**Freescale Semiconductor**
Application Note

Document Number: AN3455
Rev. 0, 04/2007

# ColdFire Lite HTTP Server

By Eric Gregori

## 1 Introduction

Hyper-Text Transport Protocol (HTTP) is the communication protocol of the world wide web. HTTP is used to transfer web pages (Hyper-Text documents) across the internet. An HTTP connection consists of two parts: the HTTP client (web browser) and the HTTP server. You can use the HTTP client to receive and view the web page and the HTTP server to store, organize, and transfer the web pages.

RFC (Requests For Comments) are the technical specifications that define the internet. RFCs are published by the Internet Engineering Task Force (IETF). See http://www.rfc-editor.org/ for more information on RFCs. RFC2616 defines HTTP version 1.1, which is the latest version. RFC1945 defines HTTP version 1.0.

**Contents**





freescale™ semiconductor

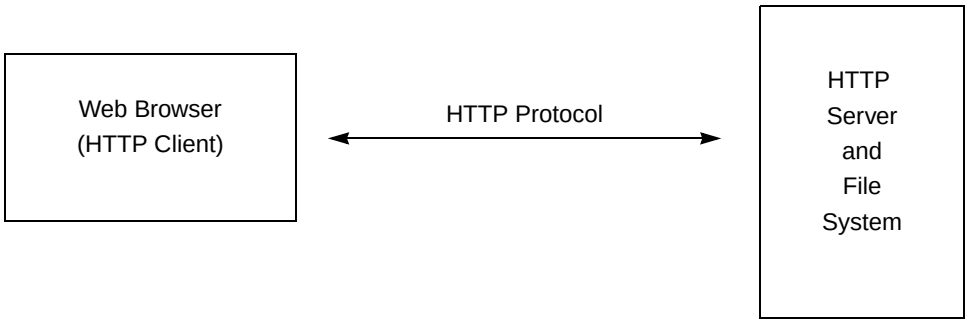


**Figure 1. HTTP Protocol**

# 2 HTTP Request Response Protocol

HTTP is a request response protocol. The client requests a web page from the server and the server responds with the web page contents (HTML-Hyper-Text Markup Language). HTTP can be used to send any type of data, including binary data. The client requests a file using the GET method (HTTP is an ASCII protocol). The server responds with an HTTP header followed by the file contents. The client can also send a file using a POST method. Within the request, the HTTP version is embedded in ASCII. This notifies the server of the limitations of the client.

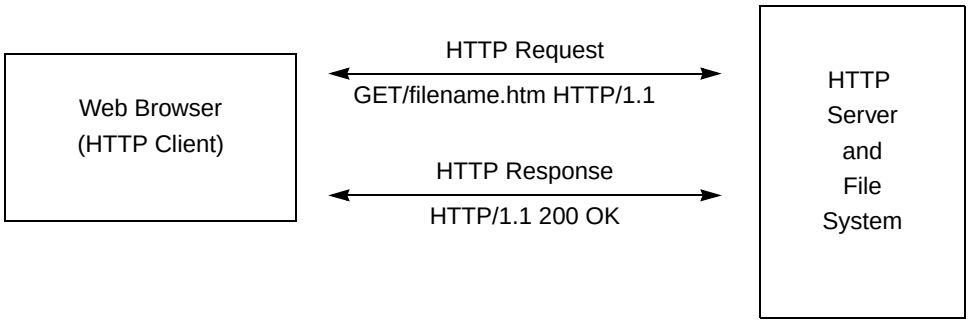


**Figure 2. HTTP Request/Response**

The following is sent from the client (web browser) to the HTTP server:

- GET/filename.htm HTTP/1.1
  - Asks the server to respond with the contents of the filename.htm
  - Tells the server that it supports the HTTP1.1 standard
- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjeg, application/msword
  - Tells the server that the client supports: if, x-xbitmaps, jpeg, and pjeg images
  - Tells the server that it supports msword documents
- Accept-language: en-us
- Accept-Encoding: gzip, deflate
  - Tells the server that the language is English, and that the gzip and deflate decompression algorithms are available
- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)

  - It tells the server that the browser is running IE6.0 on a Windows machine
- Host: www.msn.com
- Connection: Keep-Alive
  - Finally, it tells the server not to close the connection after the file is sent

The GET method is only one of the methods supported by RFC2616. Other methods are listed in Table 1.

**Table 1. Methods Supported by RFC2616**

| METHOD | RFC2616 | DESCRIPTION |
|---|---|---|
| “OPTIONS” | Section 9.2 | Request for information |
| “GET” | Section 9.3 | Request for file or data |
| “HEAD” | Section 9.4 | Identical to GET without a message-body in the response |
| “POST” | Section 9.5 | Send data |
| “PUT” | Section 9.6 | Send a file |
| “DELETE” | Section 9.7 | Delete a file |
| “TRACE” | Section 9.8 | Echo request |
| “CONNECT” | Section 9.9 | Reserved for tunnelin |

# 3 HTTP Reponse Example

The server responds to the GET method with a header followed by severa data or file contents. The HTTP/1.1 200 OK line tells the client/browser that HTTP1.1 is supported and the status code tells the client that the file was found. The server line informs the client of the web server type and version (Microsoft-IIS/6.0). The Cache-Control line tells the client to disable the cache (no-cache). The Content-Type line tells the client the type of data that follows (text/html). The Content-Encoding line tells the client how the following data is encrypted (gzip). The Content-Length line tells the client how many bytes follow (9062).

After the HTTP server sends the header, it follows with the file contents or data in raw binary (not ASCII) format.

**Table 2. Status Codes and Descriptions**

| Status Code | Description |
|---|---|
| 1xx | Informational |
| 2xx | Successful |
| 3xx | Redirection |
| 4xx | Client Error |
| 5xx | Server Error |

# 4 Transporting HTTP

HTTP defines the conversation that occurs between the server and the client using the transport protocol. HTTP can be transported using any protocol that supports bi-directional ASCII. The transport protocol for the internet is the Transport Control Protocol/Internet Protocol (TCP/IP).

TCP/IP is a connections based protocol. HTTP does not have an ACK or retry mechanism, and instead relies on the transport protocol to provide this service. The TCP portion of TCP/IP provides a negotiation system for establishing the connection. It does this using an ACKnowledgement and retry system.

HTTP does not provide a method that guarantees data transmission from server to host. This is provided by the TCP.

The Internet Protocol (IP) provides a mechanism to support multiple TCP connections. These mechanisms are referred to as ports. For each IP connection, there are 65,534 ports. Each IP connection has a unique IP address and every node on the internet has an IP address. IP addresses consist of 4 bytes written in dot notation. 75.18.69.29, for instance, is a unique address on the internet for a web server.

IP ports are assigned based on the protocol they support. HTTP uses port 80. This means that a standard HTTP server listens for a connection on port 80. HTTP servers can be assigned non-standard ports, but web browsers look for the HTTP server on port 80 by default .

The TCP/IP protocol stack is referred to as a stack because the software is designed as one protocol stacked on top of another. For instance, HTTP is on top of TCP, which is on top of IP.

**Table 3. The 7-Layer OSI Model**

| | |
|---|---|
| Layer 7 — Application | HTTP, SMTP, POP3, TFTP |
| Layer 6 — Presentation | Berkeley Socket Interface, XTI |
| Layer 5 — Session | Berkeley Socket Interface, XTI |
| Layer 4 — Transport | TCP, UDP |
| Layer 3 — Network | IP, ARP, ICMP |
| Layer 2 — Data Link (MAC) | Ethernet, PPP |
| Layer 1 — Physical (PHY) | RS232, 10BASE-T, DSL, T1 |

# 5 The Presentation Layer (Socket Interface)

At the top of a TCP/IP stack is the presentation layer. This layer is the interface the software engineer accesses as he works with the TCP/IP stack. The software engineer needs to know how to use the presentation layer to use the stack. The Berkeley socket interface is the most common presentation layer. It is supported in Linux, Windows, and many other embedded stacks.

**Table 4. Berkeley Socket Interface Function**

| Berkeley Socket Interface Function | Description |
|---|---|
| Socket () | Creates a socket |
| Bind () | Binds a socket structure to a socket |
| Listen () | Listens on a socket for a connection |
| Accept () | Accepts a connection on a listening socket |
| Connect () | Connects to a listening socket |
| Recv () | Sends data to the socket |
| Send () | Sends data to a socket |
| Close () | Closes the socket |

The Berkeley socket interface is a very robust Applications Interface (API). The HTTP server only uses a few of the available commands. The socket is the core component of the interface, and can be thought of as one end of a communications pipe. Every connection occurs through a socket. The server opens a listening socket and the client opens a connecting socket. Using the TCP layer, the two sockets negotiate a connection creating a bi-directional communications pipe with a socket on each side. After establishing the communications pipe, the client and server communicate by simply sending data to or receiving it from the socket.

The HTTP server uses the socket interface to create a listening socket on port 80. The IP layer is assigned a unique IP address during the initialization of the TCP/IP stack. By opening a listening socket on port 80, the HTTP server is opening a listening port on the IP stack's unique IP address. Using the dot nomenclature, this would be 75.18.69.29:80.

The client (web browser) opens a connection socket, specifying the server's IP address and HTTP port (80). After the server accepts the connection, the two sockets connect using TCP. After the connection is accepted, the client and server communicate using the recv() and send() interface functions. Although TCP/IP communicates in packets, the application does not need to packetize its data since the TCP/IP stack

manages that. The application simply sends a chunk of data to the stack. The stack breaks the data chunk into packets and handles all the buffering.

The ColdFire TCP/IP stack does not support full Berkeley sockets. It uses a mini-socket interface. Table 5 maps Berkeley socket functions to mini-socket functions. Table 6 maps Berkeley socket functions to mini-socket functions for server applications.

**Table 5. ColdFire Mini-Sockets Interface**

| Mini-Sockets | BSD Sockets |
|---|---|
| m_socket () | socket () |
| m_connect () | connect() |
| m_recv() and/or m_send()<br>- or -<br>tcp_send() and/or tcp_recv() -<br>(zero-copy I/O) | recv() and/or send() |
| m_close() | close() |

**Table 6. ColdFire Mini-Sockets Interface for Server Applications**

| Mini-Sockets | BSD Sockets |
|---|---|
| (n/a - merged with listen) | socket() |
| (n/a - merged with listen) | bind() |
| m_listen() | listen) |
| (n/a - handled via callback) | accept() |
| m_recv() and/or m_send()<br>- or -<br>tcp_send() and/or top_recv() -<br>(zero-copy I/O) | recv() and/or send() |
| m_close() | close() |

The biggest difference is on the server side. With mini-sockets, the user only needs one function, m_listen(), to open a listening socket. Mini-sockets do not support the accept() function. Instead, when a client connects to the server, the stack calls a callback function to tell the HTTP server that the client is connecting.

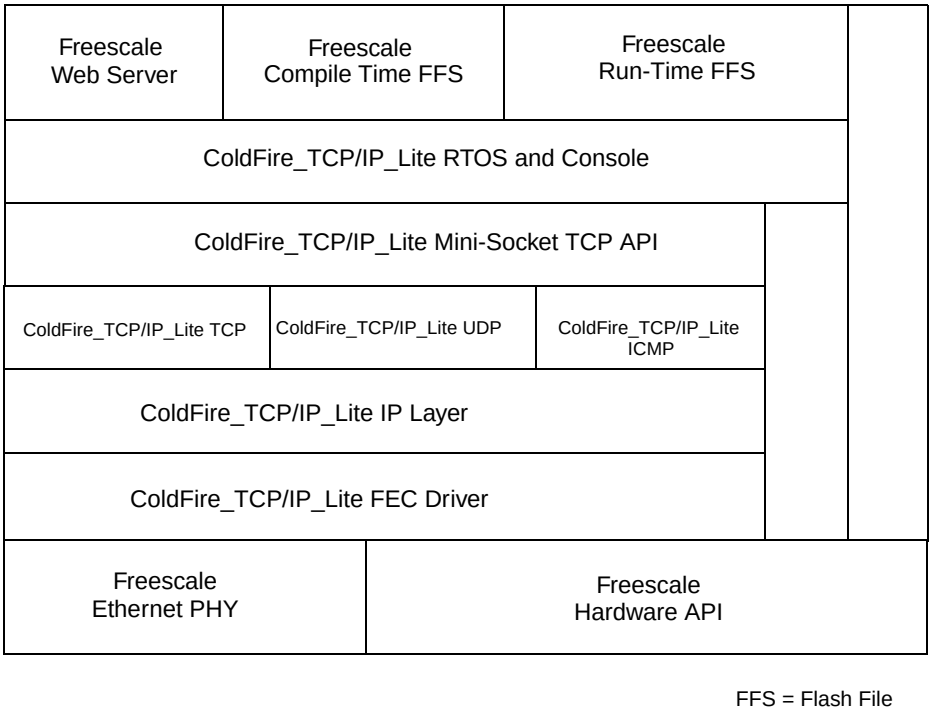


**Figure 3. ColdFire TCP/IP Stack**

# 6 Dynamic HTML

HyperText Markup Language (HTML) is used to describe a web page. An HTTP server transfers HTML pages (web pages) from the server to the client (Internet Explorer, Netscape, Firefox). Standard HTML is static and does not support dynamic content (sensor values, real-time data, system variables). This is a problem for embedded systems that can be solved by embedding tags in the HTML that are replaced with the dynamic data.

Dynamic HTML Tokens allow variable content like sensor data to be inserted into web pages without any programming. Inserting the token ~IIF; into your HTML replaces the token with the data referenced by II. Conditional tokens take this idea one step further by allowing entire HTML string to be replaced based on a data comparison to a constant.

The server supports a variable array. Each entry in the array is 32 bits (long word). The array's depth is user-configurable at build time. The array is updated when a web page containing dynamic tags is requested. The user chooses what data goes into the array at compile time. For instance, the user may choose to put A/D channels zero through eight into array positions zero through eight.

Each tag is followed by an index. The index chooses which variable in the array replaces the token when the HTML is actually sent to the client. There are two types of tokens: replacement tokens and conditional replacement tokens.

## 6.1 Replacement and Conditional Tokens

Replacement tokens, '~', are always replaced with the data in the variable indexed or with a dash mark to indicate the variable is not up to date.

```
Process sensor data request
  ~IIF;
  where
                I      = Variable Index
                F      = Format (H=hex, D=decimal)
```

In this example, <HTML> ~0D; </HTML>, the server replaces the characters ~00D; with the decimal value in variable 00.

Conditional tokens, '^', are used for string replacement based on the result of the conditional compare and a constant.

```
Process conditional sensor read
^II>C|true||false|;
where:
                I       = Variable Index
                C       = hex value for comparison
                >       = variable value greater then C
                =       = variable value equal to C
                &       = variable value and C
                !       = !variable value and C
                "true"  = ASCII string to replace if true
                "false" = ASCII string to replace if false
```

### 6.1.1 Conditional Token Example:

```
<HTML> ^00&01| replacement string if true | replacement string when false |; </HTML>
```

The server replaces the whole conditional statement with the true string if the variable 00 and 0x01 is true. Otherwise, the false string replaces the conditional statement. Conditional statements can be equal, greater than, bit and, or not equal. 1The constant can be any value from 0 to 0xFFFFFFFF.

# 7 HTTP Server for ColdFire Devices (Features)

HTTP1.0 is a compliant server that features connection persistence and multiple sessions (HTTP1.1 will be available in future revisions). It supports multiple HTTP connections and includes a flash file system supporting ColdFire internal flash and external SPI flash. Web pages can be updated in flash via Ethernet or built in at compile time. HTTP1.0 supports the HTTP GET method, and contains a simple mechanism for adding other methods.

It also features dynamic HTML support with replace and conditional tokens and serial interface support for the dynamic HTML variables. 'DIR' commands are also supported on serial interface. HTTP1.0 also provides run time and compile time flash file systems and long file name support with subdirectories. It features PC utilities for compressing compile time and run time downloadable images of multi-page web pages. There is also a PC utility for downloading run time downloadable web page images through port 80 (to pass through firewalls). It also has a 32 byte ASCII key for web page download security.

# 8 Firmware

The HTTP server for ColdFire devices is contained in the Freescale_HTTP_Web_Server directory. The full path is: ColdFire_Lite\src\projects\example\freescale_HTTP_Web_Server. This directory includes the freescale_file_api.c file, which is the the software layer between the HTTP server and the file system. It also includes the header file for project, freescale_http_server.h. Figure 4 shows the other files included in this directory along with their functions.

## 8.1 HTTP Server Stack for ColdFire Devices

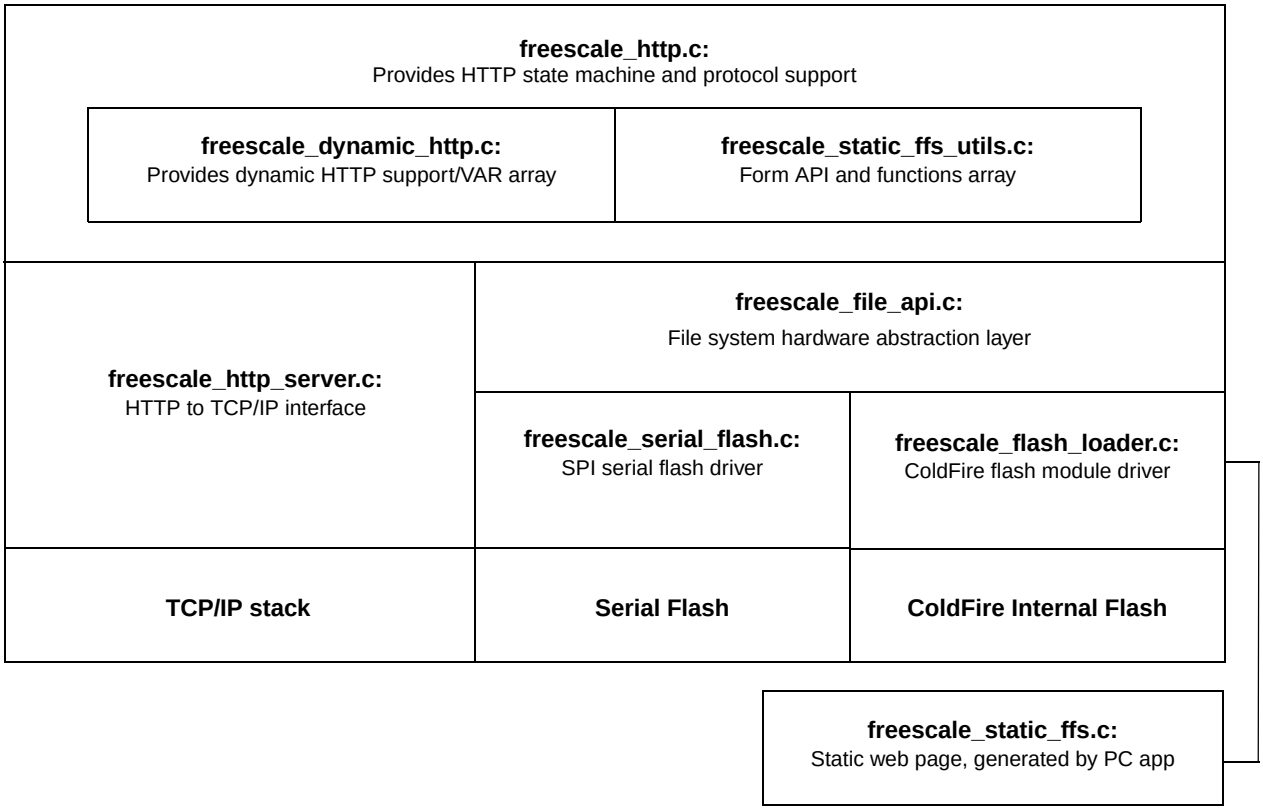


**Figure 4. HTTP Server Stack for ColdFire Devices**

# 9 The File Systems

The HTTP server also includes two flash-based, writable ethernet file systems and one static read-only compile time flash file system. These systems are used to store the web pages, xml files, graphics (BMP, JPG, PNG), and other data. The file systems support ASCII and binary data (filenames can be up to 255 bytes long) as well as directory structures.

Use the compile time FFS to store data that is not expected to change (logos, templates, graphics, etc.). The writable flash file systems can be upgraded remotely over the internet. The writable FFS should be used to hold the web pages and any other pictures or content that may need to be changed in the field.

The file search algorithm always looks for a file in the writable file system before looking in the compile time file system. This allows a file in the compile time system to be overridden (the compile time FFS is read-only, so it cannot be over written). If you store your logo in the compile time system, you can override the logo with a new one by storing a new logo with the same file name into a writable FFS.

## 9.1 The Default File

A web browser can request a default web page when the user does not specify a name in the URL. The HTTP server uses the first file in the compile time FFS or the writable FFS as the default file. It checks the writable FFS first. S‘o the first file in the writable FFS is always the default file, unless there is no files in the writable FFS, then the first file in the compile time FFS is the default.

## 9.2 Compile Time Static Flash File System

```
emg_static_ffs<filelist.txt><output_file.com>
```

The compile time FFS is created as a C file (freescale_static_ffs.c) at compile time. A PC utility called emg_static_ffs.exe converts the web page, binary graphics, and any other files into a single image in a C file format.

Filelist.txt is a text file containing the list of files to compress. Each file should be on its own line. The first file is the default. Comments can be added using a ‘*’ as the first character in a line. Output_file.ffs is the generated file containing all the files in the filelist compressed together, along with File Allocation Table used to reference the files from the Web Server. The filelist.txt file uses the same format as the static image generator discussed above. The .ffs extension was chosen for demonstration purposes.

The files listed below from the emg static web page description file are concatenated into a single C-compatible file.

```
readme.htm
CFCORESEMBLEM.gif
```

The last line must be a blank line with only a CRLF (hit enter in the last blank line). Files are stored in the order that they appear. If there are no files in the writable FFS, then readme.htm (in the above example) becomes the default file.

## 9.3 Storing Data in the Compile Time FFS

The C file created by the emg_static_ffs.exe utility consists of a group of C arrays. The data from each file is converted to a C array with the same name as the file. This is followed by an array of filenames, an array of pointers to the data arrays, an array of file lengths, an array of file types, and the number of files in the image.

```
//*********************************************************************
//* Static Flash File System Generator                                *
//* Written by Eric Gregori – Chicago FAE                             *
//*                                                                   *
//*********************************************************************

    const unsigned char readme_htm[] = {
    0x48,0x54,0x54,0x50,0x2F,0x31,0x2E,0x31,0x20,0x32,
    0x30,0x30,0x20,0x4F,0x4B,0x0D,0x0A,
         .....
    0x6D,0x6C,0x3E,0x0D,0x0A,0x00 };

    const unsigned char CFCORESEMBLEM_gif[] = {
    0x48,0x54,0x54,0x50,0x2F,0x31,0x2E,0x31,0x20,0x32,
         …..
    0x6F,0x17,0x10,0x00,0x3B,0x00 };

    const char *emg_static_ffs_filenames[] =   {
                                               “readme.htm”,
                                               “CFCORESEMBLEM.gif”
                                             };

    const unsigned char *emg_static_ffs_ptrs[] =   {
                                               readme_htm,
                                               CFCORESEMBLEM_gif
                                             };

    const unsigned short emg_static_ffs_len[] =     {
                                               22129,
                                               12919
                                             };

    const unsigned long emg_static_ffs_type[] =     {
                               0x68746d6c,
                               0x67696666
                                             };

    const unsigned char emg_static_ffs_nof = 2;
```

## 9.4 Writable Flash File Systems

You can write to the writable flash file system over Ethernet. Although web pages can be changed in the writable FFS, they are updated as one image. The image consists of a version number followed by a FAT header, followed by a File Allocation Table (array of file descriptors), followed by the binary data from the files.

## 9.5 Storing Data in a Writable FFS Image

```
Long[0] in the image = ‘EMG1’              FAT version
Long[1]                                    FAT id
Long[2]                                    image size in bytes
Long[3]                                    number of files in image
Long[4]                                    pointer to the first file descriptor
```

This header is followed by an array of file descriptors:

```
typedef struct{
                        unsigned longfile_pointer;
                        unsigned longfile_size_in_bytes;
                        unsigned longfile_type;
                }       FAT_FILE_DESCRIPTOR;
```

The file data is stored concatenated together after the last file descriptor.

# 10 Downloading Writable FFS Images to the HTTP Server

The HTTP server for ColdFire devices supports a special method called EMG. EMG puts the web server into a binary download mode. In this mode, a valid writable FFS image is stored in flash. This process erases the old image in flash. For security purposes, a password or key is required to validate the image. Without the correct password/key, the HTTP server rejects the image and does not erase flash. The password is sent to the filename field of the EMG method.

A PC utility supports the unique upload protocol for the writable FFS images. This utility uses port 80, and appears as any other request to the routers and firewalls over the internet for a web page. Most other web servers use FTP or TFTP to upload files. But these protocols are considered dangerous by most firewalls. This prevents the upload of new images. Using port 80 and a “standard” HTTP header, work around the firewalls and router to avoid being locked out of an upload.

The EMG uploader is in the file emg_https_uploader_client.c. This simple protocol uses TCP as the transport layer and 80 as the port. The first 5 bytes sent are ‘EMG /’ followed by the password (up to 32 characters), followed by a ‘ ‘(space) then 4 bytes representing the file length.

For EMG /password LLLL, the password can be up to 32 characters long. LLLL represents the length of the file being sent. This information is sent to the HTTP server. The server replies with an ACK ("Erase Complete") or a NACK, which terminates the download.

If an ACK is received, the file is sent. After the complete file is sent, two additional garbage bytes are sent to purge any buffers. The uploader then waits for a completion message ("Upload Complete").

## 10.1 Using the Dynamic (Writable) Image Creator

```
emg_dynamic_ffs  <filelist.txt>  <output_file.ffs>
```

Where:

- Filelist.txt is a text file containing the list of files to compress. Each file should be on its own line, and the first file is the default.
- Comments can be added using a '*' as the first character in a line.
- Output_file.ffs is the file generated containing all the files in the filelist compressed together, along with File Allocation Table used to reference the files from the Web Server.

The filelist.txt file uses the same format as the static image generator discussed above. The .ffs extension can actually be any extension. It was chosen for demonstration purposes.

# 11 The File API Layer

To support multiple RTOS's and flash drivers, a file API layer sits between the HTTP server and the flash drivers. This layer is designed to mimic the standard C calls for writing and reading files. The macro USE_SERIAL_FLASH determines which flash drivers are called. If defined, the API accesses the serial flash drivers. If not defined, the API accesses the flash internal to the ColdFire (CFM).

## 11.1 File API Functions (in freescale_file_api.c)

```
int emg_open(char *filename, uint32 *data_pointer, uint32 *file_size)
```

- Finds the file descriptor in the FAT and sets data_pointer to start of data.
- Sets file_size to the size of the file in bytes and returns a < 0 if error, 0 = success.

```
unsigned long emg_web_erase(unsigned char session)
```

- Erases the entire flash file system and returns the size of the flash file system.

```
unsigned long emg_web_flash_parms(unsigned long parameter, unsigned char session)
```

- Like IOCTL, determines FFS parameters. The session is ignored.

```
void emg_web_write(unsigned char session, unsigned long *buffer, unsigned long length)
```

- Writes length bytes from the buffer to the FFS.
- The session variable accesses the HTTP server session array.
- Called with small chunks of data from the HTTP server to keep from stalling the TCP/IP stack.

```
void emg_web_rewind(unsigned char session, unsigned long bytestorewind)
```

- Rewinds the session file index by bytestorewind bytes.

```
unsigned long emg_web_read(unsigned char session, unsigned char *buffer_out, unsigned long
max_bytestoread)
```

- Reads max_bytestoread from the FFS to buffer and returns number of bytes read.

```
void emg_web_open(unsigned char session, unsigned char *filename)
```

- Opens the file's filename

- Initializes the HTTP session array with the file attributes (start pointer, length, type).

```
int emg_ffs_dir(void * pio)
```

- Prints a formatted directory of the FFS to the pio stream.

During an FFS image upload, the HTTP server calls emg_web_erase() multiple times from the function freescale_http_erase_flash(). Flash is erased in chunks to keep from stalling the HTTP state machine. Erase progress is reported back to the upload client for display.

After flash is erased, the function Freescale_httpd_upload_file() calls emg_web_write() multiple times with chunks of data of size RECV_BUFFER_SIZE. Again, this is done to keep from stalling the HTTP state machine.

When the HTTP server gets a request for a file, the function Freescale_process_header() calls emg_web_open() to initialize the HTTP session array to the specific file information. Then the function freescale_http_send_file() calls emg_web_read() to read the data from the file.

The 'size' parameter returns the size of the FFS in bytes, and 'strt'returns the start address of the FFS. The 'endd' parameter returns the end address of the FFS.

# 12 Porting the FFS to other RTOS's

The file API layer makes it easier to port the HTTP server to other RTOS's. Instead of storing the image in flash and using the supplied drivers, it could be stored on any file system type that RTOS supports. To architect this, treat the image as one large file while continuing to use the EMG FAT system.

Hardcode a filename for fopen(). The uploaded image containing the EMG FAT is saved on the RTOS file system as a binary file. Extract a file out of the image by opening the hardcoded filename. Use the EMG FAT to index into the image and find the individual files.

For instance, the emg_web_open() function could wrap an fopen function with a standard C library. The file would be open for reading and the file pointer would be inserted into the session array. The emg_web_read() function would then wrap fread(), using the file pointer from the session array.

File writing is similarly wrapped. The emg_web_erase() function would wrap an fopen() for writing (opening a file with the same name automatically erases the old file). Then the emg_web_write() function would wrap the fwrite(). If you want to write to individual files, modifly the upload protocol to support filenames. The read process would only require that fopen() be called with the desired filename instead of a hardcoded filename.

# 13 ColdFire CFM Flash Drivers

Two different writable flash file systems are supported in the HTTP server. The serial FFS uses an external serial flash. The ColdFire Flash Module (CFM)-based FFS uses the Flash internal to the ColdFire.

The CFM drivers execute out of flash, and are compatible only with ColdFires containing multiple CFM arrays. This allows the driver to execute out of one array as it erases/programs the other array. The drivers are simple supporting sector erase, and programming on long word boundaries. The flash drivers are not

called directly from the HTTP server. Instead, an API layer sits between the flash drivers and the HTTP server.

`void flash_init(void)`

- The flash init routine configures the CFM clock to be within spec based on the core frequency.

`volatile void flash_page_erase(unsigned long *address, unsigned long data)`

- Erases a page of flash (address can be anywhere in the page).
- For time considerations, the flash API layer only erases the number of pages required to store the new image.
- The flash API layer calls this function multiple times, incrementing the address to the beginning of each page to erase.
- Data is a don't care.

`volatile void flash_write(unsigned long *address, unsigned long data)`

- Writes a long word (32 bits) of data to address. The flash API layer calls this function multiple times as the new flash image is downloading.

# 14 Serial Flash Drivers and SPI Drivers

The serial flash drivers are designed to support SPI type serial flash devices. IIC drivers are also included if the application calls for IIC flash. The file freescale_serial_flash.c contains the SPI driver functions and the serial flash driver functions. The ColdFire uses a Queued SPI (QSPI). This allows the SPI to perform more functions in hardware, reducing software overhead. The SPI driver takes advantage of this feature by setting up the QSPI for complete multi-byte command transfers.

## 14.1 QSPI Driver Functions (freescale_serial_flash.c)

`void init_serial_flash(void)`

- Initializes the QSPI module for 10 Mhz clock rate
- Initializes the pins for SPI operation.

`void write_to_qspi_ram(uint8 address, uint16 data)`

- Writes 16 bits to QSPI RAM. This function can be used to write to command RAM or data RAM.

```
TX_RAM     = 0x00 – 0x0F
RX_RAM     = 0x10 – 0x1F
COMMAND_RAM = 0x20 – 0x2F
```

`uint16 read_from_qspi_ram(uint8 address)`

- Reads 16 bits from QSPI RAM.

`void start_spi_xfer(uint8 bytes, uint8 csiv)`

- Kicks off the QSPI module, which then "plays back" the commands stored in command RAM.
- Bytes is the number of commands to play. Csiv determines what to do with the chip select after the commands complete.
- This function blocks until the SPI transaction is complete.

## 14.2 Serial Flash Drivers

The serial flash driver supports parts up to 16MB (128Mbits) using 1 byte commands. The command word format supported is:

```
byte 0 = command
byte 1 = (address & 0x00FF0000)
byte 2 = (address & 0x0000FF00)
byte 3 = (address & 0x000000FF)

read command = 0x03
write command = 0x02
write enable      = 0x06
bulk erase= 0xC7
```

```
void read_write_header(uint32 address, uint8 command)
```

- Builds the 4 byte command word into QSPI ram.

```
uint32 read_serial_flash(uint32 device, uint32 address, uint8 *buffer, uint32 bytestoread,
uint32 done_flag)
```

- Reads data from serial flash
- Sends the flash a read command and a start address.
- Brings the CS line for the selected device low.

```
device          = device to read - for future expansion
address         = if <0xFFFFFFFF, address of first byte to read
                  if 0xFFFFFFFF, read next byte
buffer          = pointer to buffer to copy data
bytestoread     = # of bytes to read, can be 0
done_flag       = if >0, unselect CS after data is copied
Returns >       = 0 on success, and <0 on failure
```

```
int write_serial_flash(uint32 device, uint32 address, uint8 *buffer)
```

- Writes to Serial Flash.
- Writes to flash in 256 byte chunks.

```
device        = device to read - for future expansion
address       = Must be on a 256 byte boundary.
buffer        = pointer to buffer containing data
Returns >     = 0 on success, and <0 on failure
```

```
int erase_serial_flash(uint32 device)
```

- Bulk Erase Flash

```
device        = device to erase - for future expansion
Returns >     = 0 on success, and <0 on failure
```

# 15 HTTP to TCP/IP Interface

The HTTP to TCP/IP interface connects the RTOS's TCP/IP stack to the HTTP state machine. This abstraction layer makes porting to different RTOS's easier. The HTTP server listens for a connection on port 80. When it receives a connection, the port is moved from port 80 to a random port, allowing port 80 to listen for another connection. After the port is moved, a new session is created in the session array.

The HTTP to TCP/IP Interface is in the freescale_http_server.c module. The HTTP server requires only a single thread. It handles multiple session by executing one session per pass through the thread. The max number of sessions is defined by the macro MAX_NUMBER_OF_SESSIONS.

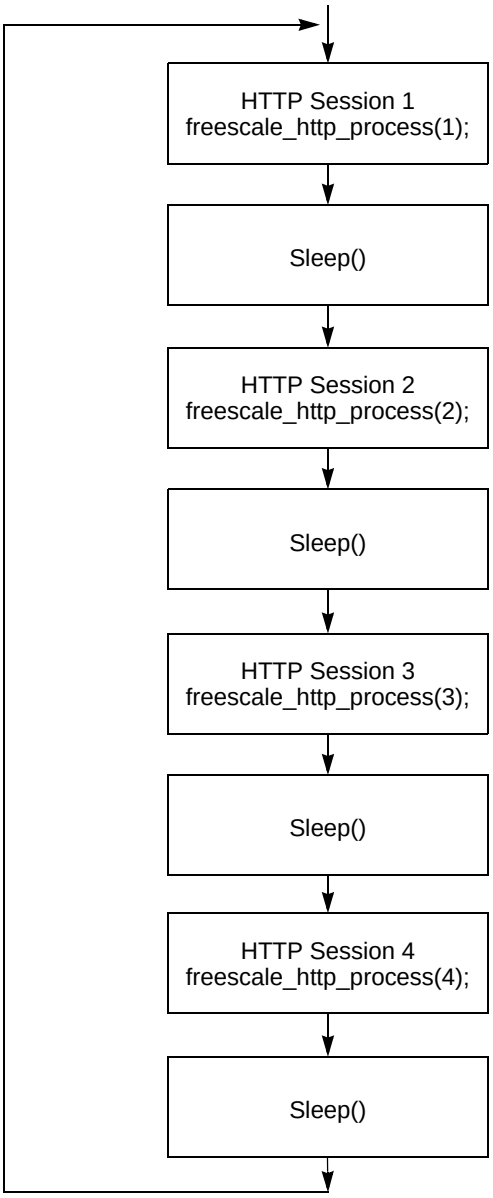


**Figure 5. HTTP to TCP/IP Interface**

The function freescale_http_init() creates a listening socket on port 80 and freescale_http_check() checks the listening socket for a connection. If a connection is found, the function freescale_http_connection() is called to create a session for the connection. After the session is created, it is called by the freescale_http_loop() function for processing.

# 16 HTTP State Machine and Protocol Handler

The HTTP server state machine and protocol handler code is in the freescale_http.c module. The state machine supports up to MAX_NUMBER_OF_SESSIONS concurrent sessions requiring only one thread from the RTOS. The server supports HTTP 1.0 and connection persistence from HTTP 1.1.

Connection persistence, or KEEP ALIVE, is a protocol feature used to increase performance by decreasing TCP/IP overhead. A normal non-persistent HTTP transaction consists of a TCP/IP connect, a GET method, followed by a file transfer, ending with TCP/IP close. This process is followed for every file the client needs (often multiple times with a single web page). The TCP/IP overhead takes a significant amount of time. With a persistent connection, the TCP/IP connect only occurs before the first file transfer, the TCP/IP connection is not closed after the file transfer. Instead the server goes into a stat waiting for another method (GET).

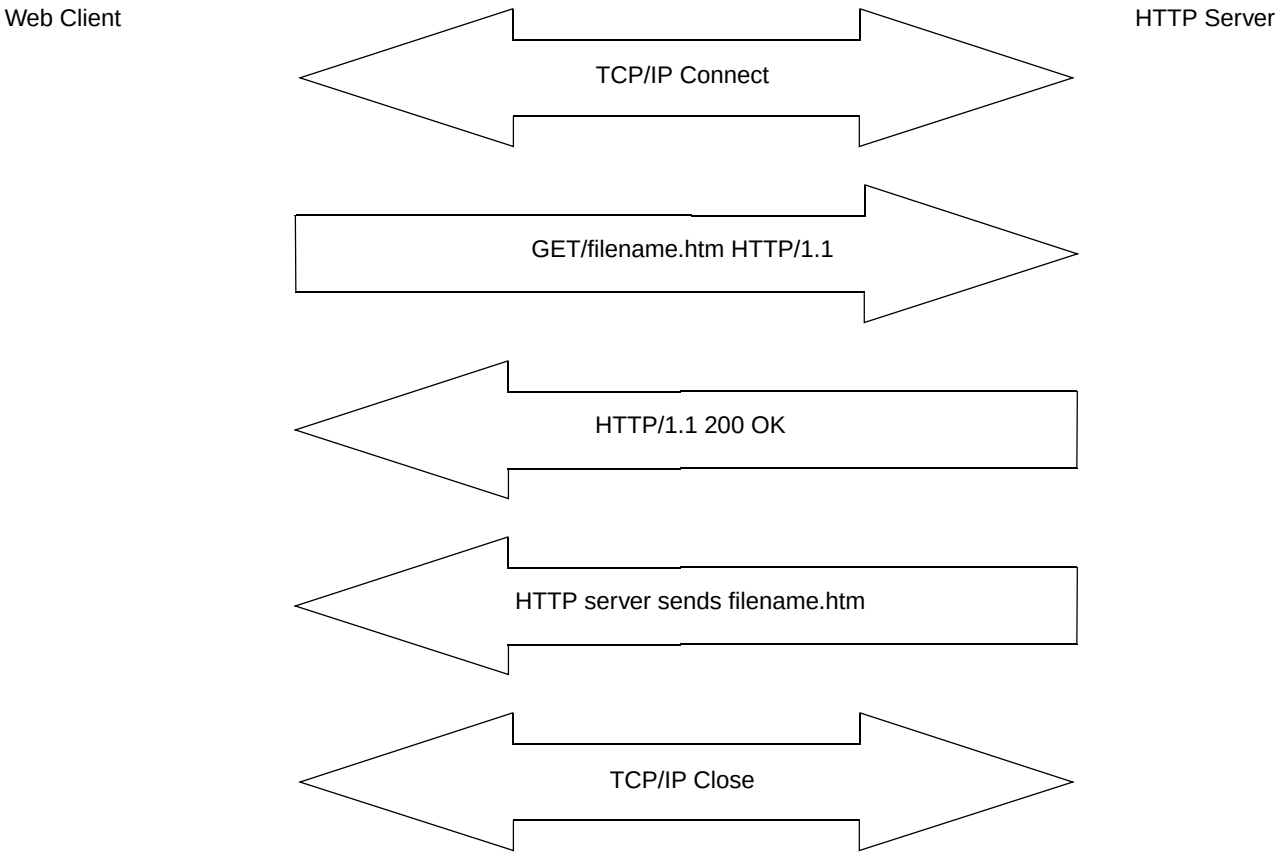


**Figure 6. A Non-Persistent HTTP Transaction**

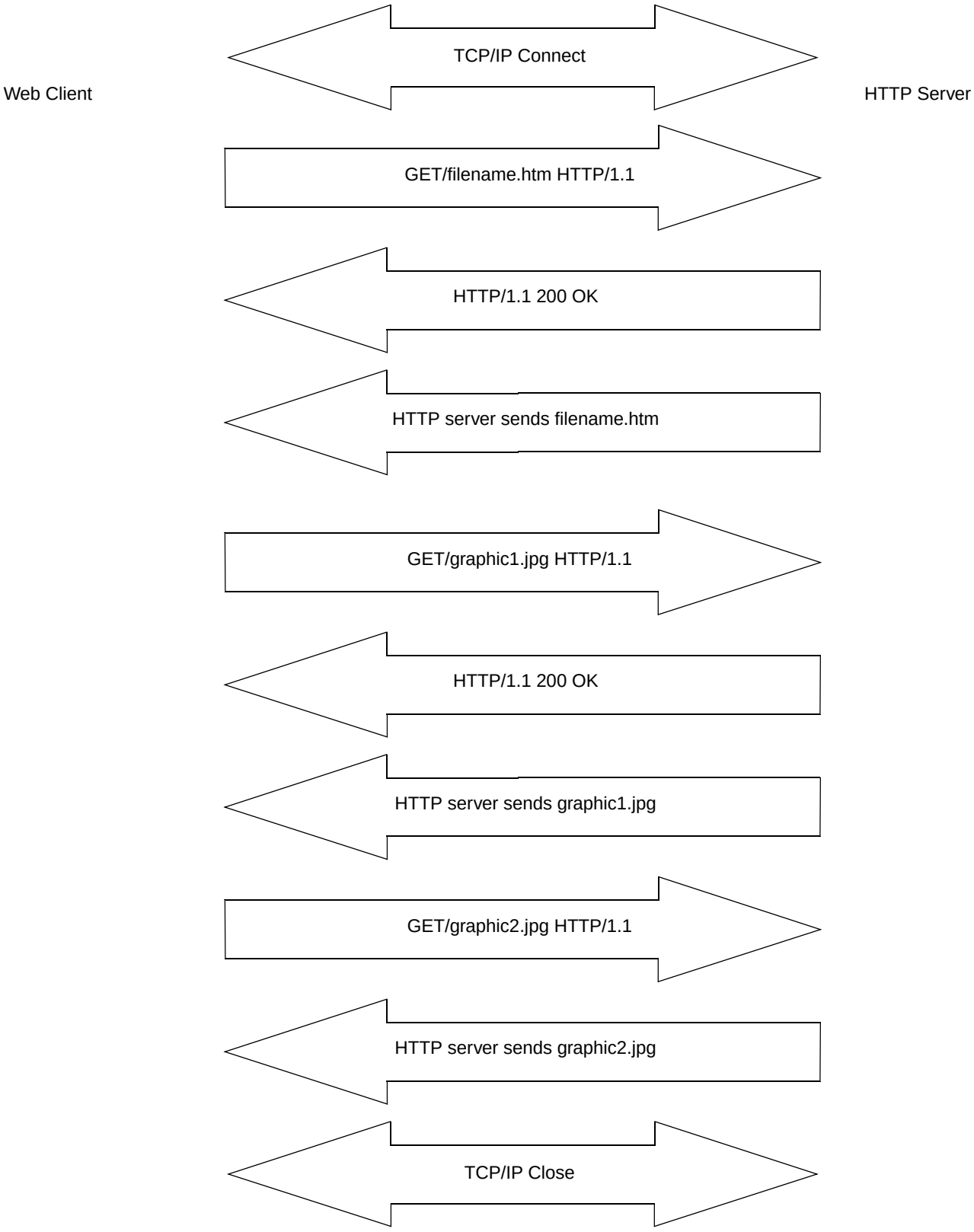


**Figure 7. A Persistent HTTP Transaction**

The state machine is implemented in the function void freescale_http_process(int session) in freescale_http.c. There are 5 valid states:

- EMG_HTTP_STATE_WAIT_FOR_HEADER,
  - The server waits for a method header (ex. GET). The method line ( GET /graphic2.jpg) is processed and parsed. The supported methods are in the header_table array:

```
{"GET",TYPE_GET},
{"EMG",TYPE_UPLOAD},
{"POST",TYPE_POST}
```

  - Depending on which method was sent, the server goes either to EMG_HTTP_STATE_SEND_FILE or EMG_HTTP_STATE_ERASE_FLASH for file upload.
- EMG_HTTP_STATE_SEND_FILE
  - The server sends data to the client. If the file was found in the FFS, the file contents are sent along with a header. If not, a pre-canned header is sent to indicate a 404 error (file not found).
  - If this is a persistent connection, go to the EMG_HTTP_STATE_WAIT_FOR_HEADER state.
  - If this is a non-persistent connection, go to the EMG_HTTP_STATE_CLOSE state. In this state, the server closes the TCP/IP connection and deletes the session.
- EMG_HTTP_STATE_CLOSE
  - The server closes the TCP/IP connection and deletes the session.
- EMG_HTTP_STATE_UPLOAD_FILE
  - The server accepts data from the client and stores it in flash.
  - Use this state for a flash image upload. The upload is handled in chunks so the other sessions of CPU cycles are not starved.
  - The next state is EMG_HTTP_STATE_CLOSE.
- EMG_HTTP_STATE_ERASE_FLASH
  - In this state the upload password is checked. If the password is valid, flash is erased.
  - After flash is erased, the next state is EMG_HTTP_STATE_UPLOAD_FILE.

# 17 RAM Usage

To keep the RAM usage to a minimum, the HTTP server uses a single dynamically allocated buffer. This buffer is passed through the HTTP protocol. The buffer is declared in void freescale_http_process(int session). Use this buffer to receive the header from the client, parse the header, send data to the client, and, when uploading, download the data from the client.

# 18 HTTP Sessions

Multiple sessions are supported via a round robin architecture. As long as the HTTP server is running, its looping through its sessions performaing whatever operation is required for each session independently. Sessions are managed by three functions in freescale_http.c and a session array.

`int freescale_http_connection(M_SOCK s)`

- This function is called when a connection is requested by the client. The socket parameter is the communication socket you use to communicate with the client. It returns 0 on success and !0 on failure.

```
int freescale_http_connection(M_SOCK s)
{
        int             i;
        int             found;

        // Walk the SESSION array looking for a invalid entry
        for(i=0, found=0; i<MAX_NUMBER_OF_SESSIONS; i++)
        {
                if(freescale_http_sessions[i].valid != VALID_EMG_SESSION)
                {
                        // Init session
                        freescale_http_sessions[i].state  =
EMG_HTTP_STATE_WAIT_FOR_HEADER;
                        freescale_http_sessions[i].socket = s;
                        freescale_http_sessions[i].valid  =
VALID_EMG_SESSION;
                        freescale_http_sessions[i].keep_alive  =
HTTP_KEEP_ALIVE_TIME;
                        found                                                   = 1;
#if HTTP_VERBOSE>2
                        printf("\nhttp %d started", i);
#endif
                        break;
                }
        }

        return(found);
}
```

`void freescale_http_delete(void)`

- Used during the HTTP server init to initialize the session arry.
- Deletes all sessions.

`void freescale_http_remove(M_SOCK so)`

- Removes a session based on its socket id.
- Goes through the session array looking for a socket id and removes the session with the same socket ID.
- Closes the socket.

The session array contains an element for each supported HTTP session. The array is declared in freescale_http_server.c.

There is one EMG_HTTP_SESSION structure per possible HTTP session.

```
typedef struct
{
        int               state;         // state of the protocol.
        int               valid;     // Data in this entry is valid
        unsigned long     keep_alive;// Connection Persistance EMG 4/14/06
        unsigned long     file_type;// File Type
        unsigned long     file_size;// Overall File size in bytes
        unsigned long     file_index;// Index for referencing data
        HEADER_TYPESheadertype;// Header type
        void                        *file_pointer;// Pointer to start data - do not change
        M_SOCK            socket;        // the open socket we are using
} EMG_HTTP_SESSION;
```

# 19 VERBOSE Support

For testing purposes, the firmware was built with multiple level of verbose. Use this macro to determine how much data is printed to studio while while the server runs. The higher levels of verbose impact server performance. A verbose level of zero disables any diagnostic messages sent to studio.

There are 6 levels of Verbose.

- Level 0 - Only fatal errors are output to the console.
- Level 1 - Future expansion
- Level 2 - File system
- Level 3 - Server + File system
- Level 4 - Server + File system + Internal variables
- Level 5 - All the above + Download progress (slows things way down).
- Level 6 - Dynamic HTML + all the above.

# 20 Dynamic HTML Handler

Dynamic HTML adds the ability to alter the web page content or the web page itself based real time data. There are many methods of supporting dynamic HTML. The HTTP server uses a token replacement mechanism. With this mechanism a token is inserted in the web page when it is created. There are many tokens, each representing a real-time user configurable variable. There are 2 types of tokens, one is replaced with numerical data, the other is replaced with ASCII text depending on a conditional result.

The file freescale_dynamic_http.c contains the firmware for supporting dynamic HTML. The function void replace_with_sensor_data(char *body_buff, UINT32 length) processes a buffer, replacing the tokens with real-time data. The buffer must contain the complete token string. The HTTP server is responsible for correctly framing the buffer to pass into the function. The buffer length effectively limits the max length of a token string.

The size of a file is calculated before the file is put into the filesystem by the PC application. Changing the file size causes an HTTP protocol violation. As a result, the search and replace algorithm in the dynamic HTML function cannot alter the size of the file. The web page creator must provide enough room around the tokens to allow for data replacement at run time. Do this by adding padding in the form of spaces (HTML ignores spaces). The padding spaces should be included within the token frame.

A token is framed by a DYNAMIC_REPLACE_TOKEN(~) or a DYNAMIC_COMPARE_TOKEN(^) at the start, and a DYNAMIC_EOS_TOKEN(;) at the end. Spaces are added before the (;) for padding.

An example of dynamic HTML frames with padding appears below.

```
<td>~03H    ;</td>
<td>~03D    ;</td>
<td><FONT COLOR=^03>0800|"RED"|"BLUE"|;>Analog Channel 0 (pot)</td>
```

# 21 The VAR array

The VAR array contains the real-time variables referenced by the dynamic HTML. Following the start of frame token (^) or (~) is a two digit decimal number representing an index in the VAR array. Use this index to reference a real-time variable in the VAR array. With only two digits, the VAR array can contain only 99 elements. The firmware can be easily modified to support larger VAR arrays.

The function void collect_sensor_data(void) is called by the HTTP server when a GET method is received. This function reads the real-time data and stores it in the VAR array. This is done in the function void collect_sensor_data(void) in freescale_dynamic_http.c. Each element in the VAR array consists of two fields, a long work (32 bits) for the data, and a byte for the valid data flag. If the valid data flag is not one, the token string is replaced with spaces. Otherwise the token string is replaced with the hex or decimal value in the data field.

**The VAR array**

```
void collect_sensor_data(void)
{
        html_vars[0]      = MCF_GPIO_PORTTC;
        html_vars_flags[0] = 1;

        html_vars[1]      = poll_switches();
        html_vars_flags[1] = 1;

        html_vars[2]      = hit_counter++;
        html_vars_flags[2] = 1;

        html_vars[3]      = read_AD(0);
        html_vars_flags[3] = 1;

        html_vars[4]      = read_AD(1);;
        html_vars_flags[4] = 1;

        html_vars[5]      = 0;
        html_vars_flags[5] = 1;

        html_vars[6]      = 0;
        html_vars_flags[6] = 1;

        html_vars[7]      = read_AD(4);
        html_vars_flags[7] = 1;

        html_vars[8]      = read_AD(5);
        html_vars_flags[8] = 1;

        html_vars[9]      = read_AD(6);
        html_vars_flags[9] = 1;

        html_vars[10]     = 0;
        html_vars_flags[10] = 1;

        html_vars[11]     = (MCF_RTC_HOURMIN & 0x00001F00)>>8;
        html_vars_flags[11] = 1;

        html_vars[12]     = MCF_RTC_HOURMIN & 0x0000003F;
        html_vars_flags[12] = 1;

        html_vars[13]     = MCF_RTC_SECONDS & 0x0000003F;
        html_vars_flags[13] = 1;
}
```

# 22 URL Form Request Handler

The HTTP server primarily handles output data. This makes it a perfect protocol for monitoring a site from a remote location. URL encoding provides some control to the server. Using URL encoding, a web client can send commands to the web server to perform specific functions, such as controlling an LED on the web server board (see labs).

URL encoding provides a mechanism for sending command and numeric data to the web server. It is referred to as URL encoding because the command and numeric data are encoded in the requested file name of the GET method. The (?) symbol tells the server that the filename field in the GET method is not a filename, but a form command.

An encoded form example is index.htm?led=LED1_TOGGLE, where:

- index.htm is the web page to send after the form is executed
- led is the form variable. LED1_TOGGLE is the value assigned to the variable.
- The '?' tells the server that the string that follows is a form assignment. Multiple form assignments can reside one after another seperated by '?'. The web server includes a form assignment parser with functions to manage the led variable.

## 22.1 Form Handling Code

Filename fields containing forms are detected in the function void pre_process_filename(char *buffer) in the module freescale_http.c. This function scans through the filename looking for (?). If a (?) is found, the string is framed and passed to the function void process_form(char *form_string), also in the freescale_http.c module. This function calls the appropriate handler for the form variable.

The const FORM_STRUCTURE forms[] array in the module frescale_static_ffs_utils.c associates a variable name string to a variable handling function.

```
const FORM_STRUCTURE forms[] =
{       // variable name string    pointer to handler function
{"led",                            form_led_function},
        {"serial",                          form_serial_function},
        {"var",                             form_var_function},
        {"",                                        form_led_function}};
```

To add additional form handlers, add the new variable name string to the structure and write a handler. The handler is passed a string with everything after the equal '=' sign.

## 22.2 Sample Form Handlers

```
void form_serial_function(char *data)
{
        printf("%s", data);
}
void form_led_function(char *data)
```

- Turns a LED on the demo board on or off

```
void form_var_function(char *data)
```

- Used to modify data in the VAR array.
- var=II_decimal or II_off
  - where:
    - II is the decimal index of the VAR to modify
    - off clears the VARS data valid flag.
    - Else, the user provides a decimal value to replace the VAR data

# 23 Conclusion

The HTTP server is a very flexible piece of software designed to offer the customer an 80% solution. The design goal is to provide the customer a basis for developing his own project specific web page-enabled application. The code provides examples designed as a general stepping stone.

The HTTP server for ColdFire devices is an HTTP1.0 compliant web server with support for multiple sessions and persistent connection (HTTP1.1). The firmware includes support for two writable and one read only flash file systems. Support for external serial flash is also provided. The flash file system supports directories and sub-directories along with long filenames. A protocol and tools are provided to support the ability to upload new web pages over Ethernet. Port 80 is used for the uploads to avoid issues with firewalls. The upload protocol supports a 32 byte password for web page security. The GET method is currently supported, but a mechanism is provided to add support for additional methods. Dynamic HTML is supported using replace and conditional tokens. Forms are supported, and easily expandable. A VAR form is provided for modifying dynamic data from the web client. A serial form is provided to allow a device connected to the web server’s serial port to be directly controlled from a web browser.

**How to Reach Us:**

**Home Page:**
www.freescale.com

**Web Support:**
http://www.freescale.com/support

**USA/Europe or Locations Not Listed:**
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

**Europe, Middle East, and Africa:**
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

**Japan:**
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

**Asia/Pacific:**
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3455
Rev. 0
04/2007



freescale™
semiconductor