Embedded Webserver Library

This software library allows you to generate a webserver that communicates using the XMOS TCP/IP server component.

Features

- Automatically package a file tree of web pages into data that be accessed on the device
- Store web pages in either program memory or on an attached SPI flash
- Call C/XC functions from within web page templates to access dynamic content
- Handle GET and POST HTTP requests
- Separate the handling of TCP traffic and the access of flash into different tasks passing data over XC channels. Allowing you to integrate the webserver in other applications that already handle TCP or access flash.

Typical Resource Usage

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
Default	0	0	0	~5.8K	1

Note that this does not include the TCP/IP stack (which is a separate library) or any web-pages stored in memory.

Software version and dependencies

This document pertains to version 2.0.1 of this library. It is known to work on version 14.1.1 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_xtcp (>=4.0.0)

Related application notes

The following application notes use this library:

- AN00122 - Using the XMOS embedded webserver library

1 Usage

To use the library you need to add `lib_webserver` to the `USED_MODULES` variable in your application Makefile.

Within your application you can also add the following files to configure the server:

web/ Directory containing the HTML and other files to be served by the webserver

web/webserver.conf A configuration file to control the generation of the web data

web_server_conf.h The file can be anywhere in your source tree and contains #defines for configuring the webserver code.

1.1 Choosing the webserver mode

1.1.1 Configuring the webserver to run from program memory

The module is set up to run from program memory by default so no extra configuration is required.

1.1.2 Configuring the webserver to run from flash

To run from flash you need to:

1. Set the define `WEB_SERVER_USE_FLASH` to 1 in your `web_server_conf.h` file
2. Add the following lines to `webserver.conf` in your web/ directory:

```
[Webserver]
use_flash=true
```

This will cause the web pages to be packed into a binary image and placed into the application binary directory. See §1.4 for details on how to write the data to SPI flash.

1.1.3 Configuring the webserver to run from flash with a separate flash task

To run from flash you need to:

1. Set the define `WEB_SERVER_USE_FLASH` to 1 in your `web_server_conf.h` file
2. Set the define `WEB_SERVER_SEPARATE_FLASH_TASK` to 1 in your `web_server_conf.h` file
3. Add the following lines to `webserver.conf` in your web/ directory:

```
[Webserver]
use_flash=true
```

This will cause the web pages to be packed into a binary image and placed into the application binary directory. See §1.4 for details on how to write the data to SPI flash.

1.2 Creating the web pages

To create your web pages just place them in the web/ sub-directory of your application. This can include text and images and be a file tree. For example, the demo application web tree is:

```
web/webserver.conf
web/index.html
web/form.html
web/dir1/page.html
web/dir2/page.html
web/images/xmos_logo.png
```

1.3 Writing dynamic content

With the HTML pages in the web directory. You can include dynamic content via {% ... %} tags. This will evaluate the C expression within the tag. This expression has several C variables available to it:

- char *buf - a buffer to fill with the content to be rendered in place of the tag
- int connection_state - an integer representing the connection state of the current HTTP connection. This can be used with the functions in §3.3.
- int app_state - an integer representing the application state. This integer can be set with the web_server_set_app_state().

The application must return the length of data it has placed in the buf variable.

All functions that are called by expressions in your web pages need to be prototyped in web_server_conf.h

For example when the following HTML is rendered:

```
<p>{%  my_function(buf) %}</p>
```

The web server will render up to the end of the first *<p>* tag and then call the my_function C/XC function passing in the buf array to be filled.

The function my_function must be declared in web_server_conf.h e.g.:

```
int my_function(char buf[]);
```

The implementation of the function needs to be somewhere in your source tree:

```
int my_function(char buf[]) {
  char msg[] = "Hello World!";
  strcpy(buf, msg);
  return sizeof(msg)-1;
}
```

Note that this example function returns the number of characters it has placed in the buffer (not including the terminating '0' character).

The web server will then insert these characters into the page. So the page returned to the client will be:

```
<p>Hello World!</p>
```

1.4 Writing the pages to SPI flash

If you configure the webserver to use flash then you need to place the data onto the attached SPI flash before running your program. To do this use the xflash command.

The build of your program will place the data in a file called web_data.bin in your binary directory. You can flash it using a command like:

```
xflash --boot-partition-size 0x10000 bin/myprog.xe --data bin/web_data.bin
```

See XM-000965-PC for more details on how to use xflash.

1.5 Integrating the webserver into your code

All functions in your code to call the webserver can be found in the web_server.h header:

```
#include <web_server.h>
```

1.5.1 Without SPI Flash

To use the webserver you must have an instance of an XTCP server running on your system. You can then write a function that implements a task that handles tcp events. This may do other functions/handle other tcp traffic as well as implementing the webserver. Here is a rough template of how the code should look:

```
void tcp_handler(chanend c_xtcp) {
  xtcp_connection_t conn;
  web_server_init(c_xtcp, null, null);

  // Initialize your other code here

  while (1) {
    select
      {
      case xtcp_event(c_xtcp,conn):

        // handle other kinds of tcp traffic here

        web_server_handle_event(c_xtcp, null, null, conn);
        break;

      // handle other events in your system here
      }
  }
}
```

1.5.2 Using SPI Flash

To use SPI flash you need to pass the ports to access the flash into the webserver functions. For example:

```
void tcp_handler(chanend c_xtcp, fl_SPIPorts &flash_ports) {
  xtcp_connection_t conn;
  web_server_init(c_xtcp, null, flash_ports);
  while (1) {
    select
      {
      case xtcp_event(c_xtcp, conn):
        web_server_handle_event(c_xtcp, null, flash_ports, conn);
        break;
      }
  }
}
```

See XM-000953-PC for details on the flash ports. You also need to define an array of flash devices and specify the WEB_SERVER_FLASH_DEVICES and WEB_SERVER_NUM_FLASH_DEVICES defines in web_server_conf.h to tell the web server which flash devices to expect.

1.5.3 Using SPI Flash in a separate task

If you configure the web server to use a separate task for flash you need to run two tasks. The TCP handling tasks now takes a chanend to talk to the other task. For example:

```
void tcp_handler(chanend c_xtcp, chanend c_flash) {
 xtcp_connection_t conn;
 web_server_init(c_xtcp, c_flash, null);
 while (1) {
   select
     {
     case xtcp_event(c_xtcp, conn):
       web_server_handle_event(c_xtcp, c_flash, null, conn);
       break;
     case web_server_flash_response(c_flash):
       web_server_flash_handler(c_flash, c_xtcp);
       break;
     }
 }
```

When serving a web page the web_server_handle_event() may request data over the c_flash channel. Later the flash task may respond and this is handled by the web_server_flash_response() case which will then progress the TCP transaction.

The task handling the flash should look something like this:

```
void flash_handler(chanend c_flash, fl_SPIPorts &flash_ports) {
  web_server_flash_init(flash_ports);
  // Initialize your application code here
  while (1) {
    select {
    case web_server_flash(c_flash, flash_ports);
    // handle over application events here
    }
  }
}
```

Again this task may perform other application tasks (that may access the SPI flash) as well as assisting the web server.

1.5.4 Communicating with other tasks in webpage content

It is possible to communicate with other tasks from dynamic content calls when rendering the webpage. The top-level main of your program will be something like the following:

```
int main() {
   ...
   par {
    ...
    on tile[1]: xtcp(c_xtcp, 1, i_mii,
                     null, null, null,
                     i_smi, ETHERNET_SMI_PHY_ADDRESS,
                     null, otp_ports, ipconfig);

    on tile[1]: tcp_handler(c_xtcp[0]);
    ...
    }
}
```

Where tcp_handler is the task that calls the web event handler and serves the web pages. Now suppose, we add a new task that we want to communicate to via the web page (in this example, an I2C bus):

```
int main() {
   ...
   par {
    ...
    on tile[1]: xtcp(c_xtcp, 1, i_mii,
                     null, null, null,
                     i_smi, ETHERNET_SMI_PHY_ADDRESS,
                     null, otp_ports, ipconfig);

    on tile[1]: tcp_handler(c_xtcp[0], i_i2c);
    on tile[1]: i2c_master(i_i2c, 1, p_scl, p_sda, 100);
    ...
    }
}
```

The `i_i2c` connection to the `i2c_master` task is passed to the `tcp_handler`.

The `tcp_handler` needs to store the connection to the I2C task as a global to allow the web-page function to access it. This is done via a xC *movable* pointer at the start of the task:

```
client i2c_master_if * movable p_i2c;

void tcp_handler(chanend c_xtcp, client i2c_master_if i2c) {
   client i2c_master_if * movable p = &i2c;
   p_i2c = move(p);

   xtcp_connection_t conn;
   web_server_init(c_xtcp, null, null);
   ...
```

This has *moved* a pointer to the I2C interface to the global variable `p_i2c`. This can now be used in a web-page function. In a seperate file you can write:

```
extern client i2c_master_if * movable p_i2c;

void do_i2c_stuff() {
   i2c_regop_res_t result;
   result = p_i2c->write_reg(0x45, 0x07, 0x12);
}
```

This function can be called from a dynamic expression in a webpage e.g.:

```
{% do_i2c_stuff() %}
```

Remember that functions call from dynamic webpage content must be prototyped in the `web_server_conf.h` header in your application.

2 API - Configuration defines

These defines can either be set in your Makefile (by adding -DNAME=VALUE to XCC_FLAGS) or in a file called web_server_conf.h within your application source tree (this will get examined by the library).

Macro	WEB_SERVER_PORT
Description	This define controls what port the server listens on.

Macro	WEB_SERVER_USE_FLASH
Description	This define controls whether the web server gets the pages from SPI flash or not. Set to 1 to use flash and 0 not to use flash.

Macro	WEB_SERVER_SEPARATE_FLASH_TASK
Description	This define sets whether the flash access should be done within the web server functions or offloaded via a channel to another task.

Macro	WEB_SERVER_FLASH_DEVICES
Description	This define should be set to the name of a global variable that defines the array of possible flash devices that the web server connects to.

Macro	WEB_SERVER_NUM_FLASH_DEVICES
Description	This define sets the size of the array of possible flash devices defined by the variable set by.

Macro	WEB_SERVER_POST_RENDER_FUNCTION
Description	This define can be set to a function that will be run after every rendering of part of a web page. The function gets passed the application and connection state and must have the type: `void render(int app_state, int connection_state)`

3 API

All functions can be found in the web_server.h header:

```
#include <web_server.h>
```

3.1 Web Server Functions

The following functions can be uses in code that has a channel connected to an XTCP server task. The functions handle all the communication with the tcp stack.

Function	**web_server_init**
Description	Initialize the web server. This function initializes the webserver.

Type	`void web_server_init(chanend c_xtcp,`
	`chanend ?c_flash,`
	`fl_SPIPorts &?flash_ports)`

Parameters	`c_xtcp` chanend connected to the xtcp server `c_flash` If the webserver is configured to use flash where flash access is in a separate task. Then this chanend parameter needs to connect to that task. Otherwise it should be set to null. `flash_ports` If the webserver is configured to use flash and flash access is not in a separate task. Then this parameter should supply the ports to access the flash. Otherwise it should be set to null.

Function	**web_server_handle_event**
Description	Handle a webserver tcp event. This function should be called when a TCP event is signalled from the xtcp server. If the event is for the webserver port, the function will handle the event.

Type	`void`
	`web_server_handle_event(chanend c_xtcp,`
	`chanend ?c_flash,`
	`fl_SPIPorts &?flash_ports,`
	`xtcp_connection_t &conn)`

Continued on next page

Parameters	c_xtcp	chanend connected to the xtcp server
	c_flash	If the webserver is configured to use flash where flash access is in a separate task. Then this chanend parameter needs to connect to that task. Otherwise it should be set to null.
	flash_ports	If the webserver is configured to use flash and flash access is not in a separate task. Then this parameter should supply the ports to access the flash. Otherwise it should be set to null.
	conn	The tcp connection structure containing the new event

Function	**web_server_set_app_state**
Description	Set the application state of the web server. This sets a single integer (which could be cast from a pointer in C) to be the application state of the server. This state variable can be accessed by the dynamic content of the server.
Type	`void web_server_set_app_state(int st)`

3.2 Separate Flash Task Functions

If WEB_SERVER_SEPARATE_FLASH_TASK is enabled then a task with access to flash must use these functions. This task will need to follow a pattern similar to:

```
void flash_handler(chanend c_flash) {
  web_server_flash_init(flash_ports);
  while (1) {
    select {
      // Do other flash handling stuff here
      case web_server_flash(c_flash, flash_ports);
    }
  }
}
```

Function	**web_server_flash_init**
Description	This function intializes the separate flash task access to the web server data.
Type	`void` `web_server_flash_init(fl_SPIPorts &flash_ports)`
Parameters	flash_ports The ports to access SPI flash (see libflash)

Function	web_server_flash
Description	This function handles the request from the tcp handling task to get some data from flash.
Type	`select web_server_flash(chanend c_flash, fl_SPIPorts &flash_ports)`
Parameters	`c_flash` Chanend connected to the tcp handling functions of the webserver `flash_ports` The ports to access SPI flash (see libflash)

In addition the task handling the xtcp connnections needs to respond to cache events from the flash handling task. This task will need to follow a pattern similar to:

```
void tcp_handler(chanend c_xtcp, chanend c_flash) {
  xtcp_connection_t conn;
  web_server_init(c_xtcp, c_flash, null);
  while (1) {
    select
      {
      case xtcp_event(c_xtcp, conn):
        // handle non web related tcp events here
        web_server_handle_event(c_xtcp, c_flash, null, conn);
        break;
      case web_server_flash_response(c_flash):
        web_server_flash_handler(c_flash, c_xtcp);
        break;
      }
  }
}
```

Function	web_server_flash_response
Description	Select handler to react to a cache response from the. This select handler can be used in a select case to handle the response from the separate flash task with some data. It should be followed by a call to web_server_flash_handler().
Type	`void web_server_flash_response(chanend c_flash)`
Parameters	`c_flash` chanend connected to separate flash task

Function	web_server_flash_handler

Continued on next page

Description	Handle incoming data from flash thread. This function should be called in the body of a select case that responds to the flash task via `web_server_flash_reponse()`.
Type	`void` `web_server_flash_handler(chanend c_flash, chanend c_xtcp)`
Parameters	`c_flash` chanend connected to separate flash task `c_xtcp` chanend connected to the xtcp server

3.3 Functions that can be called during page rendering

When functions are called during page rendering (either via the `{% ... %}` template escaping in the webpages or via the `WEB_SERVER_POST_RENDER_FUNCTION` function), the following utility functions can be called.

Function	**web_server_get_param**
Description	Get a web page parameter. This function looks up a parameter that has been passed to the current page request via either a GET or POST request.

Type	`char*`
	`web_server_get_param(const char *param,`
	`int connection_state)`

Parameters	`param` The name of the parameter to look up `connection_state` The connection state of the page being served
Returns	a pointer to the parameters value. If the parameter was not passed in as part of the HTTP request then NULL is returned.

Function	**web_server_copy_param**
Description	Copy a web page parameter. This function looks up a parameter that has been passed to the current page request via either a GET or POST request and copies it into a supplied buffer

Type	`int`
	`web_server_copy_param(const char param[],`
	`int connection_state,`
	`char buf[])`

Continued on next page

Parameters	`param` The name of the parameter to look up `connection_state` The connection state of the page being served `buf` The buffer to copy the parameter into
Returns	the length of the copied parameter value. If the parameter was not passed in as part of the HTTP request then 0 is returned.

Function	**web_server_is_post**
Description	Determine whether the current page request was a POST request.
Type	`int` `web_server_is_post(int connection_state)`
Parameters	`connection_state` The connection state of the page being served
Returns	non-zero if the page request is a POST request. Zero otherwise.

Function	**web_server_get_current_file**
Description	Return the file handle of the current page.
Type	`file_handle_t` `web_server_get_current_file(int connection_state)`
Parameters	`connection_state` The connection state of the page being served
Returns	A file handle for the current page. See `simplefs.h` for details.

Function	**web_server_end_of_page**
Description	Have we served the entire page. This function is usually used in the `WEB_SERVER_POST_RENDER` function to determine whether the whole page has been fully served (for example to update a page visit counter).
Type	`int` `web_server_end_of_page(int connection_state)`

Continued on next page

Returns	non-zero if the page has been fully served. Zero otherwise.

APPENDIX A - Known Issues

There are no known issues with this library.

APPENDIX B - Embedded webserver library change log

B.1 2.0.1

- Update to source code license and copyright

B.2 2.0.0

- Rearrange to new file structure
- Change examples to work with new xtcp and ethernet libraries
- Changes to dependencies:
  - lib_gpio: Added dependency 1.0.0
  - lib_otpinfo: Added dependency 2.0.0
  - lib_locks: Added dependency 2.0.0
  - lib_xtcp: Added dependency 4.0.0
  - lib_ethernet: Added dependency 3.0.0
  - lib_xassert: Added dependency 2.0.0
  - lib_logging: Added dependency 2.0.0

B.3 Legacy release history

B.4 1.0.3

- Various documentation updates
- Changes to dependencies:
  - sc_xtcp: 3.1.3rc0 -> 3.2.1rc0
    - Fixed channel protocol bug that caused crash when xCONNECT is
    - Various documentation updates
    - Fixes to avoid warning in xTIMEcomposer studio version 13.0.0
  - sc_ethernet: 2.2.4rc0 -> 2.3.1rc0
    - Fix invalid inter-frame gaps.
    - Adds AVB-DC support to sc_ethernet
    - Various documentation updates
    - Fixed timing issue in MII rx pins to work across different tools
    - Moved to version 1.0.3 of module_slicekit_support
    - Fixed issue with MII receive buffering that could cause a crash if a packet was dropped
  - sc_slicekit_support: 1.0.3rc0 -> 1.0.4rc0
    - Fix to the metainfo.
  - sc_wifi: 1.0.0rc0 -> 1.1.2rc0
    - Other document updates
    - Document updates conforming to xSOFTip style.
    - Resolve connection to router bug
  - sc_util: 1.0.2rc0 -> 1.0.4rc0
    - module_logging now compiled at -Os
    - debug_printf in module_logging uses a buffer to deliver messages unfragmented
    - Fix thread local storage calculation bug in libtrycatch
    - Fix debug_printf itoa to work for unsigned values > 0x80000000
    - Remove module_slicekit_support (moved to sc_slicekit_support)
    - Update mutual_thread_comm library to avoid communication race conditions

  - sc_spi: 1.3.0rc1 -> 1.4.0rc0
    * Added build option to allow SD card driver compatibility
    * Updated documents
    * Updated documents

B.5 1.0.2

- Update to xtcp v3.1.3

B.6 1.0.1

- Fix content-length handling bug
- Enable use with WIFI module

B.7 1.0.0

- Initial Version