 192.168.1.10 port 49153
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 5.0 sec   485 MBytes   814 Mbits/sec
[  4]  5.0-10.0 sec   485 MBytes   814 Mbits/sec
[  4] 10.0-15.0 sec   481 MBytes   807 Mbits/sec
[  4] 15.0-20.0 sec   481 MBytes   807 Mbits/sec
[  4] 20.0-25.0 sec   485 MBytes   814 Mbits/sec
[  4] 25.0-30.0 sec   485 MBytes   814 Mbits/sec
[  4] 30.0-35.0 sec   484 MBytes   812 Mbits/sec
[  4] 35.0-40.0 sec   480 MBytes   806 Mbits/sec
[  4] 40.0-45.0 sec   484 MBytes   812 Mbits/sec
[  4] 45.0-50.0 sec   482 MBytes   809 Mbits/sec
[  4] 50.0-55.0 sec   480 MBytes   806 Mbits/sec
^C Waiting for server threads to complete. Interrupt again to force quit.
^C
```

Press **Ctrl+C** twice to stop the server.

***Note:*** For EthernetLite based systems, use a TCP window size of 8k instead of 64k to get maximum performance numbers.

## lwIP Performance

The receive and transmit throughput applications are used to measure the maximum TCP throughput possible with lwIP using the Xilinx ethernet adapters. Table 2 summarizes the results for different configurations. Depending upon different cache configurations, the performance numbers can vary.

*Table 2:* **: TCP Performance Numbers**

| Hardware Design Name | RAW Mode | | Socket Mode | |
|---|---|---|---|---|
| | RX (Mb/s) | TX (Mb/s) | RX (Mb/s) | TX (Mb/s) |
| ml605_AxiEth_8Kb_Cache | 142 | 95 | 32 | 37 |
| ml605_AxiEth_32kb_Cache | 204 | 170 | 42 | 52 |
| sp605_AxiEth_8kb_Cache | 137 | 91 | 31 | 35 |
| sp605_AxiEth_32kb_Cache | 200 | 165 | 39 | 50 |
| sp605_EthernetLite_8kb_Cache | 37 | 28 | 18 | 19 |
| sp605_EthernetLite_32kb_Cache | 52 | 35 | 22 | 24 |
| zc702_GigE | 943 | 810 | 490 | 550 |

These performance numbers were obtained under these conditions:

- All designs are BSB designs, using clocks described previously in this application note.
- When measuring receive throughput, only the receive throughput application was enabled (in file `config_apps.h`). This also applies to the transmit throughput test.
- The host machine was a Dell desktop running Windows 7. The NIC card used on the host was an Intel 82566DM-2 Gigabit Ethernet Controller card.
- The performance numbers for RAW Mode TX for MicroBlaze processor based systems were obtained using zero-copy pbufs.

***Note:*** The RAW mode TX performance numbers for MicroBlaze processor systems were obtained using zero-copy pbufs. Zero-copy pbuf means that lwIP stack will not be copying the user buffer to a pbuf and a reference to the user buffer will be used instead of copying it. The assumption here is that until the point the corresponding Ethernet frame is transmitted out and pbuf is freed up, you are not going to alter the contents of this buffer. For the given tx perf applications it is not an issue and hence zero-copy pbufs are used. This avoids a copy of the buffer and hence improves performance. For Zynq SoC systems, the RAW mode TX performance is already high and using a zero-copy pbuf does not really give a significant improvement. Hence for Zynq SoC systems, zero-copy pbufs are not used. All other raw mode applications (echo server, web server and tftp server) do not use the zero-copy pbuf feature. Also, for socket mode applications there is no provision for zero-copy pbufs.

For MicroBlaze processor based RAW mode TX perf applications, you can change the source code in file `txperf.c` to avoid using zero-copy pbufs. This can be done by ensuring that when a tcp_write is called, the third argument passed has a value of 3 instead of 0 (present by default for MicroBlaze processor based systems).

***Note:*** By making this change, you might need to revisit the lwip140 settings in the BSP.

For information on how to optimize the host setup, and benchmarking TCP in general, see *XAPP1043 - Measuring Treck TCP/IP Performance Using the XPS LocalLink TEMAC in an Embedded Processor System.*

## DHCP Support

All applications published as part of this document are built with DHCP (Dynamic Host Configuration Protocol) support. The applications assume that there is a DHCP server available in the network that will assign an IP address to the relevant board. However, if a DHCP server is not available in the network to which the board is connected, a DHCP timeout occurs after a while. Upon a DHCP timeout a static IP address of 192.168.1.10 is assigned to the relevant board.

In a typical case when a user PC (which does not have a DHCP server running on it) is connected back-to-back to any of the boards supported in this application note, a DHCP timeout occurs and a static IP of 192.168.1.10 is assigned to the board. You can run all the available applications specified previously in this application note on the host PC (assumed to have an IP address of 192.168.1.100).

If you want to run iperf on a host PC connected to a LAN (which has a DHCP server) to which the relevant board is connected, a small change is needed in the file `txperf.c`. The current implementations of `txperf.c` assume that the iperf server has an IP address of 192.168.1.100. This address needs to be changed appropriately in file `txperf.c`.

Finally you can disable the DHCP server in the applications by altering the lwip140 settings in the relevant BSPs.

## JUMBO Frame Support

All MicroBlaze/Axi-Ethernet systems available as part of this application note are built with jumbo frame support. However, you need to change some of the BSP settings for lwip140 (on top of the existing settings provided in the BSPs in this application note) to actually send jumbo frames.

- Change the value of the lwip140 parameter "temac_use_jumbo_frames" under the category "temac_adapter_options" to true.
- Change the value of the lwip140 parameter "pbuf_pool_bufsize" under the category "pbuf_options" from the default 1700 to 9700.
- Change the value of the lwip140 parameter "ip_reass_bufsize" under the category "lwip_ip_options" from the default 5760 to 65535.
- Change the value of the lwip140 parameter "ip_frag_max_mtu" under the category "lwip_ip_options" from the default 1500 to 9000.
- Change the value of the lwip140 parameter "tcp_mss" under the category "tcp_options" from the default 1460 to 8060.
- Change the value of the lwip140 parameter "mem_size" under the category "lwip_memory_options" from the default 131072 to 524288.

To measure iperf performance numbers (with a host PC connected back-to-back with the relevant board), you must do the following:

- Ensure that jumbo frame support is enabled on the host PC NIC card/driver.
- To run iperf server on the host PC, you must execute the command "iperf -s -i 5 -l 65535 -w 1048576 -M 8060
- To run iperf client on the host PC, you must execute the command "iperf -c 192.168.1.10 -i 5 -t 50 -w 64k -M 8060 -l 128k
- The raw and socket mode applications provided in the respective folders for HW systems need not be changed for performance measurement using jumbo frame support. The necessary changes related to jumbo frames are already present.

Typical performance numbers obtained for the hardware system (ml605_AxiEth_32kb_Cache) using raw mode applications were 700 Mb/s for TX path and 850 Mb/s on the RX path. Similarly typical performance numbers obtained for the HW system (ml605_AxiEth_32kb_Cache) using socket mode applications were 190 Mb/s for TX path and 100 Mb/s on the RX path.

## Debugging Network Issues

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

1. First, 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 includes LEDs indicating the link speed (10/100/1000 Mb/s), verify that the link is established at the correct speed. For designs that include axi_ethernetlite EMAC IP, the link should be established at only 10 or 100 Mb/s. The axi_ethernetlite cannot transmit or receive data at 1000 Mb/s. The axi_ethernet 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 axi_ethernetlite systems, and axi dma_recv_handler for axi_ethernet systems, and emacps_recv_handler for Zynq SoC 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.

When these applications are ported over to a different board or hardware, ensure there is sufficient heap and stack space available (as specified in the linker script).

## Conclusion

This application note showcases how lwIP can be used to develop networked applications for embedded systems on Xilinx FPGAs and Xilinx Zynq SoC systems. 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. Applications to measure receive and transmit throughput provide an indication of the maximum possible throughput using lwIP with Xilinx adapters.

## Appendix A: Creating an MFS Image

To create an MFS image from the contents of a folder (memfs), use the relevant command from the SDK bash shell. To open a SDK bash shell, select **Xilinx Tools** > **Launch Shell**. At the command prompt, use the appropriate commands to go to the `memfs` directory (in xapp1026_14_3).

For AXI4-based systems, enter this command:

```
$ mfsgen -cvbf ../image.mfs 2048 css images js yui generate-mfs index.html
```

A typical mfsgen output is as follows:

```
mfsgen
Xilinx EDK 14.3 EDK_P.40
Copyright (c) 2004 Xilinx, Inc. All rights reserved.
css:
main.css 744
images:
board.jpg 44176
favicon.ico 2837
logo.gif 1148
js:
main.js 7336
yui:
anim.js 12580
conn.js 11633
dom.js 10855
event.js 14309
yahoo.js 5354
generate-mfs 34
index.html 2966
MFS block usage (used / free / total) = 234 / 1814 / 2048
Size of memory is 1089536 bytes
Block size is 532
mfsgen done!
```

## References

See the following for more information:


1. lwIP – A Lightweight TCP/IP Stack– CVS Repositories
2. RFC 1350 – The TFTP Protocol
3. iperf software
4. XAPP1043 *Measuring Treck TCP/IP Performance Using the XPS LocalLink TEMAC in an Embedded Processor*

## Revision History

This table shows the revision history for this document:

| Date | Version | Description of Revisions |
|---|---|---|
| 10/13/08 | 1.0 | Initial Xilinx release. |
| 6/15/09 | 2.0 | Updated to v2.0 for IDS11.1. |
| 03/20/11 | 3.0 | Updated for AXI4 interface. Updated block size for mfsgen command from 1500 to 2012 to prevent errors. |
| 04/21/11 | 3.1 | Updated “Compiling and Updating the Software” section and Appendix A. |
| 10/28/12 | 3.2 | • Updated for Zynq SoC support<br>• Added sections for DHCP and jumbo support<br>• Added a section for using the files in ready_for_download folder from XMD without using SDK<br>• Updated performance numbers for provided systems. |

## Notice of Disclaimer