XAPP1026 (v3.1) April 21, 2011

LightWeight IP (lwIP) Application Examples

Author: Stephen MacMahon, Nan Zang, Anirudha Sarangi

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 Xilinx embedded systems containing either a PowerPC® or a MicroBlaze™ processor. The information in this application notes applies to MicroBlaze processors only. 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 PLB and AXI4-based reference systems for the Xilinx ML605, SP605, and SP601 FPGA Starter Kit boards:

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

Hardware and Software Requirements

The hardware and software requirements are:

- One of Xilinx ML605, SP605, or SP601 Development Boards
- Xilinx Platform USB Cable
- RS232 USB 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 13.1 for making hardware modifications
- Xilinx SDK 13.1 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.

Reference System Specifics

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

- The ML605_AXI, ML605_PLB, SP605_AXI, SP605_PLB, SP601_AXI, and SP601_PLB folders correspond to the three supported boards and six supported 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.
- The `sdk` folder contains the software drivers and lwIP stack that need to be imported into SDK before building the software applications.
- The `ready_for_download` folder contains these relevant files for getting started with the applications:
  - **download.bit**: bitstream to be downloaded to the board
  - **system_bd.bmm**: for downloading the bitstream
  - **image.mfs**: memory file system used for tftp, Web server applications
  - **raw_apps.elf**: application ELF image to be downloaded to test the RAW API (application programming interface)
  - **socket_apps.elf**: application ELF image to be downloaded to test the socket API
- 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.

Hardware Systems

The hardware systems for the three 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 SDK installation. Table 1 provides a summary of the hardware designs for the three boards:

Table 1: **Hardware Design Details**

Board	Processor	Processor Frequency	EMAC	DMA
ML605_AXI	MicroBlaze	100 MHz	axi_ethernet	AXIDMA
SP605_AXI	MicroBlaze	100 MHz	axi_ethernet	AXIDMA
SP601_AXI	MicroBlaze	100 MHz	axi_ethernetlite	None
ML605_PLB	MicroBlaze	100 MHz	xps_ll_temac	SDMA
SP605_PLB	MicroBlaze	83.33 MHz	xps_ll_temac	SDMA
SP601_PLB	MicroBlaze	83.33 MHz	xps_ethernetlite	None

Notes:

1. The ML605_AXI and SP605_AXI hardware systems support full checksum (both TCP and IP checksums) offload feature.
2. The ML605_PLB and SP605_PLB hardware systems support partial checksum (TCP checksum) offload feature.
3. The SP601_AXI and SP601_PLB systems are Ethernetlite systems and built with PING-PONG buffers.

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). 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 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 (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 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://iperf.sourceforge.net/).

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 lwip130 library and xilkernel.

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

1. 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. lwIP also requires yield functionality available in xilkernel only when the 'enhanced_features' parameter of xilkernel is turned on.
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 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. Update the tcp timers at every timer interrupt.
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 are copied from the `xapp1026_13_1` folder into `C:/Projects` folder (which is not absolutely necessary).

Host Network Settings

1. Connect the FPGA board to an Ethernet port on the host computer via a crossover 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. Import the local AXIDMA driver and lwIP stack (located in the `xapp1026_13_1/ml605_Axi/SDK/SDK_Export/repo` folder) into SDK. These local driver/lwip stacks implement new features that are not provided in the SDK 13.1 release, but that are used in this application note. The AXI4-based systems have a local AXIDMA driver and a local lwIP stack. The PLB based systems have only local lwIP stack (AXIDMA driver is not used in PLB based systems). The local AXIDMA driver uses the newly introduced MicroBlaze instruction “mbar” to provide a safe DMA access to memory. Similarly the local lwIP stack implements full checksum offload capability that is supported in 13.1 AXIEthernet hardware IP.
3. Import the software platform and software applications to automatically compile both the software platform and the applications.
4. Download the bitstream.
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 six 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: Import Local Drivers and Software Services into SDK

Local drivers and software services can be imported into SDK. This is useful for testing any local changes made in any of the drivers or changes made in software services (for example, lwIP stack). This can be achieved with these steps:

1. Select **Xilinx Tools** → **Repositories** to open the Preferences wizard.
2. Select **New** in the wizard and browse two levels above the directory where the driver resides. For example, if the local AXIDMA driver is located at C:\Projects\ml605_Axi\SDK\SDK_Export\repo\drivers\axidma_v3_00_a, browse to C:\Projects\ml605_Axi\SDK\SDK_Export.

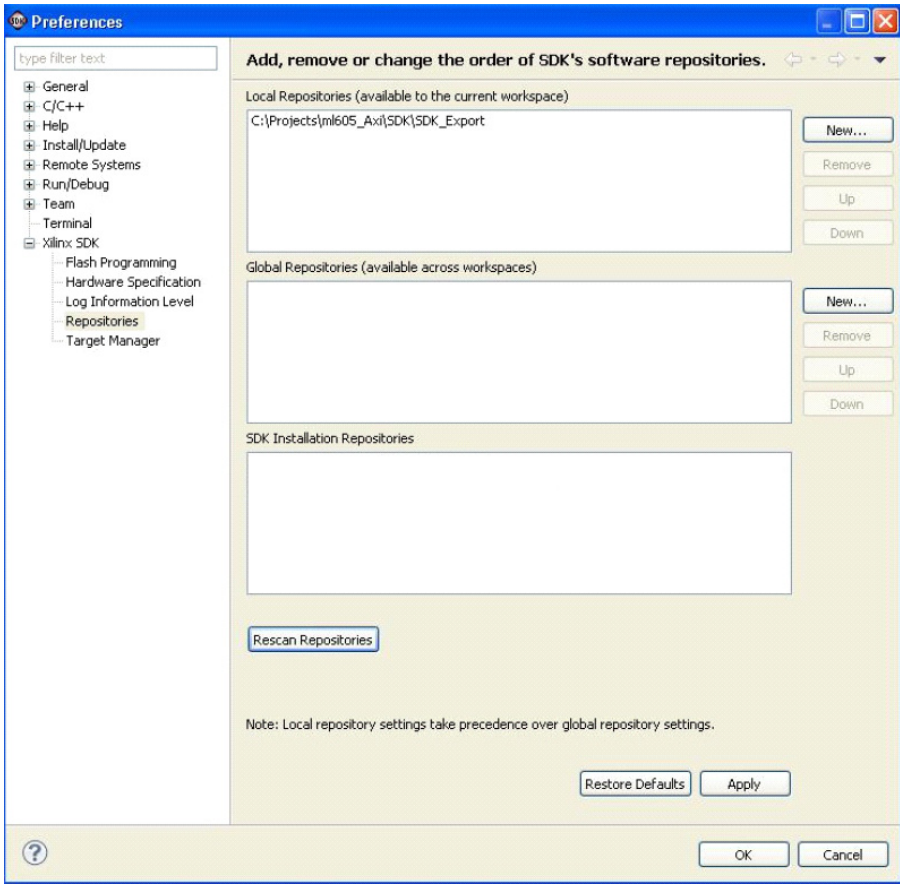


Figure 1: **Local Repositories**

3. Select **Rescan Repositories**.
4. Select **Apply** and then select **OK**.

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. For the ML605 design, this location is the `xapp1026_13_1/ml605/sw` folder.

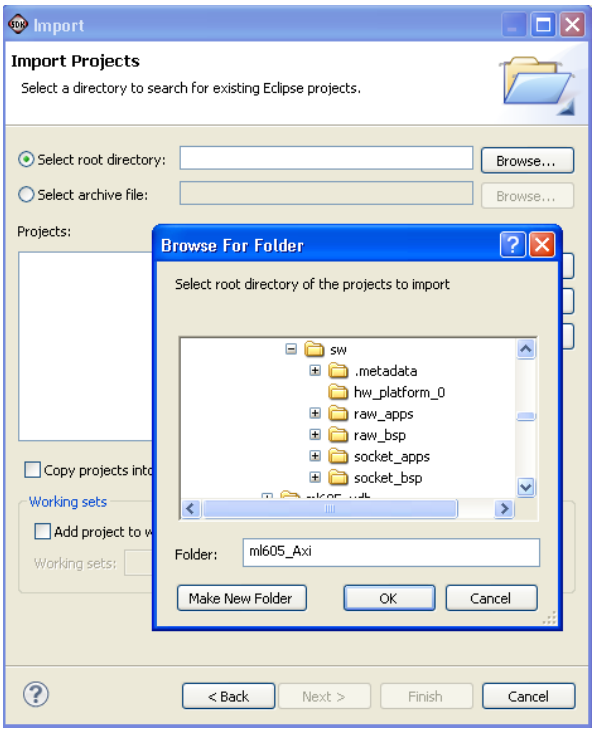


Figure 2: **Select Folder to Import Projects**

4. The import wizard displays a list of projects that are available to import. This list should include: hw_platform_0, raw_apps, raw_platform, sock_apps, and sock_platform. Select all five projects to be imported, and select **Finish**.

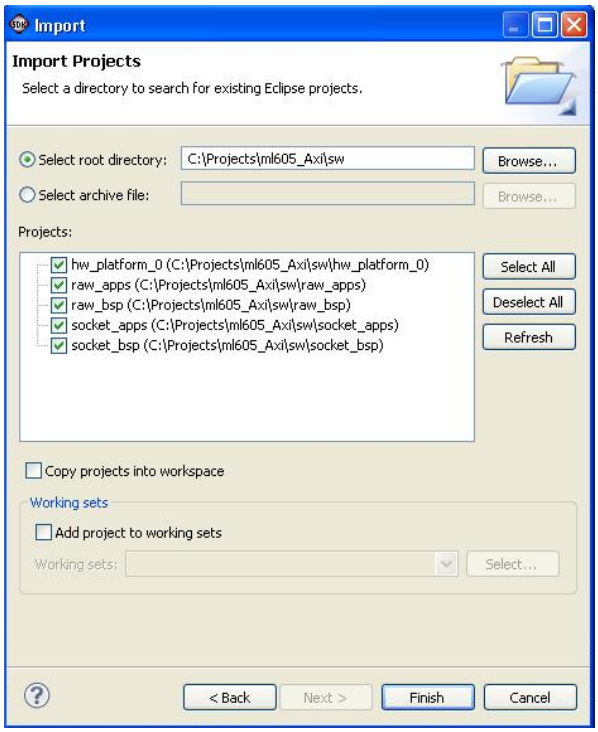


Figure 3: **Select Projects to Import**

5. Select **Yes** if the SDK prompts to overwrite an existing project file.

Step 4: Download the Bitstream

To download the bitstream, select **Tools** → **Program FPGA** to display the Program FPGA GUI dialog box. Specify both the `download.bit` file and the `system_bd.bmm` file found in the `ready_for_download` folder, and then click **Program**.

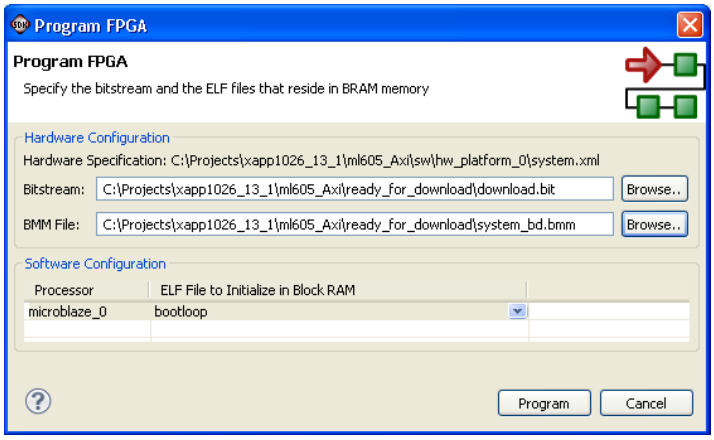


Figure 4: **Programming the FPGA**

Step 5: 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 external memory before The executable can run properly. To download the MFS image, select **Tools** → **XMD Console**. From within the XMD, navigate to the location where the `image.mfs` file has been placed (for example, cd /code/xapp1026_13_1/ml605_Axi/ready_for_download/). 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 0xCF000000`**
- For the ml605_PLB system, use:
  - **`XMD% connect mb mdm`**
  - **`XMD% dow -data image.mfs 0x5F000000`**
- For the sp605_AXI or sp601_AXI systems, use:
  - **`XMD% connect mb mdm`**
  - **`XMD% dow -data image.mfs 0xC5000000`**
- For the sp605_PLB or sp601_PLB systems, use:
  - **`XMD% connect mb mdm`**
  - **`XMD% dow -data image.mfs 0x5F000000`**

See Appendix A: Creating an MFS Image for instructions on how to create the MFS image.

Step 6: 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. To create a Run configuration, select **Run** → **Run Configuration**.
   b. Create a new run configuration by right clicking the **Xilinx C/C++ ELF** tab on the left pane.
   c. Browse and specify the Project details.

d. Browse and specify the ELF that needs to be run for the C/C++ application. If you want to run the raw mode example, use `raw_apps.elf`. If you want to use socket mode example, use `sock_apps.elf`.

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

   It might take a while before the ELF is downloaded to the hardware as SDK first builds all BSPs and applications.

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 demo application. At this point, you can interact with the application running on the board from the host machine.

Interacting With the Echo Server

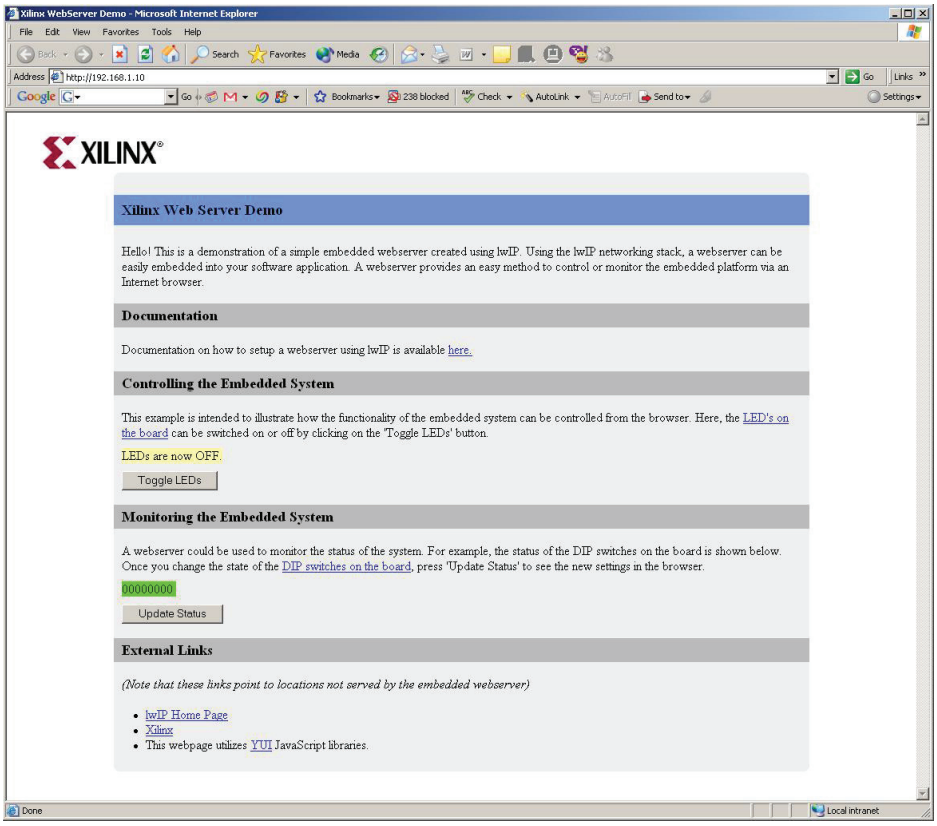
To connect to the echo server, use the telnet utility program.

```
$ 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's 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 ml605_PLB as reference) is as follows:

```
C:\>iperf -c 192.168.1.10 -i 5 -t 50 -w 8k
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 8.00 KByte
------------------------------------------------------------
[1912] local 192.168.1.100 port 1869 connected with 192.168.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[1912]  0.0- 5.0 sec  76.6 MBytes   129 Mbits/sec
[1912]  5.0-10.0 sec  76.9 MBytes   129 Mbits/sec
[1912] 10.0-15.0 sec  76.7 MBytes   129 Mbits/sec
[1912] 15.0-20.0 sec  76.9 MBytes   129 Mbits/sec
[1912] 20.0-25.0 sec  76.7 MBytes   129 Mbits/sec
[1912] 25.0-30.0 sec  76.9 MBytes   129 Mbits/sec
[1912] 30.0-35.0 sec  76.7 MBytes   129 Mbits/sec
[1912] 35.0-40.0 sec  76.1 MBytes   128 Mbits/sec
[1912] 40.0-45.0 sec  76.6 MBytes   129 Mbits/sec
[1912] 45.0-50.0 sec  76.9 MBytes   129 Mbits/sec
[1912]  0.0-50.0 sec   767 MBytes   129 Mbits/sec
```

Note: To achieve maximum throughput numbers, ensure that the executable has been compiled for "-O2" optimization level rather than "-O0" optimization.

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 ml605_PLB as reference) is as follows:

```
C:\>iperf -s -i 5
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 8.00 KByte (default)
------------------------------------------------------------
[1880] local 192.168.1.100 port 5001 connected with 192.168.1.10 port 4097
[ ID] Interval       Transfer     Bandwidth
[1880]  0.0- 5.0 sec  59.7 MBytes   100 Mbits/sec
[1880]  5.0-10.0 sec  62.2 MBytes   104 Mbits/sec
[1880] 10.0-15.0 sec  62.1 MBytes   104 Mbits/sec
[1880] 15.0-20.0 sec  62.2 MBytes   104 Mbits/sec
[1880] 20.0-25.0 sec  62.1 MBytes   104 Mbits/sec
[1880] 25.0-30.0 sec  62.2 MBytes   104 Mbits/sec
[1880] 30.0-35.0 sec  62.0 MBytes   104 Mbits/sec
[1880] 35.0-40.0 sec  62.2 MBytes   104 Mbits/sec
[1880] 40.0-45.0 sec  62.1 MBytes   104 Mbits/sec
[1880] 45.0-50.0 sec  62.2 MBytes   104 Mbits/sec
[1880] 50.0-55.0 sec  62.1 MBytes   104 Mbits/sec
^C Waiting for server threads to complete. Interrupt again to force quit.
^C
```

Press Ctrl + C twice to stop the server.

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 Receive and Throughput Results**

Design	Cache size (I-cache and D-cache sizes *are the same*)	RAW Mode		Socket Mode	
		RX (Mb/s)	TX (Mb/s)	RX (Mb/s)	TX (Mb/s)
ML605_AXI	8k	78.1	65.2	21.5	32
ML605_AXI	16k	101	80.7	24.4	38
ML605_AXI*	32k	128	104	27.5	46
ML605_PLB	8k	51.3	45.7	14.9	21.8
ML605_PLB	16k	73.9	63	17.8	26.9
ML605_PLB*	32k	129	104	24.1	39.3
SP605_AXI	8k	77.6	65	20.6	30.7
SP605_AXI	16k	93.1	81.5	23.5	37
SP605_AXI*	32k	125	102	27.4	46
SP605_PLB*	8k	51	44.2	14	21
SP605_PLB	16k	65.1	55.6	16.2	24.5
SP605_PLB	32k	97.8	81	20.4	32
SP601_AXI*	8k	39.4	25	16.1	17.8
SP601_AXI	16k	44	28.2	19.1	20.9
SP601_PLB*	8k	34	21	12.1	13.6
SP601_PLB	16k	34.4	21.2	12.4	14.0

* The reference designs for these systems along with the software projects are included in this Application Note (XAPP1026).

These performance numbers were obtained under these conditions:

- All designs are BSB designs, using the maximum clocks allowed in the BSB tool.
- 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 XP. The NIC card used on the host was based on Broadcom NetXtreme 57xx Gigabit Controller.
- For sp601, cache sizes of 32k are not possible because of resource constraints.

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.*

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. 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 xps_ethernetlite/axi_ethernetlite EMAC IP, the link should be established at only 10 or 100 Mb/s. The xps_ethernetlite/axi_ethernetlite cannot transmit or receive data at 1000 Mb/s. The xps_ll_temac/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 xps_ethernetlite/axi_ethernetlite systems, and lldma_recv_handler/ axidma_recv_handler for xps_ll_temac/axi_ethernet 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. 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_13_1).

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 13.1 EDK_O.40d
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!
```

For PLB based systems, enter this command:

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

The output is the same as the output for the AXI4-based system.

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.

Notice of Disclaimer