# Ethernet Slice Simple Webserver Application Quickstart

IN THIS DOCUMENT

- Hardware setup
- Import and build the application
- Run the application
- Troubleshooting
- Next steps

This simple demonstration of xTIMEcomposer Studio functionality uses the sliceKIT Ethernet Slice together with the xSOFTip TCP/IP Ethernet component to:

- Run a TCP/IP stack on the xCORE
- Run a very simple HTTP server to display a “hello world” webpage

## 1 Hardware setup

The sliceKIT Core Board has four slots with edge conectors: `SQUARE`, `CIRCLE`, `TRIANGLE` and `STAR`.

To setup up the system:

1. Connect the Ethernet Slice Card to the sliceKIT Core Board using the connector marked with the `CIRCLE`.
2. Connect the xTAG Adapter to sliceKIT Core Board and connect XTAG-2 to the adapter.
3. Connect the xTAG-2 to host PC. Note that a USB cable is not provided with the sliceKIT starter kit.
4. Plug the power supply into the sliceKIT Core Board and turn the power supply on.

## 2 Import and build the application

1. On your PC, open xTIMEcomposer. You need to be in the edit perspective (Window->Open Perspective->XMOS Edit). If the XMOS Edit option isn’t in the list, you are already in the edit perspective.
2. Locate the `'Simple HTTP Demo'` item in the xSOFTip pane on the bottom left of the window and drag it into the Project Explorer window in the xTIMEcomposer.

Publication Date: 2014/6/12 Document Number: XM002033A

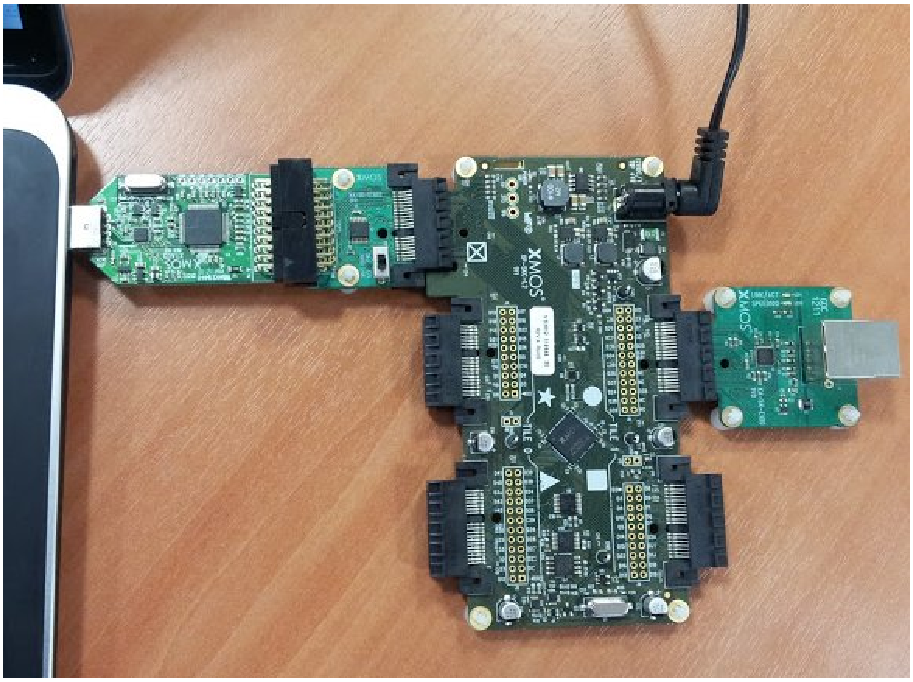

**Figure 1:** Hardware Setup for Simple HTTP Demo

This will also cause the modules on which this application depends to be imported as well.

3. Click on the app_simple_webserver item in the Explorer pane then click on the build icon (hammer) in xTIMEcomposer. Check the console window to verify that the application has built successfully.

For help in using xTIMEcomposer, try the xTIMEcomposer tutorials, which you can find by selecting Help->Tutorials from the xTIMEcomposer menu.

Note that the Developer Column in the xTIMEcomposer on the right hand side of your screen provides information on the xSOFTip components you are using. Select the `module_xtcp` component in the Project Explorer, and you will see its description together with API documentation (reached by double clicking on the `Documentation` item within the project). If you view other documentation, you can get back to this quickstart guide by cliking the *back* icon in the Developer Column until you return to this quickstart guide.

## 3 Run the application

Now that the application has been compiled, the next step is to run it on the sliceKIT Core Board using the tools to load the application over JTAG (via the XTAG-2 and XTAG Adaptor Card) into the xCORE multicore microcontroller.

1. Click on the `Run` icon (the white arrow in the green circle). The debug console window in xTIMEcomposer should then display the message:

   ```
   **WELCOME TO THE SIMPLE WEBSERVER DEMO**
   ```

   This has been generated from the application code via a call to the `printstr()` function.

2. Connect the sliceKIT Ethernet Slice your network via an ethernet cable. The application is configured by default to use DHCP to obtain an IP address. If you wish to change this to a static IP address edit the `main.xc` file in the application (you can reach this via the Project Explorer in the xTIMEcomposer) and change the `ipconfig` data structure.

3. The console window should display a message like:

   ```
   IP Address: 192.168.0.4
   ```

   the actual IP address will depend on your local network.

4. From a PC connected to the same network, open a web browser and open the link:

   ```
   http://192.168.0.4
   ```

   using the IP adress printed out by the application. This should display a “hello world” webpage. You have now got the ethernet slice up and running.

## 4 Troubleshooting

If the demo does not work try the following:

- Ensure the Ethernet Slice Card is connected to the CIRCLE connector of the core board.
- Ensure the slice network cable is fully connected. There are activity LEDs next to the ethernet connector that should illuminate if connected.
- Ensure that both the PC and Ethernet Slice card are connected to the same network and can route to each other. If you are using a dynamically allocated address, make sure your DHCP server is configured correctly. If using a static address, make sure your PC is configured to talk to that address (in Windows you need to check your Network Adapter TCP/IP settings).

## 5 Next steps

### 5.1 Look at the Code

1. Examine the application code. In xTIMEcomposer navigate to the `src` directory under `app_simple_webserver` and double click on the main.xc file within it. The file will open in the central editor window.

2. Find the main function and note that it runs the ethernet_xtcp_server() (which runs the ethernet driver and tcp stack) and the xhttpd() function in parallel.

3. Look at the xhttpd.xc and httpd.xc files. These implement the webserver logic that connects to the TCP/IP stack. In particular the httpd_handle_event function that responds to a TCP event over an xC channel and performs the functions of the webserver. See the TCP/IP programming guide for details on how the xtcp stack works.

## 5.2 Look at Other Examples

For a more complex embedded wbeserver demo that allows user interaction try the Slicekit GPIO and Ethernet Combo Demo demo application which can be found under sliceKIT->Demos categroy in the xSOFTip Explorer pane within xTIMEcomposer.

## 5.3 Go through this code in detail

This last section is optional and walks through the main application code in detail.

The toplevel main of this application sets up the different components running on different logical cores on the device. It can be found in the file main.xc.

First the TCP/IP server is run on the tile given by the define ETHERNET_DEFAULT_TILE (supplied by the ethernet_board_support.h header which gives defines for common XMOS development boards.). It is run via the function ethernet_xtcp_server(). The server runs both the ethernet code to communicate with the ethernet phy and the tcp server on two logical cores.

```
on ETHERNET_DEFAULT_TILE:
   ethernet_xtcp_server(xtcp_ports,
                        ipconfig,
                        c_xtcp,
                        1);
```

The client to the TCP/IP server is run as a separate task and connected to the TCP/IP server via the first element c_xtcp channel array. The function xhttpd implements the web server.

```
on tile[0]: xhttpd(c_xtcp[0]);
```

### 5.3.1 The webserver mainloop

The webserver is implemented in the xhttpd function in xhttpd.xc. This function implements a simple loop that just responds to events from the TCP/IP server. When an event occurs it is passed onto the httpd_handle_event handler.

```
void xhttpd(chanend tcp_svr)
{
  xtcp_connection_t conn;
  printstrln("**WELCOME TO THE SIMPLE WEBSERVER DEMO**");
  // Initiate the HTTP state
  httpd_init(tcp_svr);

  // Loop forever processing TCP events
  while(1)
    {
      select
        {
        case xtcp_event(tcp_svr, conn):
          httpd_handle_event(tcp_svr, conn);
          break;
        }

    }
}
```

**The webserver event handler** The event handler is implemented in httpd.c and contains the main logic of the web server. The server can handle several connections at once. However, events for each connection may be interleaved so the handler needs to store separate state for each one. The httpd_state_t structures holds this state:

```
typedef struct httpd_state_t {
  int active;      //< Whether this state structure is being used
                   //  for a connection
  int conn_id;     //< The connection id
  char *dptr;      //< Pointer to the remaining data to send
  int dlen;        //< The length of remaining data to send
  char *prev_dptr; //< Pointer to the previously sent item of data
} httpd_state_t;

httpd_state_t connection_states[NUM_HTTPD_CONNECTIONS];
```

The http_init function is called at the start of the application. It initializes the connection state array and makes a request to accept incoming new TCP connections on port 80 (using the xtcp_listen() function):

```
void httpd_init(chanend tcp_svr)
{
  int i;
  // Listen on the http port
  xtcp_listen(tcp_svr, 80, XTCP_PROTOCOL_TCP);

  for ( i = 0; i < NUM_HTTPD_CONNECTIONS; i++ )
    {
      connection_states[i].active = 0;
      connection_states[i].dptr = NULL;
    }
}
```

When an event occurs the `httpd_handle_event` function is called. The behaviour of this function depends on the event type. Firstly, link status events are ignored:

```
void httpd_handle_event(chanend tcp_svr, xtcp_connection_t *conn)
{
  // We have received an event from the TCP stack, so respond
  // appropriately

  // Ignore events that are not directly relevant to http
  switch (conn->event)
    {
    case XTCP_IFUP: {
      xtcp_ipconfig_t ipconfig;
      xtcp_get_ipconfig(tcp_svr, &ipconfig);

#if IPV6
      unsigned short a;
      unsigned int i;
      int f;
      xtcp_ipaddr_t *addr = &ipconfig.ipaddr;
      printstr("IPV6 Address = [");
      for(i = 0, f = 0; i < sizeof(xtcp_ipaddr_t); i += 2) {
        a = (addr->u8[i] << 8) + addr->u8[i + 1];
        if(a == 0 && f >= 0) {
          if(f++ == 0) {
            printstr("::");
           }
        } else {
            if(f > 0) {
              f = -1;
            } else if(i > 0) {
                printstr(":");
            }
          printhex(a);
        }
      }
      printstrln("]");
#else
      printstr("IP Address: ");
      printint(ipconfig.ipaddr[0]);printstr(".");
      printint(ipconfig.ipaddr[1]);printstr(".");
      printint(ipconfig.ipaddr[2]);printstr(".");
      printint(ipconfig.ipaddr[3]);printstr("\n");
#endif
      }
      return;
    case XTCP_IFDOWN:
    case XTCP_ALREADY_HANDLED:
      return;
    default:
      break;
    }
```

For other events, we first check that the connection is definitely an http connection (is directed at port 80) and then call one of several event handlers for each type of event. The is a separate function for new connections, receiving data, sending data and closing connections:

```
if (conn->local_port == 80) {
  switch (conn->event)
    {
    case XTCP_NEW_CONNECTION:
      httpd_init_state(tcp_svr, conn);
      break;
    case XTCP_RECV_DATA:
      httpd_recv(tcp_svr, conn);
      break;
    case XTCP_SENT_DATA:
    case XTCP_REQUEST_DATA:
    case XTCP_RESEND_DATA:
        httpd_send(tcp_svr, conn);
        break;
    case XTCP_TIMED_OUT:
    case XTCP_ABORTED:
    case XTCP_CLOSED:
        httpd_free_state(conn);
        break;
    default:
      // Ignore anything else
      break;
    }
  conn->event = XTCP_ALREADY_HANDLED;
}
```

The following sections describe the four handler functions.

**Handling Connections** When a `XTCP_NEW_CONNECTION` event occurs we need to associate some state with the connection. So the `connection_states` array is searched for a free state structure.

```
void httpd_init_state(chanend tcp_svr, xtcp_connection_t *conn)
{
  int i;

  // Try and find an empty connection slot
  for (i=0;i<NUM_HTTPD_CONNECTIONS;i++)
    {
      if (!connection_states[i].active)
        break;
    }
```

If we don’t find a free state we cannot handle the connection so `xtcp_abort()`‘ is called to abort the connection.

```
if ( i == NUM_HTTPD_CONNECTIONS )
  {
    xtcp_abort(tcp_svr, conn);
  }
```

If we can allocate the state structure then the elements of the structure are initialized. The function xtcp_set_connection_appstate() is then called to associate the state with the connection. This means when a subsequent event is signalled on this connection the state can be recovered.

```
else
  {
    connection_states[i].active = 1;
    connection_states[i].conn_id = conn->id;
    connection_states[i].dptr = NULL;
    xtcp_set_connection_appstate(
         tcp_svr,
         conn,
         (xtcp_appstate_t) &connection_states[i]);
```

When a XTCP_TIMED_OUT, XTCP_ABORTED or XTCP_CLOSED event is received then the state associated with the connection can be freed up. This is done in the httpd_free_state function:

```
void httpd_free_state(xtcp_connection_t *conn)
{
  int i;

  for ( i = 0; i < NUM_HTTPD_CONNECTIONS; i++ )
    {
      if (connection_states[i].conn_id == conn->id)
        {
          connection_states[i].active = 0;
        }
    }
}
```

**Receiving Data** When an XTCP_RECV_DATA event occurs the httpd_recv function is called. The first thing this function does is call the xtcp_recv() function to place the received data in the data array. (Note that all TCP/IP clients *must* call xtcp_recv() directly after receiving this kind of event).

```
void httpd_recv(chanend tcp_svr, xtcp_connection_t *conn)
{
  struct httpd_state_t *hs = (struct httpd_state_t *) conn->appstate;
  char data[XTCP_CLIENT_BUF_SIZE];
  int len;

  // Receive the data from the TCP stack
  len = xtcp_recv(tcp_svr, data);
```

The hs variable points to the connection state. This was recovered from the appstate member of the connection structure which was previously associated with application state when the connection was set up. As a safety check we only proceed if this state has been set up and the hs variable is non-null.

```
if ( hs == NULL || hs->dptr != NULL)
  {
    return;
  }
```

Now the connection state is known and the incoming data buffer filled. To keep things simple, this server makes the assumption that a single tcp packet gives us enough information to parse the http request. However, many applications will need to concatenate each tcp packet to a different buffer and handle data after several tcp packets have come in. The next step in the code is to call the parse_http_request function:

```
parse_http_request(hs, &data[0], len);
```

This function examines the incoming packet and checks if it is a GET request. If so, then it always serves the same page. We signal that a page is ready to the callee by setting the data pointer (dptr) and data length (dlen) members of the connection state.

```
void parse_http_request(httpd_state_t *hs, char *data, int len)
{
  // Return if we have data already
  if (hs->dptr != NULL)
    {
      return;
    }

  // Test if we received a HTTP GET request
  if (strncmp(data, "GET ", 4) == 0)
    {
      // Assign the default page character array as the data to send
      hs->dptr = &page[0];
      hs->dlen = strlen(&page[0]);
    }
  else
    {
      // We did not receive a get request, so do nothing
    }
}
```

The final part of the receive handler checks if the parse_http_request function set the dptr data pointer. If so, then it signals to the tcp/ip server that we wish to send some data on this connection. The actual sending of data is handled when an XTCP_REQUEST_DATA event is signalled by the tcp/ip server.

```
if (hs->dptr != NULL)
  {
    // Initate a send request with the TCP stack.
    // It will then reply with event XTCP_REQUEST_DATA
    // when it's ready to send
    xtcp_init_send(tcp_svr, conn);
  }
```

**Sending Data** To send data the connection state keeps track of three variables:

| Name | Description |
|---|---|
| `dptr` | A pointer to the next piece of data to send |
| `dlen` | The amount of data left to send |
| `prev_dptr` | The previous value of `dptr` before the last send |

We keep the previous value of `dptr` in case the tcp/ip server asks for a resend.

On receiving an `XTCP_REQUEST_DATA`, `XTCP_SENT_DATA` or `XTCP_RESEND_DATA` event the function `httpd_send` is called.

The first thing the function does is check whether we have been asked to resend data. In this case it sends the previous amount of data using the `prev_dptr` pointer.

```
if (conn->event == XTCP_RESEND_DATA) {
  xtcp_send(tcp_svr, hs->prev_dptr, (hs->dptr - hs->prev_dptr));
  return;
}
```

If the request is for the next piece of data, then the function first checks that we have data left to send. If not, the function `xtcp_complete_send()` is called to finish the send transaction and then the connection is closed down with `xtcp_close()` (since HTTP only does one transfer per connection).

```
if (hs->dlen == 0 || hs->dptr == NULL)
  {
    // Terminates the send process
    xtcp_complete_send(tcp_svr);
    // Close the connection
    xtcp_close(tcp_svr, conn);
  }
```

If we have data to send, then first the amount of data to send is calculated. This is based on the amount of data we have left (`hs->dlen`) and the maximum we can send (`conn->mss`). Having calculated this length, the data is sent using the `xtcp_send()` function.

Once the data is sent, all that is left to do is update the `dptr`, `dlen` and `prev_dptr` variables in the connection state.

```
else {
  int len = hs->dlen;

  if (len > conn->mss)
    len = conn->mss;

  xtcp_send(tcp_svr, hs->dptr, len);

  hs->prev_dptr = hs->dptr;
  hs->dptr += len;
  hs->dlen -= len;
}
```