LibHTTPD

A Library for Embedded Web Servers

API Guide and Reference

Version 1.1

March 2002

Hughes Technologies

Although best efforts have been made to ensure the completeness and accuracy of the material presented in this document, Hughes Technologies Pty Ltd does not warrant the correctness of the information provided herein.

This software product is distributed under a dual-license scheme that provides the user with a choice of licenses. This software may be used either under the terms of the GNU Public License (GPL), or the LibHTTPD OEM License. A copy of both the GPL and the OEM license is included within the software distribution. The GPL is also available from the GNU Project's web site at http://www.gnu.org. The LibHTTPD OEM License is available from the Hughes Technologies web site at http://www.Hughes.com.au.

This document has been designed to be printed on a duplex (double sided) device.

Preface

Intended Audience

This document describes the functionality and programming mechanisms provided by LibHTTPD. Any person reading this document must be a programmer with a good understanding of the C programming language. All examples and details pertaining to the API are provided in C. The reader should also have a basic understanding of the HTTP protocol and of the operation of a web server.

Document Conventions

This manual has been designed to be printed on US Letter paper. While many parts of the world utilise the A4 paper size (Australia included), it is not possible to print A4 formatted documents on US Letter paper without loss of information. However, printing of US Letter formatted documents on A4 will result in a correct representation of the document with somewhat larger margins than normal.

Throughout this manual, parts of the text have been flagged with the symbol that appears in the margin opposite this paragraph. Such pieces of text are viewed as being important. The reader should ensure that paragraphs marked as important are read even if the entire manual section is only being skimmed. Important sections will include information such as tips on improving the performance of your applications, or areas where mistakes are commonly made.

Contact Information

Further information about Ember and its related software can be found on the Hughes Technologies World Wide Web site. The web site includes the latest version of LibHTTPD, documentation, example software, references to customer sites, and much more. Our web site can be found at

http://www.Hughes.com.au

Product support and information are available over the Internet via electronic mail. For product support, please contact support@Hughes.com.au and for product information please use info@Hughes.com.au. If required, you may also use traditional mail using the address below

Postal Mail

PO Box 432
Main Beach
Queensland 4217
Australia

This page left intentionally blank

Table of Contents

Introduction

With the wide spread availability of web browsers on virtually every conceivable hardware platform, offering web based access to an application or device is becoming a common requirement. Although there are several freely available web server implementations, most standard web servers are just too large or inconvenient to use. Embedding a web server into a device or a stand-alone application requires a light-weight and low overhead implementation. LibHTTPD provides a solution to this problem.

Included in the library are the following modules:

- An implementation of a subset of HTTP
- A symbol table providing automatic handling of HTML FORM data
- A content management module allowing both static and dynamic (application call-back) content

Using the API routines provided by these modules, an application can provide a complete web based interface to either itself or the device on which it is running.

Overview

LibHTTPD provides you with the tools to incorporate a low overhead web server into your application or device. Using this library you can provide secure and efficient access to both static and dynamically generated content. Included in the library are routines that simplify all aspects of the creation and operation of an embedded web server. The basic series of operations that your application must perform is:

- Create a "web server" instance
- Define the content that is available via the web server
- Receive and interpret an HTTP request received from a browser client
- Process the request and return the required response via HTTP

LibHTTPD provides API routines to handle all the above tasks. A rough outline of the routines and the steps involved is shown below.

To create a web server instance, your application must call httpdCreate( ). If successful, an httpd handle will be returned. This handle is used in all future interaction with the API library.

If desired, the library will create both error and access logs in a format identical to that used by the Apache web server. If you wish to generate logs you must call httpdSetAccessLog( ) and httpdSetErrorLog( ) to set the destination of the log information. The default action is to not log access or error information.

The next step is to define the content available via the web server. Unlike other web server environments, content is only available via LibHTTPD if it has been specifically defined as web content. The library allows for the following forms of content:

- Static content : HTML stored in static text buffers within the application
- File content : An external file
- Wildcard content : Any file in the specified directory
- Dynamic content : A callback to a C function that generates the desired output

Once the content has been defined, the application enters a tight loop that handles the web requests. Each iteration of the loop will call the following API routines

httpdGetConnection( )	- Accept an HTTP connection
httpdReadRequest( )	- Read and pre-process the HTTP request
httpdProcessRequest( )	- Generate the required output
httpdEndRequest( )	- Disconnect the HTTP session and cleanup

When httpdProcessRequest( ) is called, the library will identify what content has been requested by the client, determine if that content is available from the server, and undertake whatever tasks are required to send the requested content to the client. The server may simply read and transmit the contents of a file or it may perform a callback to a user provided C function.

To use LibHTTPD in your application you must include the httpd.h header file in your source. The header is installed in /usr/local/include by a default installation. You will also need to link against libhttpd.a which is installed in /usr/local/lib by default. You can do that by adding the following to you link stag

```
-L /usr/local/lib -lhttpd
```

API Reference

General Server Setup Routines

The routines in this section relate to the creation and basic configuration of the web server instance. All details relating to a web serer instance are maintained in a single structure known as the server handle. Once a server handle is created, various aspects of it's configuration can be set. The server handle is used in all other calls to API routines.

Because the server handle contains all relevant information about the HTTP Server, it's content, and any requests being processed by the server, it is possible to create multiple servers in a single process. Naturally, the different servers would need to be configured to use either different IP addresses of different TCP ports. However, this library does not provide any non-blocking IO operations for server handles.

httpdCreate ( )

```
httpd * httpdCreate ( host , port )
        char        * host
        int         port
```

httpdCreate( ) is used to create an new web server instance. The returned handle is used for all further interaction with the API library that pertains to the web server created by this function. The host argument, if provided, defines a single IP Address to which the server will bind. If NULL is provided as the argument then the server will bind to all addresses available on the host machine. The port parameter is the numeric TCP port on which the server will listen. The predefined constant HTTP_PORT can be used if the server is to run on the default web server port (i.e. TCP port 80)

Example :

```
server = httpdCreate( "192.168.1.1", HTTP_PORT);
if ( server == NULL )
        perror ( "Couldn't create HTTP server" );

server2 = httpdCreate ( NULL , 2048 );
```

httpdSetAccessLog( )

```
httpdSetAccessLog ( server, fp )
        httpd       *server;
        FILE        *fp;
```

Each valid HTTP request processed can result in a "Common Log Format" log entry being created. httpdSetAccessLog( ) is used to specify the file into which the log entries are written. The fp argument must be a file pointer, as created by calling fopen( ), that is open for write access. You can specify a standard file pointer, such as stdout or stderr if you wish.

Example :

```
fp = fopen ( "/tmp/access.log", "a" );
httpdSetAccessLog ( server, fp );
```

httpdSetErrorLog( )

```
httpdSetErrorLog ( server, fp )
        httpd       *server;
        FILE        *fp;
```

Each time an invalid HTTP request is received, the library can generate an error log entry. To specify the destination of the logging you must call httpdSetErrorLog( ) with an open file pointer (see httpdSetAccessLog)

Example :

```
httpdSetErrorLog ( server, stderr );
```

httpdSetFileBase( )

```
httpdSetErrorLog ( server, path )
        httpd       *server;
        char        *path;
```

To simplify the definition of file content, a "file base" can be set for the server. If set, any relative paths used to define the location of the file associated with a content entry will be relative to the file base. Using a file base can allow you to have greater flexibility over the physical, on-disk location of your file content. If all file content is defined using a relative path, the library will always search for it using the file base to determine the actual file path. Simply altering the file base definition will allow you to relocate your entire web content tree without modifying any of the file path definitions.

Example :

```
httpdSetFileBase ( server, "/usr/local/www-pages" );
```

Content Definition Routines

The content that is to be made available via the web server is defined using the API routines listed below. A content entry (such as an HTML page) is defined using a location and a name. The location specifies the directory within the URL tree in which the content resides, and the name specifies the name to associate with the content. For example, a URL path of /graphics/logos/hughes.gif would be specified using directory of /graphics/logos and a name of hughes.gif. Each URL requested is split into it's directory and name values. The content tree is then searched for a entry matching the specified directory and name pair. If none exists a 404 File Not Found error is returned.

When content is registered with the library, it is possible to specify that it is an index entry. An index entry is the content entry that is returned to the client if the client specifies the name of a directory rather than a actual file within that directory. On most web server implementations, a file called index.html is regarded as the index entry for a directory. Within LibHTTPD's content tree, any item can be flagged as the index entry for the directory in which it resides. Naturally, defining more than one index entry makes no sense and the results of doing such are undefined.

The programmer may also associate a "preload function" with any piece of content. If a requested content entry has been defined with a preload function, the function is executed prior to the content being sent to the client. If the preload function returns a negative value then the request is aborted and the content is not returned. If a zero value is returned then the content is sent to the client as usual. A preload function can be of use when restricting access to the contents of the web server. If the server is not to be accessed outside business hours, or if only certain network addresses are allowed to use the server, then a preload function can be used to test for these conditions for all content requests.

httpdAddCContent( )

```
httpdAddCContent ( server, dir, name, indexFlag, preload, functPtr )
        httpd       *server;
        char        *dir, *name;
        int         indexFlag, (*)( )preload;
        void        (*)( ) functPtr;
```

If the content referenced by dir/name is to be returned to the client then the function referenced by functPtr will be called with a single argument of the server handle pointer.

Example :

```
void index_callbackl ( server )
        httpd       *server;
{
                    httpdOutput(server, "<HTML><BODY>Hello There</BODY></HTML>\n");
}

httpdAddCContent( server, "/", "index.html", HTTP_TRUE, NULL, index_callback);
```

httpdAddFileContent( )

```
httpdAddFileContent ( server, dir, name, indexFlag, preload, path )
        httpd       *server;
        char        *dir, *name;
        int         indexFlag, (*) ( ) preload;
        char        *path;
```

httpdAddFileContent( ) adds an external file as a content entry. The path argument is the filesystem path to the file in question (not anything to do with the URL path). If the path begins with a / then it is assumed to be a complete file path. If not, it is assumed to be a path relative to the file base path (see httpdSetFileBase)

Example :

```
httpdAddFileContent( server, "/", "index.html", HTTP_TRUE, NULL, "/usr/local/www/index.html" );
```

httpdAddStaticContent( )

```
httpdAddStaticContent ( server, dir, name, indexFlag, preload, buf )
        httpd       *server;
        char        *dir, *name;
        int         indexFlag, (*)( ) preload;
        char        *buf;
```

httpdAddStaticContent( ) adds an internal text buffer as an HTML content entry.

Example :

```
#define index_content "<HTML><BODY>Hello There</BODY></HTML>\n"
httpdAddStaticContent( server, "/", "index.html", HTTP_TRUE, NULL, index_content );
```

httpdAddWildcardContent( )

```
httpdAddWildcardContent ( server, dir, preload, path )
        httpd       *server;
        char        *dir;
        int         (*) ( )preload;
        char        *path;
```

httpdAddWilcardContent( ) instructs LibHTTPD that it may scan the directory specified by the path argument for any requested files from the dir URL path directory. If, for example, a directory existed that contained all the images associated with your HTML pages, adding a Wilcard Content entry for that directory would allow any image to be retrieved without the need for a specific File Content entry for each file.

Example :

```
httpdAddWildcardContent(server,"/graphics", NULL, "/usr/local/www/graphics" );
```

httpdAddCWildcardContent( )

```
httpdAddCWildcardContent ( server, dir, preload, functPtr )
        httpd       *server;
        char        *dir;
        int         (*) ( )preload;
        void        (*)( ) functPtr;
```

httpdAddCWilcardContent( ) registers a C function callback to be executed when any file within the specified directory is requested. It is the responsibility of the C function to determine the URL that was requested (using httpdRequestPath( ) for example) and taking the appropriate action.

Example :

```
httpdAddCWildcardContent(server,"/users", NULL, send_user_info );
```

Connection and Request Handling Routines

The basic operation of a web server is a tight loop of repeated request handling. The routines covered in this section are used to accept and process HTTP requests within the request loop. Each loop iteration includes

- Accepting a TCP/IP connection from the browser client
- Reading and interpreting the HTTP request sent by the client
- Processing the request and returning any resulting data
- Cleaning up and terminating the connection

httpdGetConnection ( )

```
int httpdGetConnection ( server )
        httpd       *server;
```

httpGetConnection( ) is used to accept an incoming HTTP connection request. When called, the routing will block until a connection is received on the server's TCP/IP socket. If an error is encountered while accepting the incoming connection, a negative value is returned.

httpdReadRequest ( )

```
int httpdReadRequest ( server )
        httpd       *server;
```

httpdReadRequest( ) reads the HTTP request from the client connections and stores any form data the is sent as part of the request. Details of the request and the symbol table containing the request data are stored within the server handle. If an error is encountered during the reading of the request, a negative value is returned.

httpdProcessRequest ( )

```
httpdProcessRequest ( server )
        httpd       *server;
```

When httpdProcessRequest( ) is called, it evaluates the request, locates the requested content, and responds to the client. If the content entry is static or file based content, the information is sent directly to the client. If it's dynamic content, the C function specified during the content definition is called to generate the response.

httpdEndRequest ( )

```
httpdEndRequest ( server )
        httpd       *server;
```

httpdEndRequest( ) must be called when the current request information is no longer required. It frees allocated memory buffers, clears out the symbol table, closes the client network connection, and performs any other cleanup that is required before a new request can be handled.

Response Generation Routines

When delivering static or file based content as a response to a request from a browser, the library takes care of all the behind the scenes work involved with sending a valid HTTP response. If the content being requested is to be generated dynamically via a C function callback, it is the C function code that is responsible for generating all the output. To simplify the process of generating dynamic HTTP responses, LibHTTPD provides a range of response generation routines.

Although a complete understanding of the HTTP protocol is not required, a basic understanding of the protocol is necessary. Any response sent by a web server to a browser client must be a complete HTTP response. An HTTP response is comprised of two sections, the headers and the body. A line containing nothing but a new line character is used to signify the end of the headers and the start of the body.

The first line of the header section contains the response code. There are many valid response codes defined in the HTTP standard. A list of the responses is provided in Appendix A. Also contained within the headers is a content type definition. The content type tells the browser how to interpret the response body (i.e. it is HTML text or a JPEG image for example). A list of commonly used content types is shown in Appendix B.

The library provides you with routines to set a response code, to set a content type, and to add any further header lines that you may wish to include in your response. However, if you just wish to generate a normal HTML response you do not have to worry about it. The library will generate a default set of headers for you. The routines previously mentioned can be used to modify or augment the standard headers if you need to. If you do not then you need not worry about them.

It should be noted that if you do not specifically generate the HTTP headers then they will be generated for you when you first call httpdOutput( ) or httpdPrintf( ). Naturally, once the headers have been sent, using the library routines to modify them has no effect. It should also be noted that all header and response information is reset to it's default value for each request received by the library. If you want to provide a specific header in every response then you will have to add it manually when each request is processed.

httpdOutput ( )

```
httpdOutput ( server, buffer )
        httpd       *server;
        char        *buffer
```

httpdOutput( ) is used to send the contents of the provided text buffer to the client browser. If the output text includes variables (specified using $varname syntax), and those variables exist in the current request symbol table, then the variable references are replaced by the current variable values. More details on variables can be found in the Form Data / Symbol Table section of this manual.

Example :

```
httpdOutput ( server, "Hello $name. Welcome to the test server" );
```

httpdPrintf ( )

```
httpdPrintf ( server, format, arg, arg, … )
        httpd       *server;
        char        *format;
```

httpdPrintf( ) is used to generate output using a format string and a variable length list of format arguments. The result of using this function is the same as performing a normal C printf( ) directed at the client browser. Unlike httpdOutput( ) any variable references contained in the format string ARE NOT expanded.

Example :

```
httpdPrintf( server, "Hello %s.  Welcome to the server running in process number %d"
        username, getpid( )  );
```

httpdSetContentType ( )

```
httpdSetContentType( server, type )
        httpd       *server;
        char        *type
```

If you need to generate a response containing something other than HTML text then you will need to use this routine. A call to this routine will override the default content type for the current request. When the HTTP headers are generated, the specified content type information will be used.

Example :

```
httpdSetContentType ( server, "image/jpeg" );
```

httpdSetResponse ( )

```
httpdSetResponse( server, responseInfo )
        httpd       *server;
        char        *responseInfo;
```

This routine is used to override the default response code returned to a client browser. By default, an HTTP Response code of 200 (successful request) is returned to the browser. If you wish to return a different response code then this routine must be called with the new response information before the headers are sent to the client.

Example :

```
httpdSetResponse ( server, "301 Moved Permanently" );
```

httpdAddHeader ( )

```
httpdAddHeader( server, header )
        httpd       *server;
        char        *header;
```

If you need to add a header line to the set of headers being returned to the client it can be done using httpdAddHeader.

Example :

```
httpdSetResponse ( server, "307 Temporary Redirect" );
httpdAddHeader ( server, "Location: http://www.foo.com/some/new/location");
```

httpdSendHeaders( )

```
httpdSendHeaders( server)
        httpd       *server;
```

If you use neither httpdOutput nor httpdPrintf to generate your dynamic content then you will need to send the HTTP headers manually. You can do that by simply calling httpdSendHeaders ( ). Once the headers have been sent (either manually by calling httpdSendHeaders, or automatically via httpdOutput or httpdPrintf) the library will not send them again for the current request.

Example :

```
httpdSetContentType ( server, "image/jpeg" );
httpdSendHeaders ( server );
generateJpegData( server );
```

Form Data / Symbol Table Routines

If a request containing user supplied data (usually the result of filling in a form) or cookies is received by the server during a call to httpdReadRequest( ), the data is extracted from the request and loaded into a symbol table. The symbol table will exist for the duration of the current request and will be cleared as soon as the request has been completed (i.e. when httpEndRequest( ) is called). Each entry in the symbol table contains a name field and a value field. The entry also contains a nextValue field, which is defined as a pointer to another symbol table entry. If the nextValue field is not NULL it points to the head of a list of subsequent values for the current variable. A variable may have multiple values if it is the result of a multi-select element in an HTML form.

As was mentioned in the explanation of the httpdOutput( ) routine, the contents of the symbol table are automatically used by some response generation routines. You can also manually access the contents of the symbol table from your own dynamic content functions. If, for example, you want to store the contents of an HTML Form into a database, you would need to extract the form field values from the symbol table in your C function. The routines listed below allow you to access the symbol table's contents in a few different ways.

httpVar * httpdGetVariableByName ( )

```
httpdGetVariableByName( server, varName )
        httpd       *server;
        char        *varName;
```

This routine will search the symbol table for an entry that matches the name provided. If one is found, the symbol table entry (i.e. a pointer to a httpVar structure) is returned. If the named entry cannot be found a NULL is returned.

Example :

```
varPtr = httpdGetVariableByName ( server, "username" );
if ( varPtr != NULL)
        uname = varPtr->value ;
```

httpVar * httpdGetVariableByPrefix ( )

```
httpdGetVariableByPrefixe( server, prefix )
        httpd       *server;
        char        *prefix;
```

If a range of variables exist with a known prefix, the first matching variable can be found using this routine.

httpVar * httpdGetNextVariableByPrefix ( )

```
httpdGetNextVariableByPrefixe( varPtr, prefix )
        httpVar     *varPtr;
        char        *prefix;
```

This routine is used to find subsequent variables in the symbol table that have a name beginning with the specified prefix..

Example :

```
varPtr = httpdGetVariableByPrefix ( server, "hughes_" );
while ( varPtr != NULL )
{
        printf("Name = %s, Value = %s \n", varPtr->name, varPtr->value;
        varPtr = httpdGetNextVariableByPrefix ( varPtr, "hughes_" );
}
```

httpVar * httpdGetVariableByPrefixedName ( )

```
httpdGetVariableByPrefixedName( varPtr, prefix, remainder )
        httpVar     *varPtr;
        char        *prefix, *remainder;
```

If an application knows a variable's prefix and the rest of the variable name, it can use this routine to access the variable rather than having to merge the values and use httpdGetVariableByName( ). This is a convenience routine and offers no functionality beyond that offered by httpdGetVariableByName( )

Example :

```
prefixPtr = httpdGetVariableByName ( server, "multi-select-values" );
while ( prefixPtr != NULL ) {
        prefix = prefixPtr->value;
        varPtr = httpdGetVariableByPrefixedName(server, prefix, "_username");
        printf("%s_username = %s\n", prefix, varPtr->value;
        prefixPtr = prefixPtr->nextValue;
}
```

httpdAddVariable( )

```
httpdAddVariable( server, name, value )
        httpd       *server;
        char        *name, *value;
```

An application may need to add one or more standard variables to the symbol table for later use in httpdOutput( ) strings or other methods of dynamic content generation. A variable can be added to the symbol table using the httpdAddVariable( ) routine.

Example :

```
httpdAddVariable( server, "background_color", "#FFFF30" );
httpdOutput( server, "<BODY BGCOLOR=$background_color>\n");
```

httpdDumpVariables( )

```
httpdDumpVariables( server )
        httpd       *server;
```

If an applications wants to see that contents of a symbol table, it can dump the entire table to the server process's standard output using the httpdDumpVariables( ) routines. This routine is intended for debugging purposes only.

Miscellaneous Routines

httpdSetCookie( )

```
httpdSetCookie( server, name, value )
        httpd       *server;
        char        *name, *value;
```

To add a cookie to the HTTP response, simply call this routine with the name and value of the cookie.

httpdUrlEncode( )

```
char * httpdUrlEncode( buf )
        char        *buf;
```

Given a text buffer, the data will be encoded into a form suitable for use as part of a URL (i.e. spaces and other offending characters are converted into their alternate form). The modified data is returned to the calling function.

httpdRequestMethod( )

```
int httpdRequestMethod( server )
        httpd       *server;
```

When called, this routine returns an integer value representing the method of the current request. The value can be one of those listed in the httpd.h header file, including HTTP_GET and HTTP_POST

httpdRequestMethodName( )

```
char *httpdRequestMethodName( server )
        httpd       *server;
```

Rather that returning a numeric value as httpdRequestMethod( ) does, this routines will return the name of the request method (e.g. GET or POST etc).

httpdRequestPath( )

```
char *httpdRequestPath( server )
        httpd       *server;
```

When called after a request has been received, this function will return the URL path of the request.

httpdRequestContentType( )

```
char *httpdRequestContentType( server )
        httpd       *server;
```

This routine will return the content type of the current request if it was included in the request information.

httpdRequestContentLength( )

```
int httpdRequestContentLength( server )
        httpd       *server;
```

This routines will return the content length of any data sent with the current request.

Appendix A – HTTP Response Codes

Informational 1xx
- 100 Continue
- 101 Switching Protocols

Successful 2xx
- 200 OK
- 201 Created
- 202 Accepted
- 203 Non-Authoritative Information
- 204 No Content
- 205 Reset Content
- 206 Partial Content

Redirection 3xx
- 300 Multiple Choices
- 301 Moved Permanently
- 302 Found
- 303 See Other
- 304 Not Modified
- 305 Use Proxy
- 306 (Unused)
- 307 Temporary Redirect

Client Error 4xx
- 400 Bad Request
- 401 Unauthorized
- 402 Payment Required
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 406 Not Acceptable
- 407 Proxy Authentication Required
- 408 Request Timeout
- 409 Conflict
- 410 Gone
- 411 Length Required
- 412 Precondition Failed
- 413 Request Entity Too Large
- 414 Request-URI Too Long
- 415 Unsupported Media Type
- 416 Requested Range Not Satisfiable
- 417 Expectation Failed

Server Error 5xx
- 500 Internal Server Error
- 501 Not Implemented
- 502 Bad Gateway
- 503 Service Unavailable
- 504 Gateway Timeout
- 505 HTTP Version Not Supported

Appendix B – Common Content Types

HTML Text	text/html
Plain Text	text/plain
Rich Text Document	text/rtf
GIF Image	image/gif
JPEG Image	image/jpeg
PNG Image	image/png
TIFF Image	image/tiff
PDF Document	application/pdf
Postscript Document	application/postscript
Comma Separated Values	application/csv

Example

```
#include <stdio.h>
#include "httpd.h"

/*
** This is a static page of HTML.  It is loaded into the content
** tree using httpdAddStaticContent( ).
*/
#define test1_html "<HTML><BODY>This is just a test</BODY>"


/*
** Below are 2 dynamic pages, each generated by a C function.  The first
** is a simple page that offers a little dynamic info (the process ID)
** and then sets up a test link and a simple form.
**
** The second page processes the form.  As you can see, you can access
** the form data from within your C code by accessing the symbol table
** using httpdGetVariableByName() (and other similar functions).  You
** can also include variables in the string passed to httpdOutput( ) and
** they will be expanded automatically.
*/
void index_html(server)
        httpd      *server;
{
        httpdPrintf(server, "Welcome to the httpd server running in process number %d<P>\n", getpid( ) );
        httpdPrintf(server, "Click <A HREF=/test1.html>here</A> to view a test page<P>\n" );
        httpdPrintf(server, "<P><FORM ACTION=test2.html METHOD=POST>\n" );
        httpdPrintf(server, "Enter your name <INPUT NAME=name SIZE=10>\n");
        httpdPrintf(server, "<INPUT TYPE=SUBMIT VALUE=Click!><P></FORM>\n" );
        return;
}

void test2_html(server)
        httpd      *server;
{
        httpVar  *variable;

        /*
        ** Grab the symbol table entry to see if the variable exists
        */
        variable = httpdGetVariableByName(server, "name");
        if (variable == NULL)
        {
                httpdPrintf(server,"Missing form data!");
                return;
        }

        /*
        ** Use httpdOutput() rather than httpdPrintf() so that the variable
        ** embedded in the text is expanded automatically
        */
        httpdOutput(server,"Hello $name");
}
```

```
int main(argc, argv)
	int	argc;
	char	*argv[];
{
	httpd	*server;

	/*
	** Create a server and setup our logging
	*/
	server = httpdCreate(NULL,80);
	if (server == NULL)
	{
		perror("Can't create server");
		exit(1);
	}
	httpdSetAccessLog(server, stdout);
	httpdSetErrorLog(server, stderr);

	/*
	** Setup some content for the server
	*/
	httpdAddCContent(server,"/", "index.html", HTTP_TRUE, NULL, index_html);
	httpdAddCContent(server,"/", "test2.html", HTTP_FALSE, NULL, test2_html);
	httpdAddStaticContent(server, "/", "test1.html", HTTP_FALSE,NULL, test1_html);

	/*
	** Go into our service loop
	*/
	while(1 == 1)
	{
		if (httpdGetConnection(server) < 0)
			continue;
		if(httpdReadRequest(server) < 0)
		{
			httpdEndRequest(server);
			continue;
		}
		httpdProcessRequest(server);
		httpdEndRequest(server);
	}
}
```