Application Note: AN00122

Using the XMOS embedded webserver library

This application note shows how to use XMOS webserver library to run an embedded webserver on an XMOS multicore microcontroller.

The code associated with the application note provides an example of using the XMOS HTTP Webserver library and the XMOS TCP/IP library to build an embedded webserver that hosts web pages. This example application relies on an Ethernet interface for the lower layer communication.

The HTTP Webserver library handles HTTP connections like GET and POST request methods, creates dynamic web page content and handles HTML pages as a file system stored in program memory or an external SPI flash memory device. The Webserver running on an xCORE device can be visited from a standard web browser of a computer that is connected to the same network to which the Ethernet interface of the xCORE development board is connected.

Embedding a web server onto XMOS multicore microcontrollers adds very flexible and easy-to-use management capabilities to your embedded systems.

Required tools and libraries

- xTIMEcomposer Tools - Version 13.1
- XMOS Embedded Webserver Library - Version 2.0.0
- XMOS TCP/IP Library - Version 4.0.0
- XMOS Ethernet Library - Version 3.0.0

Required hardware

This application note is designed to run on any XMOS xCORE device.

The example code provided with this application note has been implemented and tested on the SliceKIT Core Board (XP-SKC-L2) with Ethernet Slice (XA-SK-E100) and GPIO Slice (XA-SK-GPIO) but there is no dependency on these boards and it can be modified to run on any development board which has an xCORE device connected to an Ethernet PHY device through an MII (Media Independent Interface) interface.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the HTTP protocol, HTML, Webserver, the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the reference appendix.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary (http://www.xmos.com/published/glossary).
- For the full API listing of the XMOS HTTP Webserver library please see the document Embedded Webserver Library Programming Guide. (https://www.xmos.com/published/embedded-webserver-library-programming-guide)

1 Overview

1.1 Introduction

HTTP - Hypertext Transfer Protocol is the most prevalent application layer of the OSI model. It is the foundation of data communication for the World Wide Web. It works as a request-response protocol between a client and server. HTTP is used to transfer data like HTML pages for human interactions and XML or JSON format data for machine to machine interactions. The ubiquitous usage of HTTP has demanded networked embedded systems to host HTTP based services to provide configuration, diagnostic and management. The XMOS embedded webserver library enables the use of xCORE devices for applications that require these capabilities.

1.2 Block diagram

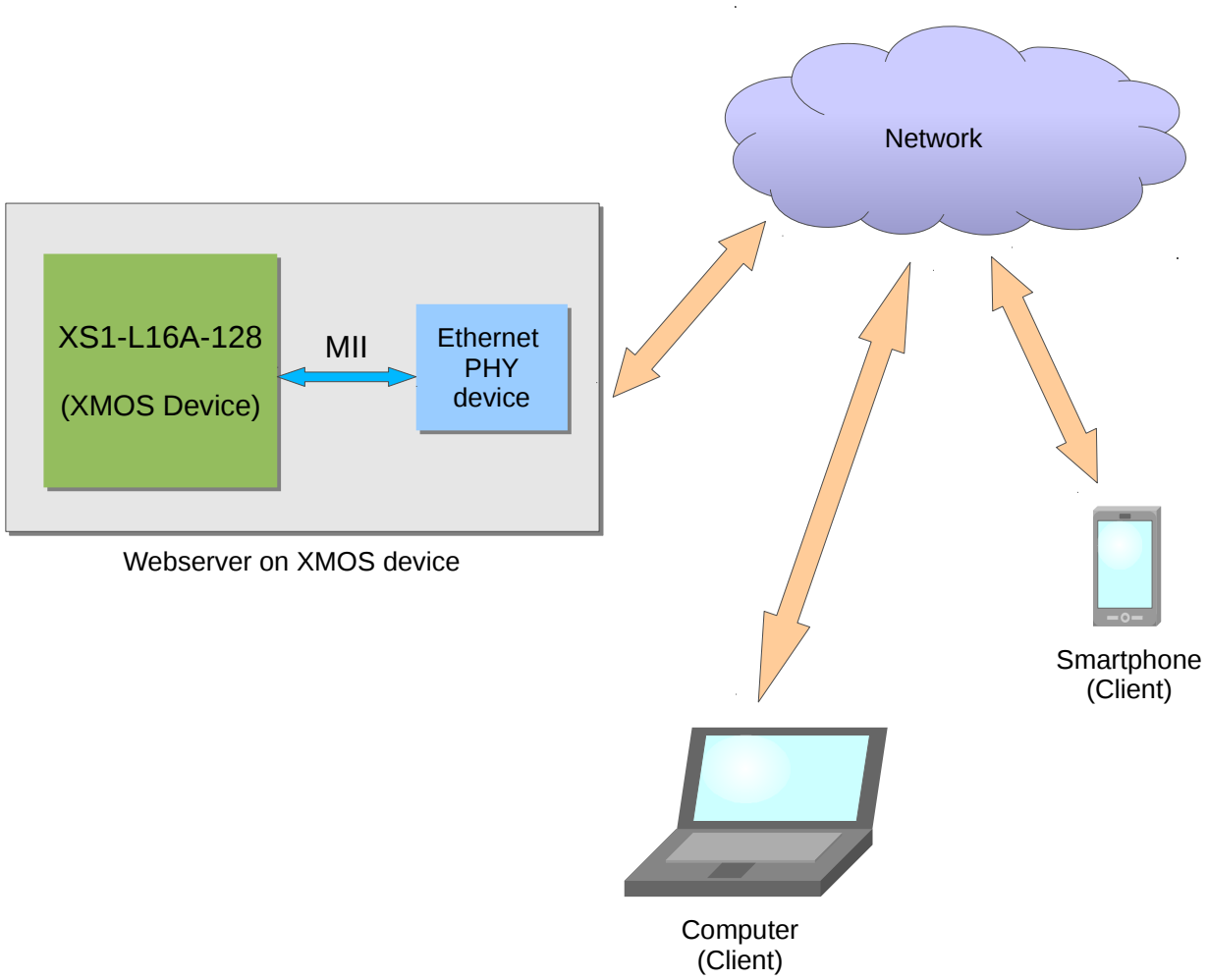

Figure 1: Block diagram of embedded webserver example

2 XMOS HTTP stack application note

The demo in this note uses the XMOS embedded webserver library and shows how to implement a simple embedded webserver that serves HTML files. In this application note, you will learn

- to setup and run an HTTP server
- to handle GET and POST requests
- to create dynamic content in web pages
- to store the web pages as a file tree in program memory

In this demo application, the server listens on TCP port 80 (default port for HTTP) for incoming HTTP requests. When a GET request for an URL is received, the server parses the URL, locates the requested HTML file and responds promptly with status code and content. The HTML files are stored in program memory using a simple file system.

To start using the embedded webserver library, you need to add `lib_webserver` to your "Makefile':

```
USED_MODULES = lib_webserver
```

You can access the Webserver functions in your source code via the `web_server.h` eader file:

```
#include "web_server.h"
```

2.1 Allocating hardware resources

The TCP/IP stack connects to the MII task from the Ethernet library which requires several ports to communicate with the Ethernet PHY. These ports are declared in the main program file (`main.xc`). In this demo the ports are set up for the Ethernet slice connected to the CIRCLE slot of the sliceKIT:

```
// Here are the port definitions required by ethernet. This port assignment
// is for the L16 sliceKIT with the ethernet slice plugged into the
// CIRCLE slot.
port p_eth_rxclk  = on tile[1]: XS1_PORT_1J;
port p_eth_rxd    = on tile[1]: XS1_PORT_4E;
port p_eth_txd    = on tile[1]: XS1_PORT_4F;
port p_eth_rxdv   = on tile[1]: XS1_PORT_1K;
port p_eth_txen   = on tile[1]: XS1_PORT_1L;
port p_eth_txclk  = on tile[1]: XS1_PORT_1I;
port p_eth_int    = on tile[1]: XS1_PORT_1O;
port p_eth_rxerr  = on tile[1]: XS1_PORT_1P;
port p_eth_timing = on tile[1]: XS1_PORT_8C;

clock eth_rxclk   = on tile[1]: XS1_CLKBLK_1;
clock eth_txclk   = on tile[1]: XS1_CLKBLK_2;
```

Note that the port p_eth_dummy does not need to be connected to external hardware - it is just used internally by the Ethernet library.

The MDIO Serial Management Interface (SMI) is used to transfer management information between MAC and PHY. This interface consists of two signals which are connected to two ports:

```
port p_smi_mdio = on tile[1]: XS1_PORT_1M;
port p_smi_mdc  = on tile[1]: XS1_PORT_1N;
```

The final ports used in the application are the ones to access the internal OTP memory on the xCORE. These ports are fixed and can be intialized with the OTP_PORTS_INITIALIZER macro supplied by the `lib_otpinfo` OTP reading library.

```
// These ports are for accessing the OTP memory
otp_ports_t otp_ports = on tile[1]: OTP_PORTS_INITIALIZER;
```

2.1.1 IP address

IP address configuration is for dedicating a unique IP address to our board. It can be either static or dynamic. In case of dynamic addressing, DHCP is used to obtain an IP address from the router. The following is the code that declares them.

```
/* IP configuration.
 * Change this to suit your network.
 * Leave all as 0 values to use DHCP */
xtcp_ipconfig_t ipconfig = {
  { 0, 0, 0, 0 }, // IP address (eg 192,168,0,2)
  { 0, 0, 0, 0 }, // Netmask (eg 255,255,255,0)
  { 0, 0, 0, 0 }  // Gateway (eg 192,168,0,1)
};
```

Use zero for IP address, netmask and gateway to obtain dynamic address using DHCP otherwise the static address values can be directly provided.

Note: Adding -DXTCP_VERBOSE_DEBUG=1 to the XCC_FLAGS in the `Makefile` will enable printing of dynamically allocated IP address in the debug console of xTIMEcomposer.

2.2 The application main() function

For the embedded webserver example, the system comprises four tasks running on three separate logical cores of an xCORE multicore microcontroller.

The tasks perform the following operations.

- The MII task which handles MII/Ethernet traffic.
- The SMI task which drives the Ethernet PHY. This does not run on a logical core on its own.
- The ethernet TCP/IP server.
- A task implementing the HTTP stack and web page handler.

These tasks communicate via the use of xC channels and interface connections which allow data to be passed between application code running on separate logical cores.

The following diagram shows the tasks and communication structure for this Embedded HTTP Webserver example.

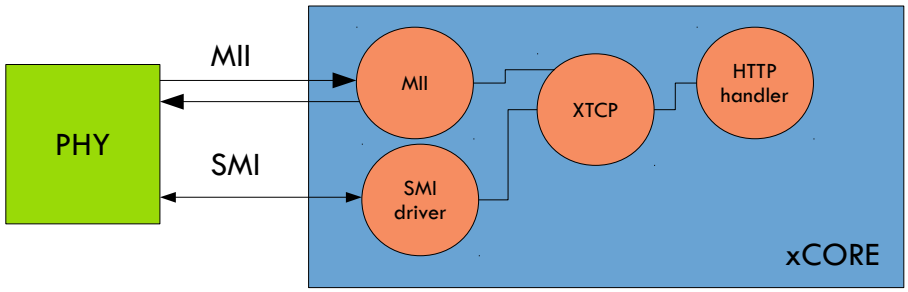

Figure 2: Task diagram of Embedded HTTP Webserver example

Below is the source code for the main function of this application, which is taken from the source file main.xc

```
int main(void) {
  chan c_xtcp[1];
  mii_if i_mii;
  smi_if i_smi;
  par {
    // MII/ethernet driver
    on tile[1]: mii(i_mii, p_eth_rxclk, p_eth_rxerr, p_eth_rxd, p_eth_rxdv,
                    p_eth_txclk, p_eth_txen, p_eth_txd, p_eth_timing,
                    eth_rxclk, eth_txclk, XTCP_MII_BUFSIZE)

    // SMI/ethernet phy driver
    on tile[1]: smi(i_smi, p_smi_mdio, p_smi_mdc);

    // TCP component
    on tile[1]: xtcp(c_xtcp, 1, i_mii,
                     null, null, null,
                     i_smi, ETHERNET_SMI_PHY_ADDRESS,
                     null, otp_ports, ipconfig);
    /* This function runs in a separate core and handles the TCP events
     * i.e the HTTP connections from the above TCP server task
     * through the channel 'c_xtcp[0]'
     */
    on tile[0]: http_handler(c_xtcp[0]);

  }
  return 0;
}
```

Looking at this in more detail you can see the following:

- The par statement describes running four separate tasks in parallel
- There is a function call to execute the ethernet XTCP server: xtcp()
- The channels and interfaces to connect the tasks together are set up at the beginning of main()
- The xtcp() task is connected to the the smi and mii tasks. This is so it can drive the Ethernet PHY.
- The IP configuration is passed to the function xtcp(). It also takes the OTP ports so it can configure the MAC address based on reading the OTP memory of the device.
- There is a function call to handle HTTP connections and other TCP events: http_handler()

2.3 Handling HTTP connections

`http_handler()` is the application function that initializes the Webserver and starts handling events from the TCP server task in an infinite loop. Each HTTP connection is handled through several events from the TCP server task.

Below is the source code of http_handler function from `main.xc`

```
/* Function to handle the HTTP connections (TCP events)
 * from the TCP server task through 'c_xtcp' channels */
void http_handler(chanend c_xtcp) {

  xtcp_connection_t conn;  /* TCP connection information */

  /* Initialize webserver */
  web_server_init(c_xtcp, null, null);
  /* Initialize web application state */
  init_web_state();
  init_gpio();

  while (1) {
    select
      {
      case xtcp_event(c_xtcp,conn):
        /* Handles HTTP connections and other TCP events */
        web_server_handle_event(c_xtcp, null, null, conn);
        break;
      }
  }
}
```

You can observe the following from the source code segment:

- There is an xtcp_connection_t type variable 'conn' to hold information on TCP connection and its event.
- A function call to initialize the Webserver: *web_server_init()*. This function:
  - initializes HTTP connection state variables
  - requests the TCP server task to listen on port 80 and report any connection events through *c_xtcp* channel.
- A function call to initialize application state: *init_web_state()*. In the Webserver example, the application state has states of LEDs and buttons connected to the xCORE device and a web page visit counter that counts how many times the web page is requested.
- An indefinite while loop to handle TCP events. The function *xtcp_event()* in the select case receives TCP events and instantiates connection parameter *conn*.
- *web_server_handle_event()* is the core function that actually handles the HTTP connections. It mainly parses the HTTP methods like GET and POST to obtain URL (Uniform Resource Locator), HTTP headers and the parameters passed with the requests. It then identifies the requested web resource, based on the URL, and responds with rendered web content.

Typically, when a client requests a URL with a GET method, the Webserver tries to find which HTML page maps with the requested URL and sends that HTML content as a response to the request.

Note: Currently the XMOS embedded webserver library supports only GET and POST HTTP methods.

The following sections discusses how the web pages are organized and served in the example application.

2.4 Organizing web content

The embedded webserver example application is supplied with a folder called 'web' with HTML files and images. This 'web' directory is the root directory from which files are served by the webserver. The URL '/' is mapped to this 'web' folder; By default the index.html is served when a GET request is received for the '/' URL.

The web tree provided with the example is shown below

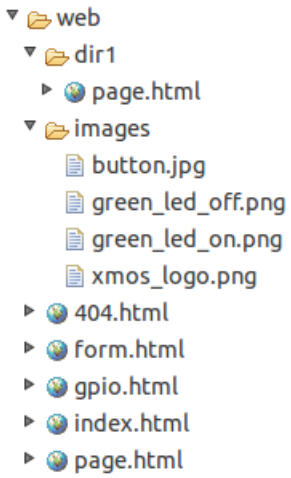

Figure 3: Web directory of the embedded webserver example

With the HTML pages in the web directory, you can include dynamic content via {% ... %} tags. The following segment of index.html file shows how the timer and page visit counter values are dynamically inserted into the page.

```
<p>You can also include dynamic content like this timestamp from an <b>XC
  timer: {% get_timer_value(buf) %}</b>
</p>

<p>
  The functions called during rendering can also include state. For
  example, here is a page load counter for this page: <b>{% get_counter_value(app_state,buf) %}</b>
</p>
```

You can notice from the above HTML text that there are functions *get_timer_value()* and *get_counter_value()* wrapped with {% ... %} tags. These functions are evaluated during page rendering to insert dynamic text content. You can find the declaration of these functions in *web_server_conf.h* and their definitions in *web_page_functions.c*.

These tagged functions present in web pages can also access the parameters that are passed with GET or POST requests. HTTP GET parameter denotes the query string sent in the URL and POST parameter denotes the form data submitted with requests. In the example, a POST parameter received as HTML form submission is displayed in the *form.html* file.

The following is the form.html file which calls a function *get_input_param()* to read the POST data.

```
<html>
<body>
<h1>Form Response</h1>

<p>You entered the text: {% get_input_param(connection_state, buf) %}</o>

</body>
</html>
```

The function *get_input_param()* is defined in *web_page_functions.c* and is shown below.

```
/* Function to get the parameter passed with POST request */
int get_input_param(int connection_state, char buf[]) {
  return web_server_copy_param("input",  connection_state, buf);
}
```

In the above snippet the *web_server_copy_param()* is the library API which returns the required parameter in *buf* array. The parameter name is passed as the first argument to this function and it is the HTML form field name 'input' in our example.

In addition to creating dynamic content, the tagged functions help to control and monitor the embedded system effectively. You can find *gpio.html* in the Webserver example that enables you to toggle LEDs and monitor push buttons connected to the xCORE device.

2.5 Memory for Web content

The embedded webserver library provides a framework to handle storage of web pages either in program memory or in external SPI flash memory. When the Webserver example application is built, the 'web' directory is packaged and added to the executable file (.xe file). This way the web content gets into program memory as file system. The packaging of the 'web' directory is achieved with a script file *makefs.py* present in the embedded webserver library.

For storing the web content in external SPI flash memory device you can refer to the webserver programming guide.

https://www.xmos.com/published/embedded-webserver-library-programming-guide

APPENDIX A - Demo Hardware setup

To setup the demo hardware the following boards are required.

- XP-SKC-L2 sliceKIT L2 core board
- XA-SK-E100 Ethernet sliceCARD
- XA-SK-GPIO GPIO sliceCARD
- xTAG-2 and XA-SK-XTAG2 adapter

Follow the steps below to setup the hardware:

- Connect the xTAG-2 to the sliceKIT L2 core board using xTAG2 adapter.
- Use a USB cable to connect the xTAG-2 to your computer.
- Connect the Ethernet sliceCARD to the sliceKIT L2 core board's CIRCLE slot (indicated by a white color circle / J6 slot).
- Use an Ethernet cable to connect the sliceCARD to your computer's Ethernet port or to a spare Ethernet port of the router.
- Connect the GPIO sliceCARD to the sliceKIT L2 core board's TRIANGLE slot (indicated by a white color triangle / J3 slot).
- Provide 12V power supply to the sliceKIT board.

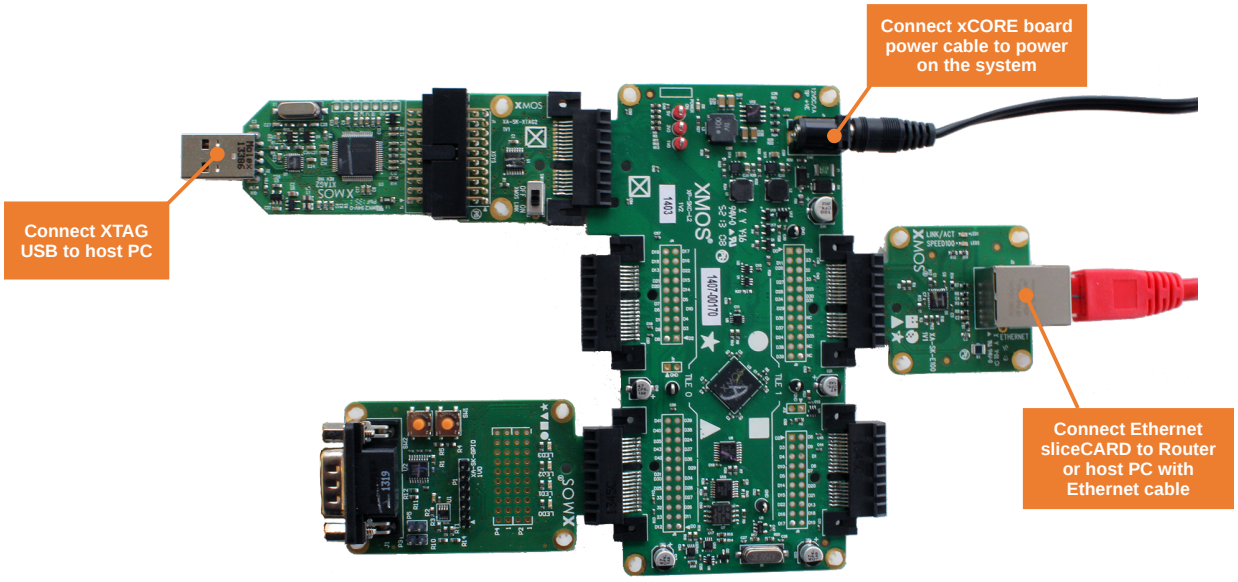

Figure 4: XMOS hardware setup for the Webserver example

APPENDIX B - Launching the Webserver

Once the demo example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer studio we can execute the application on the sliceKIT core board.

Once built there will be a `bin` directory within the project which contains the binary for the xCORE device. The xCORE binary has a XMOS standard .xe extension.

B.1 Launching from the command line

From the command line we use the `xrun` tool to download code to both the xCORE devices. If we change into the bin directory of the project we can execute the code on the xCORE microcontroller as follows:

```
> xrun --xscope app_webserver_demo.xe          <-- Download and execute the xCORE code
```

Once this command has executed the webserver will run and you can see the following text in the command line window:

```
Address: 0.0.0.0
Gateway: 0.0.0.0
Netmask: 0.0.0.0
dhcp: 172.17.0.115
```

The dhcp value is the dynamically allocated IP address for this XMOS demo hardware. This IP address can be different each time you launch the application.

B.2 Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio we use the run mechanism to download code to xCORE device. Select the xCORE binary from the bin directory, right click and then follow the instructions below.

- Select **Run Configuration**.
- Set **Target** to XMOS XTAG-2 hardware
- Enable **xSCOPE** printing in the run configuration
- Click **Apply** and then **Run**.

Once the the application is set to run you will see the following text in the xTIMEcomposer console window:

```
Address: 0.0.0.0
Gateway: 0.0.0.0
Netmask: 0.0.0.0
dhcp: 172.17.0.115
```

The dhcp value is the dynamically allocated IP address for this XMOS demo hardware. This IP address can be different each time you launch the application.

APPENDIX C - Run the demo

Now you can run the demo to see the web pages hosted by your Webserver.

- Open web browser in your development PC. You can use Internet explorer, Chrome, Mozilla firefox, Safari or any standard web browser.
- Type the IP address of demo hardware in the web browser's address bar to see the web page.
- Refresh/Reload the web page to observe updates to the XC timer and page visit counter values.
- Provide a URL that doesn't exist in the example's web tree to see 404.html popping up.
- Click on the GPIO control page hyperlink to open gpio.html. You can toggle the LEDs and monitor push button states using this page.

The picture below shows the index web page:

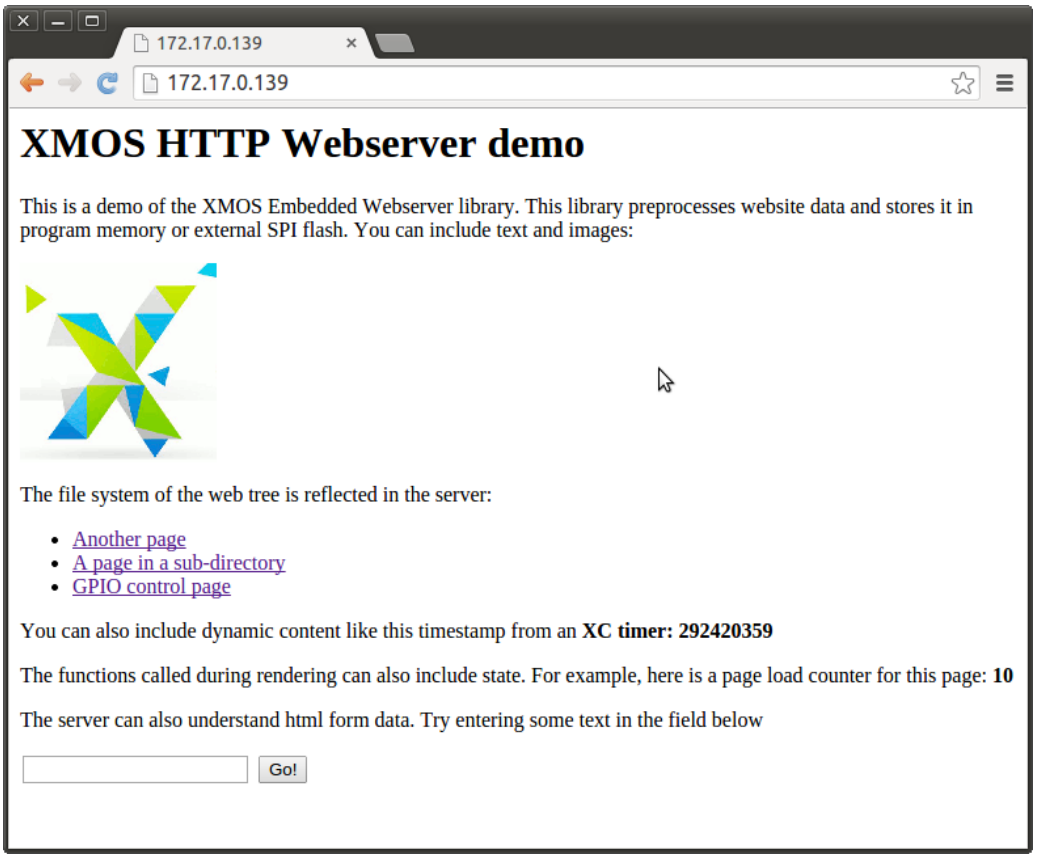

Figure 5: Index.html page of the embedded webserver example

The picture below is the GPIO page where you can control the LEDs and view button states:

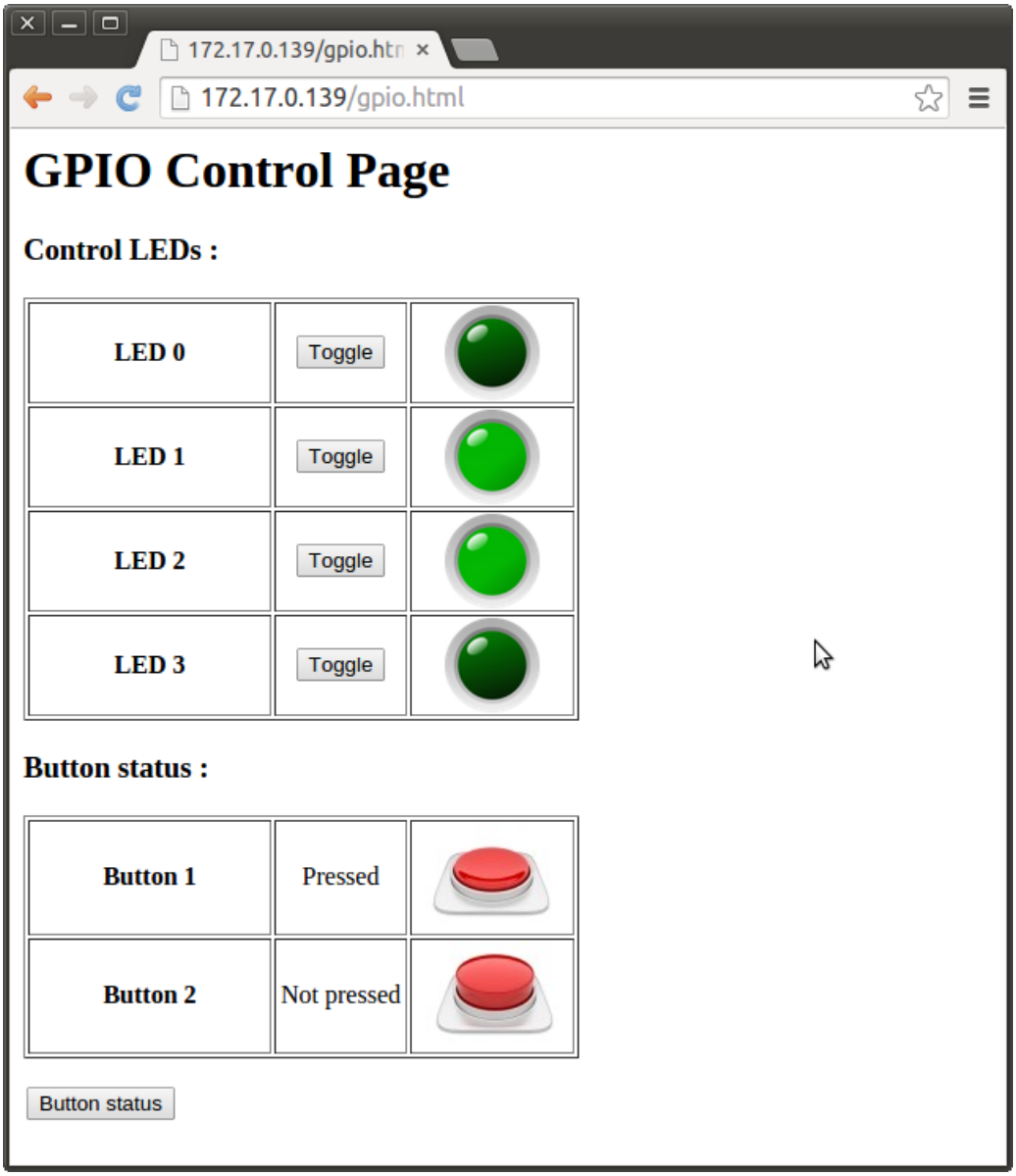

Figure 6: gpio.html page of the embedded webserver example

APPENDIX D - References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

XMOS Embedded webserver Programming Guide

https://www.xmos.com/published/embedded-webserver-library-programming-guide

HTTP/1.1 Specification

https://www.ietf.org/rfc/rfc2616.txt

Quick introduciton to HTTP methods

http://www.w3schools.com/tags/ref_httpmethods.asp

sliceKIT Hardware Manual

https://www.xmos.com/support/xkits?subcategory=sliceKIT&product=15826&component=16091

Ethernet Slice Card

https://www.xmos.com/products/xkits/slicekit#ethernet-slice

GPIO Slice card

https://www.xmos.com/products/xkits/slicekit#gpio-slice

APPENDIX E - Source code listing

E.1 Source code for main.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <platform.h>
#include <xs1.h>
#include <print.h>
#include <xtcp.h>
#include <web_server.h>
#include <stdio.h>
#include <string.h>


// Here are the port definitions required by ethernet. This port assignment
// is for the L16 sliceKIT with the ethernet slice plugged into the
// CIRCLE slot.
port p_eth_rxclk  = on tile[1]: XS1_PORT_1J;
port p_eth_rxd    = on tile[1]: XS1_PORT_4E;
port p_eth_txd    = on tile[1]: XS1_PORT_4F;
port p_eth_rxdv   = on tile[1]: XS1_PORT_1K;
port p_eth_txen   = on tile[1]: XS1_PORT_1L;
port p_eth_txclk  = on tile[1]: XS1_PORT_1I;
port p_eth_int    = on tile[1]: XS1_PORT_1O;
port p_eth_rxerr  = on tile[1]: XS1_PORT_1P;
port p_eth_timing = on tile[1]: XS1_PORT_8C;

clock eth_rxclk   = on tile[1]: XS1_CLKBLK_1;
clock eth_txclk   = on tile[1]: XS1_CLKBLK_2;

port p_smi_mdio = on tile[1]: XS1_PORT_1M;
port p_smi_mdc  = on tile[1]: XS1_PORT_1N;

// These ports are for accessing the OTP memory
otp_ports_t otp_ports = on tile[1]: OTP_PORTS_INITIALIZER;

/* GPIO slice on triangle slot
 * PORT_4E connected to the 4 LEDs and PORT_8D connected to 2 buttons */
on tile[0]: port p_led=XS1_PORT_4E;
on tile[0]: port p_button=XS1_PORT_8D;


/* IP configuration.
 * Change this to suit your network.
 * Leave all as 0 values to use DHCP */
xtcp_ipconfig_t ipconfig = {
  { 0, 0, 0, 0 }, // IP address (eg 192,168,0,2)
  { 0, 0, 0, 0 }, // Netmask (eg 255,255,255,0)
  { 0, 0, 0, 0 }  // Gateway (eg 192,168,0,1)
};

/* Function to get 32-bit timer value as a string */
int get_timer_value(char buf[])
{
  /* Declare a timer resource */
  timer tmr;
  unsigned time;
```

```
  int len;
  /* Read the timer value in a variable */
  tmr :> time;
  /* Convert the timer value to string */
  sprintf(buf, "%u", time);
  return len;
}

/* Function to initialize the GPIO */
void init_gpio(void)
{
  /* Set all LEDs to OFF (Active low)*/
  p_led <: 0x0F;
}

/* Function to set LED state - ON/OFF */
void set_led_state(int led_id, int val)
{
  int value;
  /* Read port value into a variable */
  p_led :> value;
  if (!val) {
      p_led <: (value | (1 << led_id));
  } else {
      p_led <: (value & ~(1 << led_id));
  }
}

/* Function to read current button state */
int get_button_state(int button_id)
{
  int value;
  p_button :> value;
  value &= (1 << button_id);
  return (value >> button_id);
}

/* Function to handle the HTTP connections (TCP events)
 * from the TCP server task through 'c_xtcp' channels */
void http_handler(chanend c_xtcp) {

  xtcp_connection_t conn;  /* TCP connection information */

  /* Initialize webserver */
  web_server_init(c_xtcp, null, null);
  /* Initialize web application state */
  init_web_state();
  init_gpio();

  while (1) {
    select
      {
      case xtcp_event(c_xtcp,conn):
        /* Handles HTTP connections and other TCP events */
        web_server_handle_event(c_xtcp, null, null, conn);
        break;
      }
  }
```

```
}

#define XTCP_MII_BUFSIZE (4096)
#define ETHERNET_SMI_PHY_ADDRESS (0)


/* The main starts four tasks (functions) in three different logical cores. */
int main(void) {
  chan c_xtcp[1];
  mii_if i_mii;
  smi_if i_smi;
  par {
    // MII/ethernet driver
    on tile[1]: mii(i_mii, p_eth_rxclk, p_eth_rxerr, p_eth_rxd, p_eth_rxdv,
                    p_eth_txclk, p_eth_txen, p_eth_txd, p_eth_timing,
                    eth_rxclk, eth_txclk, XTCP_MII_BUFSIZE)

    // SMI/ethernet phy driver
    on tile[1]: smi(i_smi, p_smi_mdio, p_smi_mdc);

    // TCP component
    on tile[1]: xtcp(c_xtcp, 1, i_mii,
                     null, null, null,
                     i_smi, ETHERNET_SMI_PHY_ADDRESS,
                     null, otp_ports, ipconfig);
    /* This function runs in a separate core and handles the TCP events
     * i.e the HTTP connections from the above TCP server task
     * through the channel 'c_xtcp[0]'
     */
    on tile[0]: http_handler(c_xtcp[0]);

  }
  return 0;
}
```