# Simple HTTP web server for small embedded systems

## Design objectives

This web server design needs to be:

* Light weight and small embedded system friendly
* Portable across RTOS and bare-metal systems
* Portable across networking mechanisms
* Portable across filesystems.

## Functionality

This web server design needs to include the following capabilities

* Support for static web pages
* Support for dynamic pages including:
  + Support for Server Side Includes (SSI)
  + Support for CGI

## Architecture and implementation overview

In case you are not interested in understanding the guts of this server, directly jump to the section dealing with usage. (Usage)

HTTP\_NET

HTTP\_FILE

Platform Network

Platform FS / Local FS

HTTP\_REQUEST\_PARSER

HTTP\_RESPONSE\_HEADER

HTTP\_CGI

HTTP\_SSI

HTTP SERVER

HTTP\_CONFIG

From the network’s point of view, this simple webserver works as a socket handler. The platform network handles the process of listening on a port and binding the incoming connection to a socket. Once this is done, control of the socket is handed over to the server. The server will then do the following operations

* Read the client request
* Parse client request to identify:
  + HTTP Method being requested
  + Requested file path
  + Requested file class
* Perform file operations or processes w.r.t. the file path and file class requested.
* Send response header and contents to the client.
* Close the connection socket.

Details of tasks performed by individual blocks are described in the following sections

## Building Blocks

### HTTP\_CONFIG

This layer is essentially a static configuration layer that lets the user fine tune memory usage and other configurations of the server. Refer to comments and documentation in http\_config.h to understand these configurations.

### HTTP\_NET

This module handles the interface between platform network layer and the server’s network code. During initialization, the following three functionality from underlying network layer needs to be registered by the platform porting code with http\_net

* Network read.
* Network write.
* Network disconnect.

This layer provides portability across different underlying network layers.

### HTTP\_FILE

This module handles the interface between platform file operations and server’s file handling code. During initialization , the following functionality from underlying filesystem needs to be registered by the platform porting code with http\_file

* fopen
* fclose
* fread
* fgetc
* fseek
* feof

File contents are read (and parsed) one block at a time. This block size is controlled in the configuration module. If the file size is larger than the configured block size, the server will fall back to a chunked transfer-encoding method to read and “serve” the full file. In case of an SSI file, only half a block is read so as to facilitate in-situ replacement without additional memory requirement.

### HTTP\_REQUEST\_PARSER

This module is responsible for reading the client request and parsing it to identify the HTTP method being requested , the file path and the file class.

File class is linked to the type of operation to be performed by the file path. Currently, we support 3 file classes based on the extension of the requested file path.

* SSI : for file paths with extensions "shtml", "shtm", "ssi", "xml"
* CGI : for file paths with extensions "cgi", "sh", "exe"
* None: for all files not falling in the above categorization

More files types in each class can be added by modifying httpFileType\_SSIList and httpFileType\_CGIList in http\_request\_parser.c

If a file class is identified as httpFileClass\_none the file is directly served out as a static file. Refer to section on HTTP\_RESPONSE\_HEADER to understand how the response header is formed.

In case the file type is identified as httpFileClass\_SSI or httpFileClass\_CGI one of the corresponding modules comes in to picture as described below.

### HTTP\_SSI

SSI refers to Server Side Includes. These are special tags placed in an otherwise normal (static) html file to make the server dynamically replace contents thereby making it a dynamic page.

This server lets the users define SSI strings to be used along with a function to be called to get the string with which the SSI string is to be replaced with. The function http\_SSI\_register\_replacer() from HTTP\_SSI is used for making such registrations. The total number of SSI strings supported is controlled in the config module.

To identify SSI strings from among regular html contents, the SSI strings are to be placed inside a special html comment structure as in the below example of an SSI string “timerVal”:

* <!--#timerVal#-->

When the server is requested to GET a path with an SSI extension, it will first read the file contents from the filesystem. This is followed by a replacement routine where all the registered SSI values are replaced with replacement strings dynamically generated by the corresponding callbacks.

**Note**: all replacement strings are computed beforehand before starting the replacement operation to avoid disparity in contents due to network delays.

**Note:** In case of an SSI file, only half a block of the configured read block size is read so as to facilitate in-situ replacement without additional memory requirement.

### HTTP\_CGI

HTTP CGI module is responsible for responding to requests coming in to files and paths registered under the CGI file class. Upon server initialization, the platform porting code will register a set of “paths” and corresponding “path functions” and response content types. When client requests for a registered file path in the CGI file class, the corresponding path function will be invoked. In case the pathfunction returns contents, it is served out under the registered content type. In case, of no response contents, only a success response is send back to the client.

### HTTP\_RESPONSE\_HEADER

When an http response is to be sent back to the client, this module is responsible for forming the headers based on the inputs provided by the core server’s core logic.

When this module is requested for a response header using the file path , it uses some of the hardcode mappings to find the content type associated with the file extension. However, the server code can also request this module to form a header based on explicitly mentioned content types. An example for this is for contents served as a response from CGI pathfunctions.

## Local FileSystem

The flash based local fs is not technically a part of the server. However, since it is critical in illustrating the functionality, we will look at the details of this system here.

The local file system maintains the file list in a buffer named http\_local\_filesystem . At a high level, this is similar to inittabfs in Linux. Maximum number of files in this fs is controlled by the configuration parameter HTTP\_LOCAL\_FILESYSTEM\_NUMFILES. Actual user provided files (contents) are global character arrays. These must be registered to the local fs using http\_localfs\_registerFile() . Once registered, the local fs implementation provides standard file operations like open, close, read, write, seek, rewind e.t.c that can be performed on these files. Files can also be dynamically registered and de-registered. These file APIs has the same signature and functionality as that of stdlibc file operations.

To convert a regular file into an array to be used with the local fs, use the linux tool “xxd -i” . There is also a python implementation of this tool provided with the server release for use in non-linux platforms.

## Usage

The server code is written to be as portable and compatible with multiple system implementations as possible. This section explains how the port of this server into a platform with a SAMG55 host and WINC1500 is implemented.

http\_server\_platform.c implements the porting layer and all hooks up to the networking and localfs layers.

The port function http\_server\_WINC15x0\_init() performs all the initialization tasks. tcpServerSocketEventHandler() is to be called from within the WINC socket callback to process webserver specific tasks.

http\_server\_WINC15x0\_net\_read() , http\_server\_WINC15x0\_net\_write() and http\_server\_WINC15x0\_net\_disconnect() implements the hook ups to the underlying WINC driver and is registered with the server using http\_net\_register\_netops().

In the reference example, we have three “files” each from a different file class. “Index.html” and “timer.ssi” contents are in the resources folder as index\_html.c and timer\_ssi.c . The original file contents are placed in the same files under comments. These are registered to the file system using http\_localfs\_registerFile() in http\_server\_WINC15x0\_init() . The CGI path registered is “ledt.cgi” and is mapped to the pathfunction httpServerTest\_cgiPathFunction() using http\_CGI\_register\_pathFunction() .

timer.ssi contains two SSI Strings in the html file contents. These are:

* <!--#timerVal#--> to include a server uptime millisecond counter in the served page and
* <!--#timerValCC#--> to include the total number of times the page has been requested.

These strings and corresponding functions are mapped using http\_SSI\_register\_replacer()

This flow is illustrated in the following init flow diagram:

Init local file system (LFS) and register files to it.

http\_localfs\_ini(), http\_localfs\_registerFile()

Register fops to http\_file

http\_file\_register\_fops()

Register netops to http\_net

http\_net\_register\_netops()

Register SSI strings and replacers

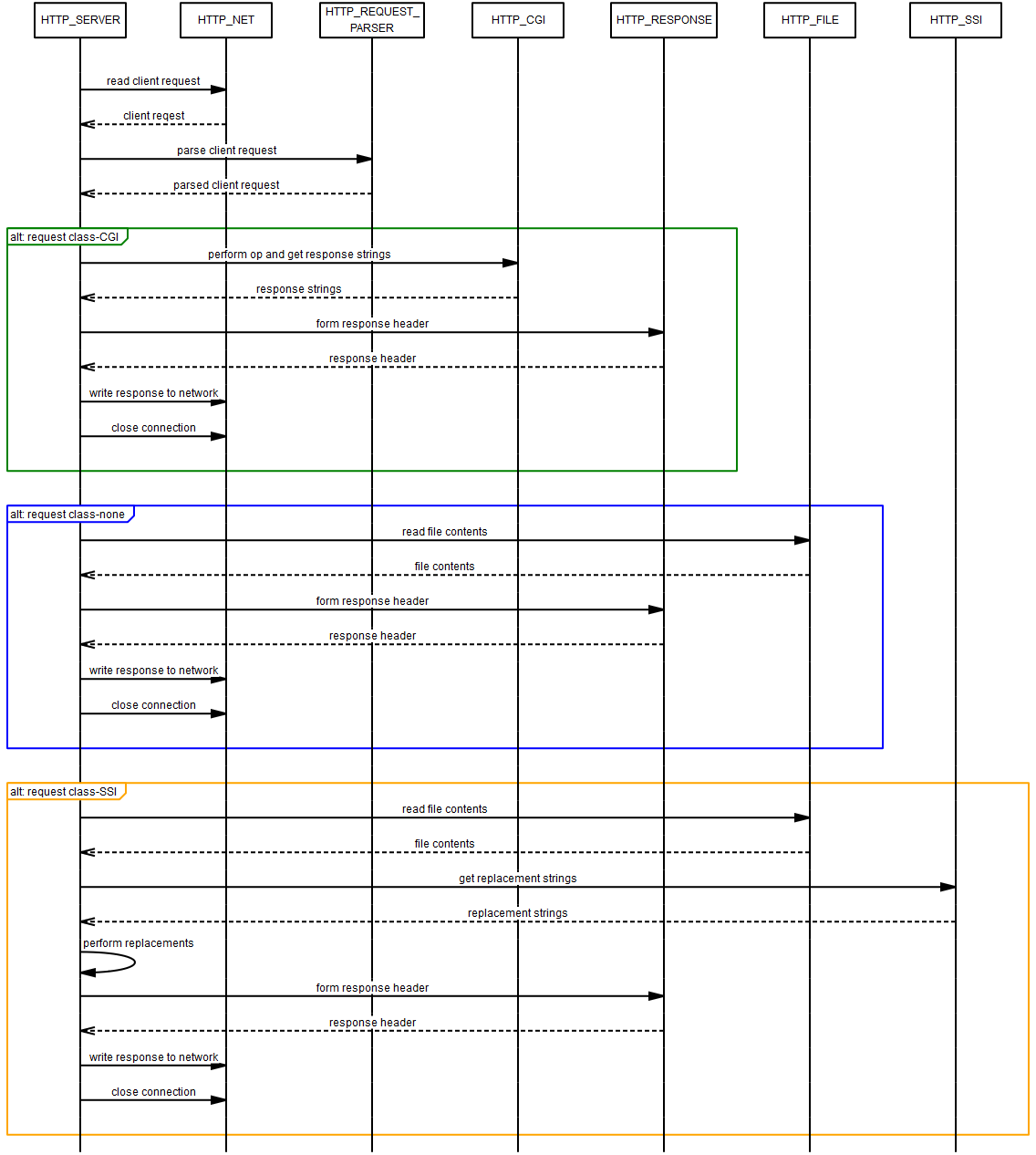
http\_SSI\_register\_replacer()

Register CGI paths and pathfunctions

http\_CGI\_register\_pathFunction()

Now, run a TCP server in the main loop in port 80 and call http\_server() every time a connection is accepted and bound in SOCKET\_MSG\_ACCEPT

The operation flow within http\_server once it is called after binding a client is illustrated in the flow below.



## Tests

The core server stack is written in pure ANSII C (C-99 standard). This is to ensure that the stack can be compiled and run using any standard microcontroller tool chain. To ensure that the compliance is met during all phases of development, a CI system has been linked to the Github repo to compile the stack with all pedantic warnings and using two compilers (GCC and Clang).

## Limitations

The current version of the webserver stack has the following limitations.

* Supports only GET method
* No support for query strings
  + Future development is planned to enable this only on CGI fileClass.
* No custom error pages
  + Error messages like 500, 404 etc. will be proper but without page contents.
* Only one filesystem will be supported at a time
  + Current example uses localfilesystem. This can be easily replaced with an SD card based FS. But this has not been tested.
* Only one network interface will be supported at a time
* SSI strings in file read block boundaries will result in undefined behavior.
  + This will be addressed in future releases.