# HTTP and Web Servers Technical Reference

Interniche Legacy Document

Version 1.00

**Date:** 11-May-2017 13:49

# Table of Contents

# 1 Introduction

This document is a "how-to" manual to enable an experienced embedded systems programmer with a conceptual understanding of what an embedded web server does, to port the InterNiche Internet Services to his embedded system. The porting process can take as little as a few hours depending on which of the required resources your system already has and the complexity of your HTML pages. When you finish, your system will be able to serve HTML text files, forms, graphics, and all the other features of the World-Wide-Web. Some HTML knowledge is needed, but everything you need to know about HTTP to port the server is in this document. If you have no previous experience with HTML, getting a good book on it and writing a few practice web pages is suggested.

## 1.1 Terms and Conventions

In this document, the term, "InterNiche Internet Services", refers to the combination of InterNiche modules: HTTP Server, Webserver, VFS, and Menu Commands, as well as the InterNiche sockets and CLI interfaces. These are shown with bold type labels in the InterNiche Internet Services Architecture diagram in the Architecture section of this document. The terms, "InterNiche Internet servers", or just, "Internet servers", refer to the HTTP Server and Webserver code as ported to an embedded system. "System" refers to your embedded system. "Sockets" refers to the TCP API developed for UNIX at U.C. Berkeley. The server is delivered with example implementation notes for Sockets since many embedded systems already have sockets. A copy of the Sockets API documentation is available from InterNiche upon request. A "user" or "porting engineer" usually refers to the engineer who is porting the server. An "end user" refers to the person at the browser who ultimately ends up using the "user's" product, and "Web" refers to the World-Wide-Web

Names of files, C structures, and C routines are displayed as follows: `c_routine()`

Source from C programs or HTML files is displayed in these boxes:

```
/* C source file - yet another hello program. */
main() {
   printf("hello, world.\n");
}
```

## 1.2 The InterNiche Difference

The code for InterNiche Internet Services is written in the C programming language, conforming to the ANSI standard and compatible with C++. It has debugging macros to cooperate with source level debuggers and In-Circuit Emulators when debugging.

InterNiche Internet Services are different from most others in that they were designed from scratch for embedded systems. Smaller embedded systems, such as DVDs, cell phones and controllers, often have tight size, weight and cost restrictions that limit the amount of CPU power and memory they can have; thus every added feature, including Web services, must be weighed against the system resources it will require. For this reason, the design goals of our servers are, in some ways, drastically different from those of conventional UNIX web servers. UNIX servers tend to optimize speed at the cost of size and complexity; whereas we favor smallness and simplicity. Here are some examples of how we took a different approach:

- InterNiche system calls (e.g., memory, file system, and Socket calls) use the standard APIs that are in widespread use throughout the embedded systems industry. However, InterNiche's system calls have been written to use small amounts of system resources and to be easily mapped to a very wide variety of specific environments. All the mappings of typedefs, local MACROS, including of system header files, etc., are done in a few porting include files that come with the C source files.
- Memory Requirements: A primary consideration of every aspect of the design was minimization of the required amount of firmware storage, static RAM, and dynamic RAM.
- Multitasking: Since many embedded systems have limited multitasking and user shell capabilities, the handling of a connection by the UNIX-like `fork()` and `exec()` may not be practical. Our server handles multiple connections through a linked list of dynamic memory structures containing state information (usually one per connection). A single regular timer tick loops through the list and gives each connection CPU time to do some work -- usually pump some data into a socket.
- File System: Many embedded systems have no hard disk or file system capabilities. Even when they do, the file system on the target embedded system is often different from the file system on the development platform where programmers write and debug their code. InterNiche provides a lightweight Virtual File System (VFS), which, in cooperation with the VFS Compiler, handles keeping HTML data in ROM or RAM and supports compression and decompression. VFS uses APIs that readily map to the common APIs standard on most file systems. If the code is written using the VFS APIs, these can usually be mapped to another file system through simple defines and typedefs in a porting include files. However, even on platforms that already have a file system, it is often more convenient to simply use the VFS file system to store all web file.

If your system communicates with other systems using the TCP/IP protocol family, it probably has the resources to support the InterNiche Internet servers. Several optional features of these servers (e.g., support for HTTPS, MD5, File uploading, etc) are implemented in code within `ifdef` blocks. Each of the optional features will only become part of the compiled code if its associated `define` is enabled.

## 1.3 The Windows Example Programs

InterNiche Internet Services are delivered with example code that provides many examples of how to write both client and server application code to take advantage of the wide variety of functions provided by InterNiche Internet Services. The example application assumes that it will be running on one of InterNiche's own TCP/IP stacks, though it was designed to be portable to other communication implementations. We encourage embedded system engineers to play with these, as a learning activity, before starting their actual port.

The InterNiche VFS Compiler was designed to greatly facilitate the integration of the files for your web pages into the VFS file system. While it is optional, it is strongly recommended. The VFS Compiler can be invoked from within a standard makefile. It takes any file you would normally put on a web server, including binary files, and "compiles" them into C structures that become VFS files in your embedded server. All the HTML files are normally found in a single directory and they compile into a single, easily linkable library.

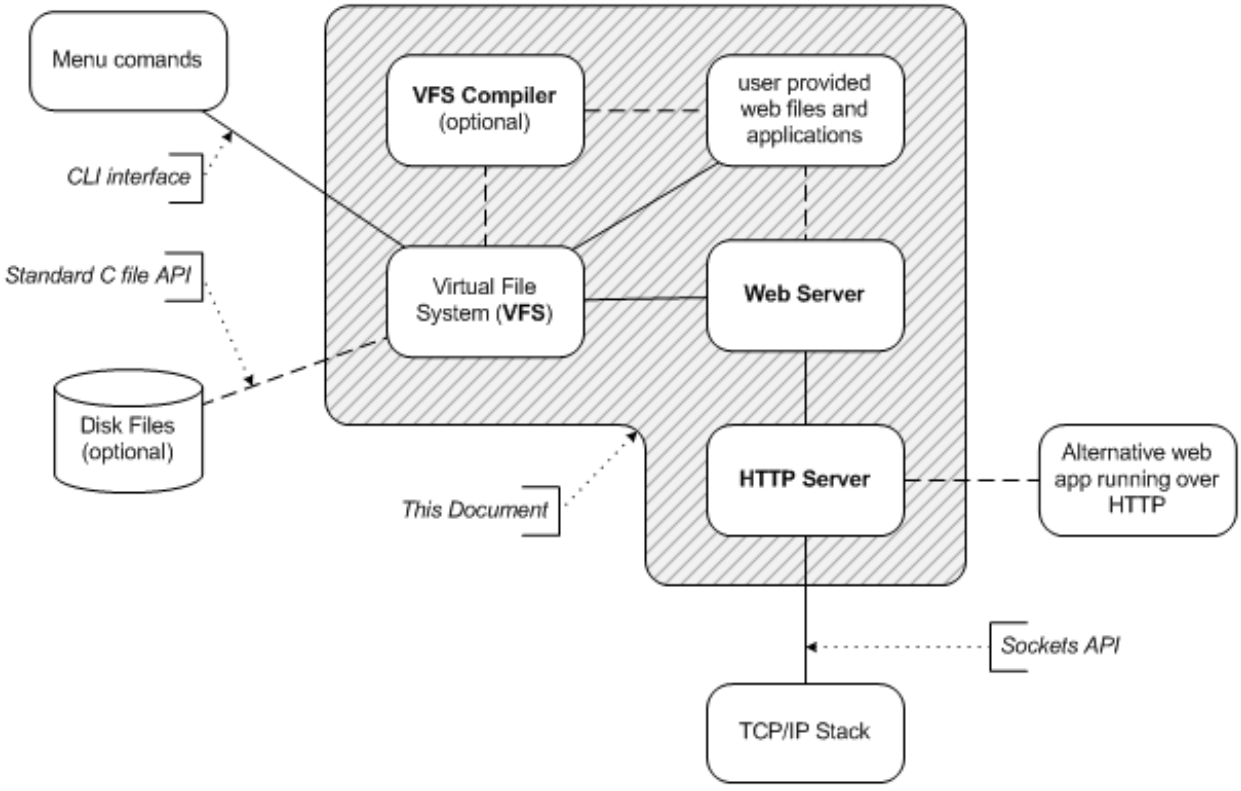
The example HTML files can be modified or extended, and new HTML files can be created until the "look and feel" is right. Any new files would be added to the demonstration directory's makefile for compiling into the VFS. This allows the porting engineer to implement and step through actual executions of the sample scripts and CGI routines

# 2 InterNiche Internet Services Architecture

If you are already familiar with the server code and want to start porting right away, you can go to Porting Step by Step. Otherwise we recommend you browse through the next few chapters to get a sense of what facilities are available in your server and how to make the best use of them.

## 2.1 Overview

The InterNiche Internet servers are divided into two separate modules, the HTTP Server and the Webserver. As the names suggest, the HTTP server handles HTTP protocol (transport) issues, while the Webserver is the standard, but optional, application, that makes use of the HTTP to provide web server services. The Webserver will always run over HTTP, but other applications (e.g., RTSP), may run over HTTP independently of the Webserver. The diagram shows the relationships between the main components of InterNiche Internet Services. The following paragraphs give a brief description of each of these components.




**TCP/IP and the sockets API**

The HTTP Server uses a subset of the standard sockets API to send and receive messages from the TCP /IP stack. The HTTP Server does not require the use of InterNiche's version of sockets and TCP/IP. Any reliable transport method could be used as long as it provides an implementation of basic socket-like functions that can be mapped to the HTTP Server's sockets interface. The HTTP Server can also be used with InterNiche's "NicheLite" version of TCP/IP

**HTTP Server**

The HTTP Server listens for, and accepts connection requests on the HTTP port. It then reads HTTP requests over that connection (socket), extracts the key information into a data structure that it places on a queue for the Webserver. It provides routines that the Webserver uses to create HTTP response headers and to send HTTP response messages. Finally, when the Webserver signals that the request has been completed, it either frees any allocated structures and closes the socket, or it reinitializes the structures so they can be reused for a new request over the existing socket.

**Webserver**

The Webserver processes requests that have been placed on its queue by the HTTP Server. It calls the appropriate internal functions and web applications to perform the requested function or to deliver the requested pages. It uses HTTP Server functions to format the HTTP response header and it calls HTTP functions to either send the response data or an error message.

**Virtual File System (VFS)**

The InterNiche VFS is a flat (non-hierarchical) file system, in which the files are stored in the target system memory and are implemented as a linked list of structures. Each structure holds key information such as flags and the file's size, and each has a pointer to a buffer that contains the associated file's contents. The Webserver uses VFS to access all files and CGI commands that it uses. Although multiple CGI routines are generally stored in a single file, the name of each CGI routine is associated with its own VFS file structure and the Webserver accesses each just as if it were a file.

**VFS Compiler (optional, but highly recommended)**

The VFS Compiler takes as input a list of files and CGI commands. Each line in the list of files can have one or more parameters that indicate such things as the required authentication or whether the line represents a CGI command. For each file or CGI command, the VFS Compiler creates a VFS structure, initializes its fields, and adds it an array of VFS structures. It then "compiles" the files into strings that will be the actual VFS files. The VFS Compiler is designed to be invoked from a makefile as part of the build process. Its output is .h file and one or more C files that are linked into the target executable. The VFS Compiler also provides "string compression," which can be used to significantly compress VFS files.

**Menu Commands**

Most InterNiche modules provide a set of commands that can be used to control the module or obtain information about the module. A modules set of commands can be accessed as a menu from a command line prompt. Menu commands can also be embedded in files so that when the file is requested via HTTP, the output of the menu command will be added to the file at the point were the commands was found.

**User Provided Applications**

The Webserver provides the means for the porting engineer to supply files that can be accessed over the Web, as well as web applications and CGI functions that can be executed via an HTTP request. The Webserver will deliver any files or results provided by the web applications.

## 2.2 The HTTP Server

The bulk of the HTTP Server code is in the file **http.c**, which the porting engineer should not need to change

The HTTP Server handles the following jobs:

- Parses incoming HTTP requests from a newly connected socket and places them on the in-queue for the web application (usually the Webserver).
- Allocates the required http data structures and ensures that when the connection is terminated, any allocated structures are freed and all related files are closed.
- Accepts data over connections that use IP Version 4 or IP Version 6 or NicheLite.
- Keeps clickable bitmap file info encoded in small tables and resolves HTTP requests based on clicked bitmaps.
- Supports both HTTP1.0 and HTTP1.1, including persistent connections, pipelining and data chunking.
- Provides routines that can be used by the web application to build an HTTP reply header and to send messages over HTTP.

For each new connection, the HTTP Server creates an httpd structure (defined in `httpd.h`) and adds it to the HTTP queue. A single regular time tick loops through the queue looking for new requests, additional messages for existing requests, and completions states for existing requests.

### 2.2.1 Persistent connections, pipelining and data chunking

The HTTP Server is capable of dealing with clients that use HTTP 1.1 or the legacy HTTP 1.0 protocol.

The original HTTP 1.0 specification only supported non-persistent connections. With non-persistent connections, the HTTP Server informs the client that it has completed sending data for the transaction by closing the connection.

The HTTP 1.1 specification provided a standard for persistent connections. If the connection is a persistent connection, then it can be reused for the subsequent requests related to the original request. With persistent connections, it is normally the browser that determines when the connection should be terminated. In the InterNiche implementation, incoming and outgoing data buffers and most allocated structures and substructures are retained throughout the persistent connection, and the HTTP Server ensures that at the beginning of each request, data fields are reinitialized as necessary.

If multiple requests are "pipelined" on a persistent connection, they are processed in the order in which they are received.

With persistent connections, there are two ways to inform the client that the server has completed sending the requested data. If the data to be returned is a file or text message of fixed length, the server uses the "`Content-length`" parameter to tell the peer the length of the response data.

When the Web application dynamically builds the return data, the length of the response is not known in advance and "data chunking" is used. With data chunking, as the application obtains or creates each portion of the data needed to support the request, it formats the data and then calls HTTP to send that piece of the response to the browser. HTTP adds a length field to each piece of data, before it sends it. When all data has been sent, the application or the Webserver calls sets a last-chunk flag and HTTP will the send the "lastchunk" string (a zero followed by two sets of carriage return-line feed characters.). This signals the browser that all data for the response has been sent.

In InterNiche Internet Services code, support for persistent connections, pipelining and data chunking is enabled by defining `HT_PERSISTENT` in `ipport.h`.

Systems utilizing InterNiche's menu system (controlled by the define `INCLUDE_CLI`), can dynamically enable or disable support for persistent connections at runtime using the "`http config`" command.

## 2.2.2 Authentication and Security

The HTTP 1.0 specification also specified a user/password authentication scheme, which is generally referred to as "Basic" authentication. Basic authentication provides only minimal security. RCF 2617 specifies a much more secure authentication scheme that is generally called "Digest" authentication. Both of these authentication schemes can be used with the Secure Sockets Layer (SSL) protocol in order to provide a much higher level of security.

Access to each VFS file and each CGI command can be controlled separately, because each is associated with a separate `vfs_file` structure that contains a field indicating the authorization type required to access that file or command.

**Basic Authentication**

The advantages of Basic authentication are that it is simple, it uses few system resources, and it is supported by all popular web browsers. However, because the user name and password are sent in the clear, it is rarely used on publicly accessible web sites. It is still used on small, private, embedded systems or systems where only minimal security is sufficient

When end users on a browser attempt access to a protected page, the browser asks them to enter a user name and password. The correct information must be entered before the server will provide the secure page.

Internally, several steps happen when a user first tries to access a protected page. First, the browser does a standard HTTP `GET` or `POST` request for the HTML page. The server notes that the page is protected and scans the HTTP header for an appropriate Authorization field. If this field is missing, as it usually is on the first request, the server sends an authorization failure error back to the browser. The browser then displays a popup window to the user asking for the name and password. The user types this in and the browser will encode the username/passed into a sequence of base-64 characters (This process is called, "uuencoding"). The browser re-sends the HTTP `GET` request with the Authentication field (the uuencoded password info) appended to the HTTP request header. The server converts the uuencoded string back into plain text, checks the name and password, and, if it is OK, returns the protected HTML page.

It should be noted that uuencoding is NOT done to provide security. Passwords may contain special characters that could break the HTTP protocol. Uuencoding prevents this by substituting other characters in place of those that might cause problems. There is no secret involved in converting uuencoding back into plain text.

**Digest Authentication**

The InterNiche HTTP Server also provides a more sophisticated authorization scheme, Digest Authentication. Digest Authentication also uses the user/password scheme and most of what is described above for Basic Authentication also applies to Digest Authentication. The difference is that the username and password are now encrypted and several additional fields are added to improve security.

If a requested file is marked for Digest Authentication, the HTTP server will send a response that includes the authentication options it supports and a special token. The client's response indicates the options it has selected. The response is encrypted in a manner that proves the client or system knows the password. If the server is satisfied with the response, it provides the requested file(s).

It should be noted that while Digest Authentication is secure enough for most real-world applications, it has vulnerabilities. It is not as good as state of the art public and private key encryption systems.

For ordinary business office applications where the security requirements are fairly straightforward, the porting engineer simply needs to indicate which files are protected with the user/password or Digest security and provide a mechanism for checking password info. The implementation details of both of these are covered in Porting Step by Step.

## 2.2.3 HTTP over SSL

The HTTP Server is optionally available with support for the Secure Sockets Layer (SSL). This is referred to as “https” throughout this document. SSL is a protocol layer which may be placed between TCP/IP and an application protocol layer such as HTTP. SSL provides secure communication between client and server over the internet. If `INCLUDE_HTTPS` is defined in `ipport.h`, the HTTP Server will automatically use SSL to establish and utilize secure connections. If you are using InterNiche's SSL product, then normally no porting effort will be required within the HTTP layer for HTTPS. The implementation details for InterNiche's SSL product are covered in the *SSL Technical Reference manual*.

## 2.3 The Webserver

The Webserver processes requests that have been placed on its queue by the HTTP Server. It calls the appropriate internal routines and web applications to perform the requested functions or deliver the requested pages.

The Webserver handles two request types, GET and POST. There are many similarities between the two. With the InterNiche Webserver, everything that can be done with a GET could be done with a POST. However, the reverse is not true; there are many tasks that can be done with a POST that cannot be done with a GET. A GET is designed for simple requests that take only a few simple parameters. A POST is normally used to send the name/value pairs entered into a form. Only POST messages should used to send requests with more than a simple set of parameters.

The browser sends requests to:

1. Obtain a file (Normally done with GET)
2. Request that one or more functions be performed on the server (GET or POST depending on whether form data is sent)
3. Upload a file to the server (POST only)

### 2.3.1 Embedded include files and commands

If the browser requests a simple file, the Webserver will generate an HTTP response header containing the length of the file that will be returned. It will then read the file and return it to the browser by writing to the socket used for the Internet. There will be no change on the server.

In the InterNiche implementation, a Web application file with the extension "`.iws`" or "`.iwx`" is called a "script" file. When a script file is requested, the Webserver will parse the file looking for escape sequences that contain embedded commands or that list one or more other files to include. An escape sequence within a file is always begins with the string, "<#", and ends with the string, "#>".

When the Webserver encounters this escape sequence, it will:

1. Send the file data prior to the escape sequence,
2. Parse out the include file or embedded command, including any parameters,
3. Execute the embedded command or read the include file,
4. Send any data output as a result of the command or include file,
5. Continue reading and outputting the file named in the request until it finds another escape sequence or the end of the file.

If the file extension is "`.iws`", the Webserver will set the file type parameter in the response header to "`text/html`". If the file extension is "`.iwx`", it will set the file type parameter to "text/xml".

A single file may contain any number of embedded commands or include files, and each include file (assuming it has an "`.iws`" or "`.iwx`" extension) could also contain embedded commands and include files. The nesting of include files is only limited by configuration parameters or available memory.

In order to include one file within another, the included file name must be within an escape sequence and the file name must be preceded by the word "include" and followed by a semicolon. For example: <# `include footer.htm; #>`. Note that there is only white space (space, horizontal tab, new-line, vertical tab, and form-feed) between the word "include" and the file name. There are no quotes in the escape sequence. The file to be included must be one that is known to the VFS file system.

As an example of how this might be used, a set of Web pages could all include, at the appropriate spots, a . gif file to display the company logo, a file containing HTML to display the page header, and another file with HTML for the page footer.

In the InterNiche implementation, a single escape sequence could contain one or more commands and/or include files. Each command or include file must be followed by a semicolon. The semicolon separates multiple commands, but it must always be used, even if there is only one command or include file in the escape sequence.

An embedded command must be either a:

1. a menu command known to the menu system,
2. a built-in command ("`include`" or "`echo`"),
3. the name of a user defined CGI routine known by VFS (discussed below),
4. a script-a set of one or more commands, include files, or names of CGI routines within a single escape sequence.

Commands can be embedded within a file simply by putting the command within an escape sequence at the point within the file where you want the output from the command to be displayed. For example:

```
<# cticks; #>
```

Note again that the command is always followed by a semicolon with no quotes around the command name.

An embedded command can include parameters. A command begins with the first non-white space character either after a begin-escape sequence or after the last semicolon within an escape sequence. A command ends with a semicolon. The Webserver uses VFS to find the appropriate routine to handle the command. It then calls the routine passing the entire string from the beginning of the command to the last character before the semicolon.

Within an escape sequence, white space characters are ignored (space, horizontal tab, new-line, vertical tab, and form-feed). For readability, any number of white space characters may be used anywhere within the escape sequence. For example:

```
<#
currtime;
mycommand  -x -f filename -n nvalue;
include myfile.htm;
#>
```

## 2.3.2 Built-in Commands

Currently there are two built-in commands: "include" and "echo".

**Include**

The "include" command is simply the "include" part of

```
<# include filename; #>
```

The "include" command is treated almost exactly like any other command. The difference is that it is implemented within Webserver code, and it is not part of the menu system or VFS.

**Echo**

The "echo" command is used to send text to the browser. The text to be sent is delineated by either single or double quotation marks (whichever is found first following the echo command.) The text within the quotation marks can contain any characters or escape sequences acceptable to the browser, except quotation marks. In particular, the text can contain semicolons. This permits the use of special HTML escape sequences such as " ", which is used to code a space. For example:

```
<# echo "The current time is "; currtime; echo " GMT";  #>
```

The echo routine simply forwards all characters between the beginning and the ending quotation marks. In the above example, the browser will display something like:

```
The current time is 10:14:11 GMT.
```

## 2.3.3 Menu Commands

The InterNiche server code includes a Command Line Interface (CLI), which is common to all InterNiche Modules. The CLI provides access to the InterNiche menus. During initialization, most InterNiche modules use the CLI interface to provide a set of commands (a Menu) that can be used to control features in that module or to obtain information about the module. Each module's Menu becomes a submenu with the InterNiche menu system. Each menu entry represents a routine that will be executed when the entry is selected. (See the **NicheStack Reference Manual** for more information).

Assuming the proper permissions, commands can be executed locally from the command prompt or remotely via telnet.

It is also possible for the porting engineer to set up specific menu commands for execution via the Internet. A simple way to do this would be to embed the command with an escape sequence in a script file. The script file could contain just enough HTML or XML surrounding the command to properly display its output as a web page.

It is also possible for the porting engineer to write a routine that allows a menu command to accept remote parameters. This will be described below in the porting section.

### 2.3.4 CGI Commands

The term CGI refers to the name of a routine or command that can be directly executed via a GET or POST request. As described above, both InterNiche provided Menu commands, as well as user provided commands, can be executed remotely via the Webserver by setting up script files (files that contain embedded commands and include files.) The CGI mechanism allows a GET or a POST to directly request the execution of a command. With the exception of the two built-in commands, "include" and "echo", the porting engineer must provide the routines that will enable the commands to be directly executed via the Webserver. Like Menu commands, a CGI routine may also be executed from an escape sequence in a .iws or .iwx file.

In the most common use of a CGI, the end-user on a browser first requests an HTML page from the server, which contains a form. The user fills in the form data and submits the form. The code that displayed the form also specifies the name of the routine that should be executed when the form is submitted. Normally, a POST request is used, and the form data is appended to the file name as a string of uuencoded text. It is also possible to call a CGI with a GET message. In this case, any parameters are attached to the request as a question mark separated list.

On a conventional UNIX web server, the server then executes a process (indicated by the file name in the form) and passes it the form data. The process executes a C program or Perl script that reads the form data and returns HTML text to be sent back to the browser.

The InterNiche HTTP Server does not assume the existence of a shell-like exec capability. The file name /command indicated by the GET or POST reply to a form is looked up in the VFS. For CGI routines, the VFS structure for the file will point to the CGI routine to be executed. This CGI routine will have been supplied by the porting engineer as part of the porting effort.

The HTTP Server will parse all form/parameter data from the GET or POST request and store this as name /value pairs in a form structure (struct formdata in httpd.h). The Webserver will subsequently call the CGI routine, passing an httpd structure that contains all information needed by the CGI routine, including the socket used for the transaction.

Porting engineers can use this routine to do whatever they want with the passed data. The routine can change parameters on the server, write any desired HTML text directly over the socket, or it can ask the Webserver to return a file. This returned file could be a script file, which the Webserver will parse and handle just as if it had receive the file request from GET or a POST. With some care, it is also possible for a CGI routine to block or loop indefinitely sending occasional updates to the browser. See the CGI Routines section for more details.

## 2.4 The Virtual File System

The InterNiche server uses the VFS for access to all files and CGI commands that it needs. VFS contains all of the file IO routines called by the Webserver. VFS is a flat file system in which the files are stored in the target system's memory.

Each VFS file is associated with a struct `vfs_file` (`vfsfiles.h`). VFS also stores a separate `vfs_file` structure for each CGI function. This allows the Webserver to obtain a pointer to the CGI routine as if it were a file. Among other things, `vfs_file` structure contains:

- The real size of the file
- Its compressed size
- A pointer the buffer containing the file's contents
- A flag field
- If the structure represents a CGI function, it contains a pointer to that function

The flags field contains a number of flags that are important to HTTP and the Webserver:

- Whether the file has been compressed
- Authorization type for the file
- Whether the file can be cached by an intermediate Web server
- Whether the file structure is used to display a variable

Some of the advantages of having all Web files and CGI functions in the VFS system are:

- Provides fast access to files
- Provides direct access to CGI functions
- Handles decompression of compressed files "on-the-fly" during read operations
- Provides the security of having file data in ROM

See the VFS Section of the *NicheStack Reference Manual* for a much more extensive description of VFS.

### 2.4.1 Layering VFS on a Pre-Existing File System

Normally the system programmer will never have to worry about the interface to the VFS, but if you should decide to modify it you will notice that the VFS calls are directly analogous to standard C library calls. Setting the `ifdef` above actually defines macros which map the VFS calls directly to the library calls. The only difference is that a `VFILE` pointer (to a VFS structure) is used instead of a system FILE pointer. The VFS calls and their standard equivalents are listed below:

| VFS Call | Standard | Notes |
|---|---|---|
| vfopen() | fopen() | Ignores mode, all files assumed read-only |
| vfread() | fread() | |
| vfseek() | fseek() | Only seeks to beginning or end of file |
| vfclose() | fclose() | |
| vgetc() | getc() | |

The HTTP server and the Webserver can use the VFS file system alone or layered over a native file system. All file system calls are made via one of the VFS routines listed in the table above.

Support for a native OS is enabled by a define in the ipport.h file:

```
#define HT_LOCALFS   1     /* TRUE if there is a local file system */
```

If `HT_LOCALFS` is enabled, the Webserver will expect your embedded system to support the standard file IO routines listed.

See the VFS Section of the *NicheStack Reference Manual* for more details.

### 2.4.2 File Compression

When using the optional, though highly recommended, **VFS Compiler**, HTML file data is compresses as it is encoded into C files. This simple compression, based on frequently occurring text in the HTML files, is herein called "Tag Compression" (as opposed to the generic compression applied to HTML files) since it is based on HTML tags and patterns. Tag compression is designed to be very fast to decompress, and decompression is implemented with only a few lines of code. Tag Compressed files are quite amenable to further compression by standard techniques. Tag compression is described in much more detail in the VFS Compiler User Guide.

## 2.5 System Interfaces

System interfaces are the APIs that make system resources available to the server. These fall into two general categories: Memory access (`malloc` and `free`) and Network access (sockets). On systems with native file systems, the file IO calls also fall into this category. On many embedded systems all of the required APIs already exist, and all you need to do is include the system's include files (e.g. `memory.h`, `sockets.h`) in `ipport.h`.

On systems where the existing memory and/or TCP APIs do not match those used by the server, the porting engineer will have to write a glue layer that implements the expected functionality of each of the servers' required routines. The section Provide the System Routines lists the required routines and their semantics.

# 3 General Functions

Name

`ht_sendbuf()`

Syntax

```
int ht_sendbuf(struct HttpSocketInfo *si, char *buf, int length);
```

Description

This routine is called whenever data needs to be sent by the HTTP Server. Depending on flags on structure HttpSocketInfo, it calls `sys_send()` for non-secure HTTP connection and calls `ssl_send()` for secure HTTP connection.

If data chunking is being used in reply to a HTTP request, then `http_sendbuf()` does the appropriate formatting. That is it sends the length fields, followed by CRLF, followed by data, followed by CRLF. If buff is `NULL` and length is 0, then it means that all data has been sent and then the last-chunk needs to be sent (0 followed by CRLF CRL).

When applications like CGI, SSI, Server Push, need to send data, they can call `ht_sendbuf()` with the first argument being "`&hp->si`". These application functions are called with "struct httpd * hp" and the address of "si" field needs to be passed to `ht_sendbuf()`.

Returns

Returns bytes after writing from secure or non-secure HTTP connection.

The extra bytes sent for chunking are not reported in the return value. That is, the maximum return value is equal to the received length.

# 4 System Requirements

If your system runs the TCP protocol (as in TCP/IP), then it almost certainly has the requisite hardware and system software for the InterNiche Internet Services. Here is a brief list of specific requirements, each of which is then explained in more depth.

- TCP and Sockets (or similar) Interface
- Static Memory
- Dynamic Memory - `malloc()` and `free()`
- Periodic Clock Tick

## 4.1 TCP and Sockets Interface

Browsers communicate with the HTTP Server via the TCP protocol and the most common API for TCP in embedded systems is Sockets. The HTTP Server code assumes that your embedded system already has TCP and, further, that the embedded system has some kind of API to get at the TCP connection services. If it doesn't have TCP, InterNiche provides portable TCP layers as a separate product. If your API is Sockets, then the code in the HTTP Server package will map directly to your Sockets API and the bulk of your porting work is already done.

If your API is not sockets, then you will need to provide the ability to do a TCP LISTEN, to accept incoming connections and to read and write to TCP connections. Details of how to do this are provided in Porting Step by Step.

## 4.2 The Clock Tick

Not all HTTP requests can be completely resolved at the time the port software calls `http_connection()`. For example, a `GET` on a file larger than the TCP window size may block since large files may not fit into the system's available socket buffers. Thus the server needs to periodically get some CPU cycles to do the next chunk of work and to clean up when the connection's request is finally finished.

Although we call this a clock tick, it is usually not called on every clock interrupt. The HTTP Server only needs these cycles when connections are open, and even then calling it more than a few dozen times a second is probably overkill. As far as resource planning goes, an Ethernet based Sockets/TCP network optimized for one browser at a time should optimally call the clock tick about 10 times a second.

# 5 Porting Step by Step

This section outlines what you need to do, step by step, to get the InterNiche Internet Services working in your embedded system.

## 5.1 Provide the System Routines

This is the step in the port where you provide the glue routines to initialize the web server and give it access to the networking and memory resources. The subsections below outline the routines needed and provide examples and suggestions. In all cases examples are available in the example packages.

### 5.1.1 Memory: npalloc and npfree

All the HTTP Server's dynamic memory is allocated by calls to `npalloc()` and released by called to `npfree()`. The syntax for these is exactly the same as the standard C library calls `malloc()` and `free()`, with the exception that buffers returned from `npalloc()` are assumed to be pre-initialized to all zeroes. In this respect npalloc is like `calloc()`.

If your embedded system already supports standard `calloc()` and `free()` calls, all you need to do is add the following lines to ipport.h:

```
#define npalloc(size)  calloc(1, size)
#define npfree(ptr)    free(ptr)
```

In the event your system does not support `calloc()` and `free()`, you will need to implement them. An exhaustive description of how these functions work and sample code is available in "*The C programming Language* by Kernighan and Ritchie.

### 5.1.2 TCP and Sockets

The only Sockets calls made directly from the HTTP engine are `recv()`, `send()`, and `close()`. These are coded as `sys_recv()`, `sys_send()`, and `sys_close()` in the C sources. Since the *sys_* commands calling semantics are identical to the originals, the following defines in **ipport.h** would map directly to a standard UNIX Sockets library:

```
#include "sockets.h"   /* required includes will vary from system to system */
#define sys_recv(sock, buf, len, flags)  recv(sock, buf, len, flag)
#define sys_send(sock, buf, len, flags)  send(sock, buf, len, flag)
#define sys_close_socket(sock)  close_socket(sock)
```

### 5.1.3 String Library

The HTTP Server expects the C compiler's library to provide the routines it uses for string manipulation. These are all standard routines defined in ANSI C. The routines are:

- strcat()
- strcpy()
- strcmp()
- strncmp()
- stricmp()
- strstr()
- strchr()
- sprintf()

All these routines are probably already available in the embedded system's runtime libraries. If they are not, they are easy to code.

The exception is `sprintf()`. This is not easy to code. And in many C development systems, the `sprintf()` call is intertwined with the C run-time system. To resolve this problem, InterNiche provides source code for a `sprintf()` routine in the example package. It is in the file ttyio.c.

### 5.1.4 Timer Tick Routine

The timer tick routine is simply a routine that needs to be called periodically to further work on HTTP connections that could not previously complete. This is usually due to a large HTML file being sent on a TCP connection with limited resources. The whole file cannot be sent at once since the networking layer does not have enough buffer space to store it while the network moves it.

What actually happens in the HTTP Server is that it does a "reasonable" amount of work each time it gets control, and then returns to the system. In these cases, the httpd structure (and its sub members) will be left filled in with state data to allow the work to resume next time the timer is called.

How often should the timer be called? This is a tradeoff between web performance and system efficiency. For embedded systems that use the web pages for status monitoring and configuration, 10 times a second is about right.

The timer tick routine to be called is named `http_loop()`. It is called with no parameters and returns the number of connections that had pending work to be done. In a system where CPU cycles are a critical resource this can be used to back off the calling frequency when little work is pending (i.e. http_loop() == 0) and increase it when several connections are pending.

## 5.2 Setting Up the Source Tree

### 5.2.1 Header Files

InterNiche normally places all header files that are shared by more than one module in the h directory. Header files that are only used by one module are placed in that module's source directory. The table below lists the primary header files for the HTTP Server and the Webserver. The top portion of `httpd.h` contains a section of tunable parameters. The porting engineer may want to change one or more of these, although it is not usually necessary. The remaining portion of this file as well as the other two header files should normally not be changed.

| Filename | OK to Change? | Description |
|---|---|---|
| h/httpd.h | Yes | Defines struct httpd and its many substructures. Defines HTTP states, request types, socket flags, file types, and HTTP return codes. |
| h /webserver. h | No | Contains prototypes for Webserver functions and has externs for functions and variables needed by the Webserver |
| httpsvr /ce_http.h | No | Contains prototypes for HTTP Digest Authentication code |

### 5.2.2 Server Source Files List

The main source files in HTTP and Web Servers are listed below. Please take special note that only a few of these files should be changed directly, as doing so will make it harder for InterNiche to provide technical support and may make them incompatible with future upgrades of product.

| Filename | OK to Change? | Description |
|---|---|---|
| http.c | No | Main HTTP server loop and primary http routines. Includes `ht_sendbuf()`, which is indirectly called by web applications via `GIO_OUT`. |
| htutils.c | No | HTTP server subroutines, HTTP protocol token strings, and the file types array. Includes the `buildstatusline()` function which may be called by web applications. |
| http_mod.c | No | Contains the http task and module arrays and the http prep, init, start, and close functions. |
| htauth.c | No | Authorization code sources. |
| httpconn.c | No | Functions for listening for, and accepting, new connections and for checking for data on existing connections. |
| htpers.c | No | Persistent connections and pipelining logic. |
| http_nt.c | Yes | code for http menu functions and the http menu and parameter definitions used by the CLI. The porting engineer may modify or add to these. |
| httpport.c | Yes | Functions likely to be modified by the porting engineer. |
| webserver.c | No | Main Webserver loop and the primary Webserver functions |
| wbs_mod.c | No | Contains the Webserver task and module arrays and the Webserver prep, init, start, and close functions |
| wbscgi.c | No | CGI handling code and the output functions used by the GIO module |
| wbsupload.c | No | Functions for handling a `POST` request for a file-upload. |
| htmllib.c | No | Functions for getting values from, and putting values to, forms. |
| wbs_nt.c | Yes | Webserver menu and parameter definitions used by the CLI and the code for Webserver menu functions. The porting engineer may modify or add to these. |

### 5.2.3 VFS Compiler Source Files List

The VFS Compiler has a single large C file that calls only standard C library routines. It shares header files with the Webserver and the HTTP Server, but it is not linked into target executable. It is unlikely that you would need to change this code. Moreover, it normally only needs to be compiled once for your development system.

| | |
|---|---|
| vfscomp.c | source for VFS Compiler |

The remaining files read by the VFS Compiler or they are outputs from the compiler. All are specific to the web application, and they are located in the web application directory. These files are described in more detail in the VFS Compiler User Guide.

| webfiles.txt | input | Name of each web file in the application and any compiler parameters for that file. |
|---|---|---|
| cmprstrings.txt | output & input | List of strings that will be replaced with tags. Compiler generate initial list. User may modify. |
| cmprreport.txt | output | Report to tags used for compression and the saving gained with each. |
| htmldata.c | output | Compiled VFS files, struct vfs array, and cgi_stub code produced by compiler. |
| htmldata.h | output | Declarations and prototypes for `htmldata.c`. |
| cgi_template.c | output | Contains stubs for CVARs and cgi routines. |

## 5.3 Authorization

The NicheStack HTTP server supports two types of authentication. These are informally called "Basic" authentication and "Digest" or "MD5" authentication. These methods were described above in the section Authentication and Security. As described above, both methods are based on user names and passwords. With Digest authentication, the user names and passwords are encrypted and the Digest includes several other fields to improve security.

When the VFS Compiler is used, the compiler reads any authorization parameters from the line for the file in `webfiles.txt`. To use authentication, the line for the file must contain the parameter "`-b`" for Basic or "`-d` " for digest authentication.. The compiler adds the appropriate flags to the `vfs_file` structure for that file. `htmldata.c` contains an ASCII representation of the array of `vfs_file` structures. The HTTP Server and Webserver determine the authentication required for each file by looking at the flags field in the `vfs_file` structure for that file. If the VFS Compiler is not used, then array of `vfs_file` structures must be created by hand, and each structure must contain the proper authorization flag.

### 5.3.1 User and Password Lookup Routine

In HTTP the user/password concept is used by both Basic and Digest authentication. The user_auth routine calls check_permit(), one of the functions in misclib/userpass.c. The syntax for the routine is as follows:

```
int
check_permit(char *username,
             char *password,
             uint32_t appcode,
             uint32_t permissions
```

The user name and password are passed in plain text - the uudecode is already done. For both appcode and permissions, the parameter value will be ANDed with the table value. The result must match the parameter value. Note that this method leads to two special cases:

- A parameter of 0 will match any value in the table: any value ANDed with 0 equals 0. For added security, the porting engineer could modify the `check_permit` routine to made 0 an invalid parameter.
- A table value of 0xFFFFFFFF will match any parameter value. Any value ANDed with 0xFFFFFFFF will equal the original value.

The `appcode` field allows a user entry to be restricted to a specific module. For example, if the entry contains an appcode of “http”, then the appcode parameter in the call must be “http” to match this entry.

The `permissions` field enables the porting engineer to devise a mechanism for restricting a valid user to only those permissions granted by the matched entry.

Entries can be added to the user table by the `change_usertab()`function. If `INCLUDE_CLI` is defined, then the entries could be added to the table by an init time script that makes repeated calls to the user menu command. For example the command

```
user –a –u guest –p guest
```

would give the user/password combination “guest”, “guest” all permissions for all modules.

Please see the Command Reference document for details concerning the `user` command.

### 5.3.2 The make_nonce() routine for Digest Authentication

As with Basic Authentication, Digest Authentication is based on a simple challenge-response paradigm, and the initial request from the browser usually does not contain the proper authentication field. The server gives a 401 error response, but with Digest Authentication, this response includes a "nonce" value and it specifies the algorithm to use to create the Digest for authentication. The MD5 algorithm is the most common algorithm used, and it is the only one supported by the InterNiche implementation.

The browser will then use the MD5 algorithm to create a digest based on the user name and password, the given nonce value, the HTTP method, and the requested URI. The InterNiche HTTP Server supports the additional Quality of Service (qos) parameters required by most browsers today. With Digest Authentication, the username and password are never sent in the clear. Just as with Basic, the username and password must be prearranged in some fashion.

The method for computing the nonce value is a large factor in the strength of Digest Authentication. The sample `make_nonce()` routine in `http_port.c()` uses the current time as a seed to a random number generator. The porting engineer is encouraged to substitute a stronger method for creating the nonce value.

## 5.4 Creating Web Applications

The first thing your HTTP Server will need is something to serve. You need at least one HTML page to see if anything works or not. If some or all the pages are to go in VFS, then you probably want to set up your makefiles and HTML compiling in an organized way right from the start.

You can start with one page, or a whole tree. They can be created by any commercial HTML page authoring tool or built manually with a simple text editor.

If you want to get started coding right away and do not have pages of your own, you can use some pages from the example program. HTML links do not have to be resolvable for you to prototype a server, but any graphics files your pages reference should be included.

### 5.4.1 Things to Consider

It is possible for one HTTP connection to send a very large file in one call. TCP flow control and task switching should allow other non-webserver modules to continue, but other HTTP connections will not be serviced until this file has been completely sent. With today's high-speed network connections, many megabytes of data can be sent in a very short period. Normally the problem only occurs with very large files or when the file to be sent contains data that is dynamically constructed in nested CGI functions.

### 5.4.2 The index page

Whenever you connect to a web server without specifying a specific file (for example by typing in a URL like http://www.company.com or http://192.168.2.2), the browser will then send a GET request for a file, but the filename will be only "/", the UNIX slash, for "root". Nearly all servers are coded to return a default web page when they receive a request of this type. By tradition, the name of the default web page is called "index.htm". Typically, it is at the root of the links to the sites web application pages. If you already have a set of HTML pages, one of them is probably named index.

With the InterNiche servers, you are likely to want to name the default web page, "index.iws". The "iws" extension allows you to embed included files and scripts within the default web page. The name of the default web page is set at compile time by the define, "DEFAULT_WEB_PAGE". It can be set dynamically by the menu command http config.

### 5.4.3 The VFS base and VFS paths

If HT_LOCALFS is NOT defined, then VFS provides the entire file system, and the variable vfs_root_path will be set to "/". When HT_LOCALFS is defined, VFS files can live in a special directory specified by the VFSROOTPATH variable defined in ipport.h. By default, this path is "/vfs/". The path to the file system root must use UNIX slashes. VFS will convert these to Windows back slashes if necessary. VFSROOTPATH must **not** set to NULL.

VFS is flat file system with no concept of subdirectories. However, files can be put in subdirectories as long as all references to them include the path from the VFSROOTPATH, the vfs base.

For example, if the root path is "/something/vfs" and a gif file is located at "/something/vfs/images/mylogo.gif" then the VFS name for the file would be "images/mylogo.gif". Assuming the HTTP Server was located at the IP address "192.168.2.2", a GET request could request for the file "mylogo.gif" would be http://192.168.2.2/images/mylogo.gif.

### 5.4.4 Command Line Utilities

The HTTP and Web servers provide several CLI utilities to assist in development of your custom web applications. Inclusion of these in your target application is dependent on whether `INCLUDE_CLI` is defined in `ipport.h`.

### http htdump

Command Name

`htdump - dump contents of httpd structure(s)`

Syntax

```
htdump [-a] [-f] [-r] [-s] [-u] [-w]
```

Parameters

| | |
|---|---|
| | Command without arguments dumps all fields for all active httpd structures. |
| `-a` | Dump all active httpd structures. |
| `-f` | Dump all form information. |
| `-r` | Dump content of request(httpfinfo structure)(s). |
| `-s` | Dump content of httpsockinfo structure(s). |
| `-u` | Dump content of uploadfile structure(s). |
| `-w` | Dump http structure for the calling web application. |

Description

This command dumps all or the requested field(s) from the httpd structures of the active http connections. The options may be combined. For example, you can dump the form and request information for each active http connection (-f -r).

Notes/Status

- A struct httpd is allocated when a connection is made to the HTTP server and freed when the connection is closed. A struct httpd has pointers to several substructures that are allocated as needed.
- The -w option can only be used when htdump is called from a web application.
- Unless the -w option is specified, the information will be displayed for all active connections.

Location

This command is provided by the `httpsvr` module when `HTTPSVR` is defined.

### http config

Command Name

`http config - display or modify http server configuration`

Syntax

```
http config [-d | -e] [-i <timeout>] [-m <numbytes>] [-n <timeout>] [-p
<http_root_path>][-r <numbytes>] [-t <numbytes>] [-w <webpage>]
```

Parameters

| | |
|---|---|
| (none) | Command without arguments displays the current state of http configuration parameters |
| `-d` | Disables the HTTP Server. |
| `-e` | Enables the HTTP Server. |
| `-i` | Integer: idle timeout in seconds for persistent connections |
| `-m` | Max bytes HTTP will transmit without yielding |
| `-n` | Integer: idle timeout in seconds for non-persistent connections |
| `-p` | Argument of type `string` sets the HTTP root path. |
| `-r` | HTTP Receive buffer size in bytes. Range: 512-16384 bytes |
| `-t` | HTTP Transmit buffer size in bytes. Range: 512-16384 bytes |
| `-w` | Argument of type `string` sets the default web page. |

Description

This command displays or sets http configuration parameters

Notes/Status

- An attempt to re-enable the HTTP Server (`-e` option) within 15 seconds of disabling it, may not be successive. Repeating the command after a few seconds should succeed.
- The `-i` option is only available if HT_PERSISENT is defined. Normally it will be defined.
- For the `-i` and `-n` options, an argument of '0' represents an infinite timeout.

Location

This command is provided by the `httpsvr` module when `HTTPSVR` is defined.

### http netstat

Command Name

`http netstat - display HTTP Server statistics and status`

Syntax

```
http netstat
```

Parameters

This command takes no arguments

Description

This command displays status information for the httpsvr module.

Location

This command is provided by the `httpsvr` module when `HTTPSVR` is defined.

## 5.5 The VFS Compiler

The VFS Compiler is optional, but unless your site has very few web pages, using the VFS Compiler will greatly simplify integrating your web pages into VFS.

Refer to the VFS Compiler User Guide section for more details.

## 5.6 CGI Routines

A conventional web server's CGI capabilities are usually oriented toward executing a separate process and returning the results. Because many embedded systems have limited multi-process capabilities (but lots of C code capabilities), the Webserver's CGI mechanism is oriented toward calling a C function. This C function can do most anything it wants, however the calling interface is designed assuming it will be processing the results of an HTML form.

The following example shows the semantics of the CGI calls:

```
int setip_cgi(struct httpd *hp,     /* connection the get or post came in on */
      struct httpform *form,        /* a list of name/value pairs */
      char **filetext)              /* optional text the return to web server */
```

The input to these routines is fairly simple assuming you are familiar with HTML forms. You should understand what a name/value pair from a form is before you attempt to code a CGI routine. The CGI routine is passed a form structure that contains all the information from the filled in form. The structure is defined as follows:

```
/* a form reply name/value pair: */
struct name_val {
   char *name;
   char *value;
};

struct httpform {
   char *action;                /* our ".frm" or ".exe" file name */
   int      nv_ct;              /* number of entries in name_val table */
   struct name_val  nameval[1]; /* the first name/value entry */
};
```

Actually there are two structures here, The first contains a single name/value pair. The memory for the strings is obtained by `npalloc()` before the CGI routine is called and is freed via `npfree()` after the CGI routine returns. Therefore the CGI routine should copy any data it will need later.

The second structure is primarily a list of name/value pairs. The first member is a pointer back to the file name, which may be useful if you want to write a CGI routine which is invoked by more than one form. The next member is the number of name/value structures which follow. The rest of the structure is an array of the name/value structures.

The name/value structures are filled in the order they were received from the browser and not necessarily the order they appear in the form.

Once the browser has sent a form to the server, it is expecting an HTML page in response. This form reply page may be anything from a simple static "thanks for filling in our form" type of message to the most complex page in the system. Normally, the requested CGI function will simply carry out its action and then return a message or a file. However, it can do anything it wants, so we will briefly mention two other possible actions, before describing the normal case.

From the Webserver's perspective, it is possible for the CGI application to close the existing connection or to even open another connection. If the application opens a different connection, then it would be entirely responsible from managing that connection and coordinating with the client browser.

Sometimes it may take so long to obtain and send the response data that the application may want to yield the CPU and spread the responses over several scheduling periods. In order to do this, the CGI application would set a timer causing it to renew execution periodically and it would return `GIO_PENDING` from the initial function call. This will tell the Webserver to not to close the socket and not to clear the busy flag for the connection. The function can then periodically call `GIO_OUT` to send data over the connection. When it finally completes sending all data, it must call `wbs_closecb()`, which signals the Webserver to clean-up the connection.

In the normal case, the CGI routine will carry out its function and then return. The return should consist of one of the following values:

```
GIO_PENDING      /* Processing will continue periodically */
FP_ERR           /* Internal (code) error                 */
FP_FILE          /* fileptr is valid                      */
FP_DONE          /* CGI did everything, just clean up     */
EHT_BADREQ       /* Bad request from the browser          */
```

If the return is not one of the above values, or if it is `FP_ERR`, the Webserver will return an HTTP 500 (Server error) to the client browser.

If the application wants to send a text message in response to the request, it should format the response, send it via `GIO_OUT`, and then return `FP_DONE`. This tells the Webserver to clean up the connection. The text message could either be success response, or an error response. If the application returns EHT_BADREQ, the Webserver will send a HTTP 400 (bad request) response to the browser with the text, "Form Parse Error".

One of the parameters in the call to the CGI function was the address of a pointer variable `(char **fileptr)`. The application can set this to point to a file and return `FP_FILE` to the Webserver. The Webserver will then send the file as a response to the request. The file may be a simple file or a file containing embedded commands and/or include files.

# 5.7 File Upload interface and sample application

## 5.7.1 File Upload Introduction

The HTTP Server supports file upload. This allows a client or a browser to upload a file to the host device. The implementation of file upload is based on RFC1867.

The file upload feature in the HTTP Server is implemented in generic way, so that it can be fine-tuned for specific requirements. Complete implementation, along with a sample application is provided. Either use the application as such, or modify it as per requirements.

The browser loads a HTML page containing a <FORM>. The fields of the form include username, password and file. The file is specified using "type=file" attribute of <INPUT> tag. When the form is submitted, the HTTP Server authenticates the username and password and upon successful validation it saves file to disk /flash.

## 5.7.2 How to use

Write a HTML page with a form having the four fields mentioned in previous section. Sample coding for FORM tag is shown below. Note the enctype and method attributes. The action attribute should be set the address of the Webserver.

```
<form enctype="multipart/form-data" method="POST" action="http://10.0.0.21">
```

For testing file upload, the following form was used.

```
<html>

<head>
<title>multipart/form-data test</title>
</head>

<body>

<table cellspacing="0" cellpadding="0">
  <tr>
    <td align="center" valign="top" width="150">
      <a href="http://www.iniche.com/">
        <img src="embweb.gif" border="0" alt="The Embedded Web" WIDTH="150" HEIGHT="100">
      </a>
      <br>
      <br>
    </td>
    <td>
      <h1 align="center">
        <img src="ilogo3.gif" alt="InterNiche" width="350" height="80">
      </h1>

      <p>Watch this!</p>
```

```
      <form enctype="multipart/form-data" method="POST" action="http://10.0.0.21">
      <div align="center">
      <center>
      <table>
        <tr>
          <td>
            Username:
          </td>
          <td>
            <input type="text" name="username" size="40" tabindex="2">
          </td>
        </tr>
        <tr>
          <td>
            Password:
          </td>
          <td>
            <input type="password" name="password" size="40" tabindex="3">
          </td>
        </tr>
         <tr>
          <td>
            Filename:
          </td>
          <td>
            <input type="file" name="file" size="40" tabindex="1">
          </td>
        </tr>
      </table>
      </center>
      </div>
      <div align="center">
      <center>

      <input type="submit" border="0" name="login" value="Submit">
      </center>
      </div>
    </form>
    </td>
  </tr>
</table>
</body>
</html>
```

### 5.7.3 Interface

File upload feature is implemented based on RFC1867, "Form-based File Upload in HTML". As specified in the RFC, form data is sent using HTTP POST with a Content-type: multipart/form-data. The RFC describes how the encoding for multipart/form-data is done.

This feature is enabled by defining HT_FILEUPLOAD in ipport.h file. The following declarations should also be included with the webport.h file.

```
#ifdef HT_FILEUPLOAD
extern char * ht_formstart(char *host);
extern char * ht_formpara(char *host, char * start, int len, int flags);
extern char * ht_formdone(char *host);
#endif
```

The above three functions define the interface for file upload. The HTTP Server receives the form with HTTP POST message and multipart/form-data, parses it and then hands over the data to file upload application using the above API.

The web browser does file upload by submitting a FORM (with a <INPUT type=file> control) to the HTTP Server. HTTP Server parses the data and sends the name-value pairs to the application. "name-value" pairs are sent to application using the following API.

- ht_formstart() is initially called, so that application can prepare itself to received data for a new form.
- ht_formpara() is then called for each name-value pair. HT_NEW flag is passed for a new pair. If all the data for the name-value pair is present, then flag HT_DONE is also passed.
- If all data has not been received for a name-value pair, then whenever web-server receives new data, it calls ht_formpara(). This time HT_NEW is not set. If all the data for the name-value pair is present the flag HT_DONE is set.
- When all name-value pairs have been passed to the application, ht_formdone() is called to inform the application.

All APIs have a host parameter, which points to the host sending the form, file. It is provided so that access can be restricted based on "host". All APIs return a "char *", which is a message that they may want to send to the browser. If no message is to be sent, they return NULL. By default, ht_formstart() returns a message indicating that form processing has started. Similarly ht_formdone() returns a message stating that form processing is complete. ht_formpara() returns "#" characters to show file upload in progress.

### 5.7.4 Sample File Upload Application

This file upload application is written so that it accepts the following name-value pairs from a form.

- username
- password
- file (to be uploaded)
- submit

It validates the username and password and then saves file on the disk/flash. The generic method of `misclib/userpass.c` is used for authentication. The username, password combination is set to "guest", "guest".

It is assumed that username and password are the first two fields, followed by file. With this assumption, we start saving file as we receive it.

The file upload application is implemented in the file `htupload.c`. By default, it shows a '#' for every 2048 bytes of received data. If `HT_VERBOSE` is disabled in `htupload.c`, the '#' marks are not displayed.

# 6 VFS Compiler User Guide

## 6.1 VFS Compiler Introduction

The VFS Compiler is a software program, which takes the files for your web pages and "compiles" them into the C structures that become VFS files in your embedded server. The web page files can be any file you would normally put on a web server. HTML and GIF files are the most common, but the compiler can accept any binary file.

The executable file for the VFS Compiler is `vfscomp.exe`. It and its makefile are located in the directory `utils/vfscomp` directory of your source code distribution. When you are ready to build your web application, you should make and then move the executable to the directory that contains your web pages. When invoked, it will create a file named `htmldata.c` that includes the VFS file data and structures for your input files. This file will be linked into your embedded image by the initialization routines described in the next section.

**webfiles.txt**. The porting engineer must provide an input file, "`webfiles.txt`". This file will contain a list of all files that are to become part of the embedded HTTP Server's VFS, including the names of any GIF, JPEG, or Java bytecode files that you want in your VFS.. It is a simple text file with one file name per line. Each file name may be followed by one or more options specifying how the compiler should handle each file You can create it with any plain text editor, such as vi, Notepad, or Brief.

Any kind of file can be included in `webfiles.txt`. If the VFS Compiler does not understand the type (as indicated by the file extension), it will simply encode a binary image of the file.

Options. The VFS Compiler supports several optional features which are useful in building even a modest web server. The most important are:

- Tag Compression of HTML files
- Set Basic or Digest Authentication flags on a per file basis
- Create the routine stubs and prototypes for CGI executable files

To build an initial prototype web server with just a few pages, you can probably use the defaults. Options will be described in detail in the sections below

**htmldata.c**: At its simplest, the compiler takes the list of files in `webfiles.txt` and produces, as output, a file, "`htmldata.c`". The first part of this file consists of the "compiled" data that will be read into memory by VFS to represent each file. The data for each file appears as a single C string array. One of the options allows the data for some, or all, of the files to be placed in an output file that has a different name.

**vfslist[ ]**: The second part of `htmldata.c` consists of an array of structures, "`vfslist[ ]`". Normally, this array would be created by the VFS Compiler. The array contains one entry for each VFS file. Each entry provides the initialization values for a `struct vfs_file`, which contains the information and pointers that allow VFS to access the file.

## 6.2 VFSComp - VFS Filesystem Compiler

VFSComp

`VFSComp - VFS Filesystem Compiler`

Syntax

`VFSComp [-options]`

Options

These options apply to all files and so must be set on the command line

| | |
|---|---|
| `-arr <arrayname>` | File system array name. Default = "<basename> files" |
| `-base <basename>` | Base name for the following fields. Default = "vfs" |
| `-i <infile>` | Input filename. Default = "<basename>.txt" |
| `-html` | Normal Webserver mode. The equivalent to setting the following options on the command line: `-c -arr webfiles -i webfiles.txt -o htmldata` |
| `-nvfs` | Do not generate vfiles structure |
| `-ss` | Sort tags in cmprreport.txt by bytes saved. Default is to sort alphabetically |
| `-v` | Toggle verbose mode |
| `-z const` | Generate code using `Const` keyword |
| `-D` | Emit dependency information for make |

These options may appear either on the command line or within `<infile>.txt`

| | |
|---|---|
| `-b` | Require Basic Authentication (user name/password) for access to this file |
| `-c` | Do file compression |
| `-cs` | File is case sensitive |
| `-d` | Require Digest Authentication (MD5) for access to this file |
| `-o <outfile>` | Put output for this file into the specified file |

These per-file options are only valid within `<infile>.txt`

| | |
|---|---|
| `-cgi <funcname>` | Generate a CGI function stub for this symbol |
| `-cvar type nat TOKEN` | Generate full mapping of C variable in `htmldata.c`. See Displaying C Variables in Web Pages section of the HTTP and Web Servers Technical Reference |
| `-uhdr` | Call `ht_adduserheaders()` to add additional user-specified HTTP headers. The user must add code to this stub in `democgi.c` |

Description

The VFS compiler is a developer tool used to prepare and insert a list of files into the VFS filesystem. These files may be web pages, images, private key files, configuration, or any other file that will not grow in size over the product lifetime. To use the resulting VFS files, the output `.c` file should be added to the build, and a call to the `XXX_setup()` routine should be called before first use.

The `-cgi` and `-cvar` options are primarily used in relation to web pages for the HTTP server. They provide code generation features to facilitate the calling of CGI functions and the display of certain variable values within the code.

Notes/Status

- The VFS Compiler replaces the HTML Compiler

The following are some lines picked from the `webfiles.txt` file in the Web example included with the Webserver delivery.

```
#Sample lines that might be in webfiles.txt

index.iws
xmltest.iwx -cs
demo.css
header.inc
prod.crypto
maze.jpg -o gifdata.c
profdata -cgi profdata_cgi -b
```

The first two lines illustrate a comment followed by a blank line. The VFS Compiler will ignore any line that begins with a pound sign ('#"), a space, a carriage return or a line feed character.

The first file name has the InterNiche extension that designates a script file with HTML output. The second file has the InterNiche extension that designates a script file with an XML output. The `-cs` option is required because XML is case sensitive.

The final line shows the use of multiple options. The "`-cgi`" option requires that the next parameter specify a CGI function name. The "`-b`" option, could have be placed before the `-cgi` parameter.

By default, all VFS related output from the VFS Compiler is written into the file, "`htmldata.c`." One common use of the "`-o`" option shown above would be to have all image data written to a file, "`gifdata.c`". The following lines illustrate another use:

```
logoA.gif -o companyA.c
logoB.gif -o companyB.c
```

GIF data that applies only to specific cases could be placed in different files.

The `webfiles.txt` file in the web example code included with the Webserver delivery provides many more examples of the use of options.

## 6.3 Tag Compression

Tag Compression works by taking a list of text patterns that occur in the web application files and replacing each pattern with a 1 byte codes. The text patterns are often HTML tags, hence the name. To the VFS Compiler, a "tag" is any expression that is surrounded by '<' and '>' characters that appear on the same line. As described below, unless compression is turned off, the VFS Compiler will automatically scan the files and compress HTML tags that are used more than once. Together, these simple techniques provide file compression on the order of 20-50% or more. The VFS decompression routine is very fast and requires only a few lines of code. If the porting engineer wishes, HTML compressed files can be further compressed with standard compression techniques.

The basis of tag compression, is to substitute an 8-bit (1 bytes) value in the files, in place of tags and strings that are used more than once in the total set of file. The 1 byte compression codes are simply a 7 bit index in an array of tag text strings. The high bit is always set so the decompression logic can identify the tags in the compressed HTML stream. Because the most significant bit is used to indicate that the value is compression symbol, this method cannot be used for binary files that contain 8-bit values. Before attempting to compress a file, the VFS compiler scans the entire file to ensure that it does not already contain any 8-bit values. The VFS compiler does not scan or attempt to compress files with the GIF, JPG, or MAP extensions.

With one bit used to signify the compression symbol, there are only 127 unique values that can be used to substitute for tags and strings. However, in most case, this is more than enough to provide substantial compression.

To the VFS compiler, a "tag" is any expression that is surrounded by '<' and '>' characters that appear on the same line. The VFS compiler is actually more general. It can compress any string pattern that is used more than once in the entire set of web files. provides, "string compression", a way to significantly compress VFS files. With this method the decompression code in the target binary is very small and fast. The basis of string compression, is to substitute an 8-bit value in the files, in place of tags and strings that are used more than once in the total set of file.

The HTML decompression code can be omitted by omitting the #define

```
#define HTML_COMPRESSION       1
```

from your build. If this is done then the porting engineer should ensure that the VFS Compiler's command lines also have the Tag Compression feature disabled.

**"CmprStrings.txt"**: Unless the `-nc` option is specified on the command line, the VFS compiler will automatically search through the non-image files listed in `webfiles.txt` and create a list of all HTML tags that are used more than once in the total collection of files. It then selects the tags which provide the greatest savings and writes them into the top portion of "`CmprStrings.txt`", with one selected tag per line.

The VFS compiler keeps an internal copy of this list of tags and uses it to substitute a single 8-bit value for each copy of the tag found in any of the listed files. For each file in the `vfsfiles` list (in `htmldata.c`), there is a field in the VFS structure that is used by the VFS compiler to indicates whether or not the file has been compressed.

During run-time operation, VFS keeps an array in memory that corresponds to the lines in `cmprstrs.txt`. When a VFS file is read, VFS checks the file structure to see if the file has been compressed, and, if so, VFS will substitute the original tag for each 8-bit value found in the file.

**usercmprstrs.txt**: In most cases, the compiler will not find 127 unique HTML tags that are used more than once and a few of the selected tags will provide very little savings. For this reason, the VFS compiler will also accept as input, a file created by the user, "`usercmprstrs.txt`". The porting engineer can examine the text in the web files and select strings that are commonly used. These strings are entered into `usercmprstrs.txt`, with one string per line. The specified string may contain any number of 7-bit characters (e.g., any of the standard ASCII characters), including blanks and quotation marks. The string is terminated by a newline character(s). The compiler skips lines that start with one of the characters: '\0', '\n', '\r', or '#'. Thus comments can be entered by putting'#' as the first character on the line.

The compiler will attempt to use all strings entered in `usercmprstrs.txt`. The compiler will reduce the maximum number of tags that it can find by the number of compression string lines in `usercmprstrs.txt`. For example, if `usercmprstrs.txt` contains 60 compression strings, and the compiler finds 80 HTML tags that are used more than once, it will only select the 67 HTML tags that provide the most savings (60 + 67 = 127).

**cmprreport.txt**: Each time the VFS compiler runs, it creates a report, "`cmprreport.txt`". Each line in the report lists:

- The entire string that was compressed
- The number of characters in the string
- How many times the string appears in the set of files
- The saving obtained by compressing all occurrences of this string

Because one copy of the entire string and the character that represents it must be stored in runtime memory, the savings is computed by the formula:

```
(string frequency - 1) times (string length - 1)
```

**"-ss" option**: By default tags in `cmprreport.txt` are sorted in alphabetically order, making it easier to spot tags that are very similar. If it would not affect the programs operation or clarity, the porting engineer may want to provide greater compression by changing the web pages so that these similar lines are identical.

The `-ss` option tells the VFS compiler to sort tags in `cmprreport.txt` by the amount of savings obtained by using this tag. This option applies only to tags that are automatically found by the compiler. Strings taken from `usercmprstrs.txt` will always be shown in the same order they appeared in the file.

**"-cs" option**: The `-cs` option forces the compiler to use different values to represent tags that differ only by the case of one or more characters. This option only applies to tags that are automatically found by the VFS compiler. Strings in `usercmprstrs.txt` are always treated as case sensitive.

Because HTML is not case sensitive, the VFS compiler uses case insensitive comparisons by default as it selects tags for compression. Each time the compiler finds a new tag, it must compare it to the list of tags already found. Before it does this comparison, it first substitutes lower case characters for all alpha characters in the tag (the set of characters surrounded by '<' and '>' characters that appear on the same line). In this way, for example, the same 8-bit value can be substituted for both "<HEAD>" and <head>. When VFS decompresses tags, all alpha characters in the decompressed tags will be lower case. For case sensitive files, you must prevent this conversion to lower case, either by using the -cs option or by using the -nc options to suppress tag compression.

Tips for maximum savings. The following are some tips on how to achieve maximum savings:

**Note**: In the tips below, the quotes used to indicate specific text are not used in `usercmprstrs.txt`.

- Don't put quotes around strings in `usercmprstrs.txt` unless you want the quotes to be part of the string to be matched.
- Run the VFS compiler a number of times and adjust the entries in `usercmprstrs.txt` based on the cmprreport.txt that is written after each run.
- Strings do not need to be full words. Certain letter combinations found in many different words may make a good entry in `usercmprstrs.txt` (e.g., "tion");
- When the VFS compiler performs compression, it uses strings in the order they are found in the file. For this reason, shorter strings should be placed at the bottom of `usercmprstrs.txt`. For example if the entry "tion" is above the entries "implementation" and "authentication", then the compiler will substitute a value for "tion" before it looks for the latter two words, and therefore, it will not find either.
- If a word is nearly always preceded by a space, then including the space in the string may achieve more savings than the string without the space.
- There are several reasons why cmprreport.txt might show negative savings for a specified string. The two most common are:
    1. Strings higher up in `usercmprstrs.txt` caused substitution to occur before the compiler searched for these strings.
    2. There may be unwanted whitespace following the desired string on the line in `usercmprstrs.txt`.

The `usercmprstrs.txt` file found in the HTML Demonstration directory provides examples of strings that were selected for compression in the Web files provided in this example. These strings were selected simply by looking for common words in the files and running the compiler a number of times to see which provided the best savings. The selected tags and strings used for compression reduce the size of the non-image files by more than 40%. If reducing file size was of high importance, then significantly better savings could have been achieved in this example by putting more effort in selecting the best set of strings.

## 6.4 VFS CGI Routines

The `-cgi` option tells the compiler that the file name at the start of the line refers to a CGI command (executable routine) that should be called whenever a file is requested.

Generating the VFS file entries and stubs for the corresponding code for the functions to be executed when the CGI file are requested can be tedious and error prone, so the VFS Compiler does it for you. It will generate a function stub for the code, write a prototype for the function in `htmldata.h`, create a `vfs_file` structure in the `vfslist[ ]` array, and it will generate a stub for the CGI function, which it writes to a section of `htmldata.c` that is not compiled (ifdefed out).

The porting engineer should copy the function stub from `htmldata.c` into a file that will be compiled, and then add the code needed to carry out the intended function.

# 6.5 VFS Compiler C Code Generation Features

## 6.5.1 Displaying C Variables in Web Pages

Rather than asking the VFS Compiler to generate code stubs to be called whenever a web application wishes to display the contents of a variable, the `-cvar` option provides a far simpler way. Consider the following example:

```
tcptimeout -cvar number %04x realVariableName
                                                    XXXXX
```

This line contains 5 fields, which direct the code generation. The fields in the above example are explained below:

| | |
|---|---|
| tcptimeout | VFS file name: The name of the SSI file created in the VFS. |
| -cvar | Compiler Option: Causes vfscomp to generate full mapping of a C variable to HTML output. |
| format identifier | One of the format identifiers listed in the next table |
| realVariableName | The name of the C language variable you want to appear in the HTML output. The variable will be cast to the type above, so make sure they are compatible. |
| XXXXX | A name for a `#define` which will be generated in vfscomp's output files (by default `htmldata.h` and `htmldata.c`). This can be any legal, unique pattern of a `#define` . It should be selected to make the generated code more readable. |

Legal format identifiers are listed below. For numerical values (e.g., long, short, etc.) the format identifier is the actual sprintf-type format string starting with the '%' sign, e.g., `%04x`.

| Format identifier | Used for: |
|---|---|
| charptr | Display a C string |
| ip_addr | Display IPv4 address |
| ip6_addr | Display IPv6 address |
| <format> | sprintf format string for display of a numerical value |

The C code generated in the default HTML output file is a single routine named `wbs_prog_cvar()`. It is surrounded by an "`#ifdef NOTDEF`", so you will need to copy it to a local per-port file to compile it. You may need to add externs for the compiler to find the referenced variables.

The generated code uses formatting and display routines defined in the file htmllib.c Both the generated code and htmllib.c are quite compact, allowing a lot of dynamic content to be added with very little time and space.

### 6.5.2 Parsing Values from Forms

When a CGI application is called as a result of a `POST` or `GET` request, all parameters provided in the request are passed in an array in the "`httpform`" structure that is part of the "`httpd`" structure passed to the CGI application. The format of array included in httpform was described in CGI Routines. The routines described in this section allow the CGI application to request the value of specific form entries by name.

For each of the various parameter types: string, integer, Boolean, IPv4 address, and IPv6 address, there are two separate versions of the routines. The first version is more general and is required when the form may contain several instances of the parameter, all with the same name. The second version is a simpler and easier to use common cases.

The general versions of the functions all start with the "ht_" prefix and all have exactly the same form:

Name

`ht_get_form_XXX ()`

Syntax

```
int ht_get_form_XXX(struct httpd *hp, char *name, XXX *addr, int index)
```

Parameters

(see below)

Description

This function searches for a making name in the name/value array within the specified httpd structure. It writes the value of the variable at the address pointed to by the XXX parameter. If more than one name /value pair ...etc.

Returns

Number of name/value pairs with the specified name.

The following is a list of the general and the simpler version of the functions for returning values from a form:

```
/* Return a string */
int ht_get_form_ str(struct httpd *, char *name, char **str, int index)
char  *get_form_ str(struct httpd *hp, char *name);

/* Return an integer */
int    ht_get_form_int(struct httpd *hp, char *name, uint32_t *value, int index);
int    get_form_int(struct httpd *hp , char *name, uint32_t *value);

/* Return a boolean-typically whether or not a checkbox or button was selected */
int    ht_get_form_bool(struct httpd *hp, char *name, int *value, int index);
int    get_form_bool(struct httpd *hp, char *name);

/* Convert a dotten notation IP4 address into an ip_addr(4-byte binary address */
int    ht_get_form_ip4addr(struct httpd *hp, char *name, ip_addr *ipptr, int index);
char  *get_form_ip4addr(struct httpd *hp, char *name, ip_addr *ipptr);

/* Convert a colon separated ASCII IP6 address to a 16-byte binary address */
int    ht_get_form_ip6addr(struct httpd *hp, char *name, uint8_t *buf, int index);
int    get_form_ip6addr(struct httpd *hp, char *name, uint8_t *buf);
```

## 6.6 Using the VFS Compiler: An Example

This section will guide you through creation of a very simple web-application and demonstrates how to invoke a "C" function from a webpage, how to extract form values and how to use the VFS Compiler to put it all together. Our single page application draws a form and a box, the background color of which was provided to a "C" function within the form.

As with all applications created with the VFS Compiler, we first create entries in the "`webfiles.txt`" file, allowing it to create a source file containing the VFS file entries and the compressed and encode the display files. For this example, the `webfiles.txt` contains only 2 lines. The file `index.iws` is the actual HTML file and the second line creates a keyword called "`formcolor`" which is mapped to the internal function "`echo_formcolor()`". When invoked, this function returns a string used to set the background color of the box.

```
index.iws formcolor
-cgi echo_formcolor
```

The embedded command is delimited "<#" and "#>" and can be seen on line 8 of the file:

```
1. <html>
2. <head>
3. <title>Simplest Demo I Could Devise.</title>
4. <style type="text/css">
5. div.demobox {
6.    border: 1px solid #000;
7.    padding: 5px;
8.    background-color: #<# formcolor; #>;
9.    display: inline-block;
10. }
11. </style>
12. </head>
13. <body>
14. <form>
15.    <input type=text size=6 name=color>
16.    <input type=submit value="Enter RRGGBB and Click This Button">
17. </form>
18. <div class="demobox">
19.    <p>The background color of this box was set by echo_formcolor()</p>
20. </div>
21. </body>
22. </html>
```

Among other things, the VFS Compiler produces the file "`htmldata.c`", a portion of which is:

```
#ifdef NOTDEF /* Don't use these here; copy them to another file */

/* echo_formcolor() - CGI routine.
Returns bitmask (see manual). */

int
echo_formcolor(struct httpd *hp, struct httpform *form, char **filetext)
{
/* add form processing code here */

    return (FP_DONE); /* generic return */
}

#endif /* end of routine stubs */
```

We will take this code fragment into our own application and enhance it to read the value of the form's "`color`" field and include it in the output stream. If the function was not invoked as part of a form submission, it will return a default value.

```
int
echo_formcolor(struct httpd *hp, struct httpform *form, char **filetext)
{
   GIO *gio = hp->ctx->gio;

   if (gio == NULL)
      return (-1);

   if ( !form )
      gio_printf(gio, "ff00ff");
   else
      gio_printf(gio, get_form_str(hp, "color"));
   return (FP_DONE); /* generic return */
}
```

Finally, bringing `htmldata.c` and the rest of our application into an executable will give us a webserver that, as its default document, displays a form and a box - the background color of which can be changed by submitting legal RRGGBB values in the form.