Porting the ARM Webserver

Version 1.6

Programmer's Guide

ARM

ARM DUI 0075D

Porting the ARM Webserver
Programmer's Guide

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Change
Jan 1999	A	First release
Feb 1999	B	Second release, minor changes
Sept 2000	C	Third release
June 2001	D	Fourth release

Proprietary Notice

all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded. This document is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Contents
Programmer's Guide

Preface

This preface introduces the ARM Webserver porting procedure. It contains the following sections:

About this book

This guide is provided with the ARM Webserver protocol software.

It is assumed that the ARM Webserver porting sources are available as a reference. It is also assumed that the reader has access to programmer guides for C and ARM assembly language.

Intended audience

This document has been written for experienced embedded systems programmers, with a general understanding of what an embedded webserver does.

Organization

This book is organized into the following chapters:

Chapter 1 ***Introduction***

This chapter introduces the ARM Webserver software.

Chapter 2 ***About the ARM Webserver***

This chapter provides an architectural overview of the ARM Webserver, and discusses the *Virtual File System* (VFS), security issues, and system requirements.

Chapter 3 ***Porting Step-by-Step***

This chapter outlines what you need to do, step-by-step, to get the server working in your embedded system.

Chapter 4 ***Using the HTML Compiler***

This chapter describes how to use the HTML Compiler, a software program which takes the files for your web pages, and compiles them into C structures which become VFS files in your embedded server.

Appendix A ***Building the Demonstration Program***

This appendix details the requirements, installation procedure, and steps required to build the demonstration program.

Typographical conventions

The following typographical conventions are used in this document:

bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate.
italic	Highlights special terminology, denotes internal cross-references, and citations.
`typewriter`	Denotes text that may be entered at the keyboard, such as commands, file and program names, and source code.
`typewriter`	Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`typewriter bold`	Denotes language keywords when used outside example code.

Further reading

The following documents are referenced in this guide or provide reference material.

ARM publications

Refer to the following ARM publications for more information:

- The documentation set for the *ARM Developer Suite* (ADS).
- *Porting TCP/IP Programmer's Guide* (ARM DUI 0144).
- *ARM Network Protocols Command-line Interface Reference Guide* (ARM DDI 0145)

Other publications

The following publications may also provide useful background reading:

- Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, 2nd Edition, 1988, Prentice-Hall (ISBN 0-13-110370-8).
- RFC 1945, Berners-Lee, T., Fielding, R. and Frystyk, H., *Hypertext Transfer Protocol—HTTP/1.0*, May 1996.

Feedback

ARM Limited welcomes feedback on the ARM Webserver and documentation.

Feedback on the ARM Webserver software

If you have any problems with the ARM Webserver and you have a valid support contract, please contact your supplier. To help us provide a rapid and useful response, please submit error reports in the form specified in the support agreement, giving:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type, and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem.

Feedback on this book

If you have any comments on this book, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM Webserver porting procedure. It contains the following section:

1.1 About the ARM Webserver

The information in this manual enables you to port the ARM Webserver to an embedded system. By the end of the porting procedure, your system will be able to serve *Hypertext Markup Language* (HTML) features, for example:

- text files
- forms
- graphics
- Java Applets.

Some HTML knowledge is needed, but everything you need to know about *Hypertext Transfer Protocol* (HTTP) to port the server is in this document. If you have no previous experience with HTML, it is suggested that you learn the basics of HTML programming and write a few practice web pages.

1.1.1 Terms

In this document, the following terms are used:

Server When used without other qualification, refers to the ARM Webserver code as ported to an embedded system.

System Refers to your embedded system.

Sockets Refers to the *Transmission Control Protocol/Internet Protocol* (TCP/IP) *Application Program Interface* (API) developed for UNIX at the University of California, Berkeley.

End user Refers to the person who ultimately uses your product.

You Indicates the user or engineer who is porting the server.

Conventions used throughout the document, such as the use of bold or italic font, are explained in *Typographical conventions* on page ix in the Preface.

1.2 Demonstration program

The ARM Webserver is provided with a sample port. This links the server code to a TCP/IP stack to build a sample webserver. Currently, the demonstration program is shipped for an ARM Integrator Board, using the ARM TCP/IP stack. The ARM Integrator Board and TCP/IP can be purchased from ARM (see the ARM website at `http:\\www.arm.com`).

The demonstration program compiles with the ARM Developer Suite (ADS), version 1.0.1. It includes a sample set of HTML pages that demonstrate the basic features of webservers.

Embedded-systems engineers are encouraged to experiment with these examples before starting the actual port. The example *Virtual File System* (VFS) is layered on the debug file system on the host, so you can:

- create HTML files on disk
- edit them there until the look and feel is right
- compile them into the VFS.

The ARM ADS includes source level debuggers, so you can implement and perform actual executions of sample *Common Gateway Interface* (CGI) routines.

The ARM Webserver was designed with the restrictions of embedded systems taken into account. Like most embedded systems code, it is written in the C programming language, conforms to the ANSI standard, and is compatible with C++. It has debugging `ifdefs` and macros to cooperate with source level debuggers and *In-Circuit Emulators* (ICE) when debugging. The memory and sockets APIs were selected based on their widespread use in the embedded systems industry, but are used in such a way that they can easily be mapped to less standard environments. The HTML Compiler is designed to be invoked from within a standard makefile, and can have HTML files as dependencies. All the HTML files are designed to be placed in a single directory and compiled into a single library that can be easily linked. The entire mapping of `typedefs`, local macros, including system header files, and so on, is done in a single include file, `webport.h`, which is included by all the C source files.

If you are already familiar with the server code and want to start porting immediately, see Chapter 3 *Porting Step-by-Step*. Otherwise, it is recommended that you browse through the next chapter to get an idea of what facilities are available on your server, and how to make the best use of them.

Chapter 2
About the ARM Webserver

This chapter provides an architectural overview of the ARM Webserver, and discusses the VFS, security issues, and system requirements. It contains the following sections:

2.1 Server architecture

The ARM Webserver was specifically designed for use in an embedded system, unlike most UNIX servers. UNIX servers tend to optimize speed at the expense of size and complexity, whereas ARM favors simplicity and minimal code size. Here are some examples of where ARM has taken a divergent approach.

Memory requirements

A primary consideration of every aspect of the design is to minimize the amount of firmware storage, static RAM, and dynamic RAM required.

Multitasking

Since many embedded systems have limited multitasking and command shell capabilities, handling a connection by UNIX-like `fork()` and `exec()` functions may not be practical. The ARM server handles multiple connections by way of a linked list of dynamic memory structures (usually one per connection) with state information. A single regular timer tick loops through the list and gives each connection CPU time to do some work. This work usually involves sending more data into a socket.

File system Many embedded systems have no hard disk or file system capabilities. Even when they do, it is not always the best place to keep the HTML data. Therefore, this server has a lightweight, read-only VFS which, in cooperation with the HTML Compiler, handles keeping HTML data in ROM or RAM, and supports compression and decompression. For information on the HTML Compiler, see Chapter 4 *Using the HTML Compiler*.

Name/value pair support

Most embedded systems that implement a webserver encounter a problem when they need to *iterate* pages (that is, when they need several pages that are all nearly identical, but numbered differently). An example is the page describing port statistics, used in the demonstration program for the eight-port Ethernet hub. On a conventional webserver, you would have to maintain eight separate pages for the eight ports, for example, `portstat1.html`, `portstat2.html`, and so on. A clickable bitmap, whereby you could get statistics by clicking on a port, would have links to the eight different pages.

The name/value support allows you to maintain a single page and to code your links with the port number (or whatever data you want to pass) as standard form name/value pairs. In the example above, a single page named `portstat.html` could be linked to by coding `portstat?port=1`,

`portstat?port=2`, and so on. This saves space and makes your code cleaner and more maintainable. Refer to the demonstration program for examples.

The major components of the server are the HTTP engine, the VFS, and the CGI. Figure 2-1 shows how they fit together, and they are described in more detail in this section. The gray shaded boxes represent the modules provided as part of the ARM Webserver.

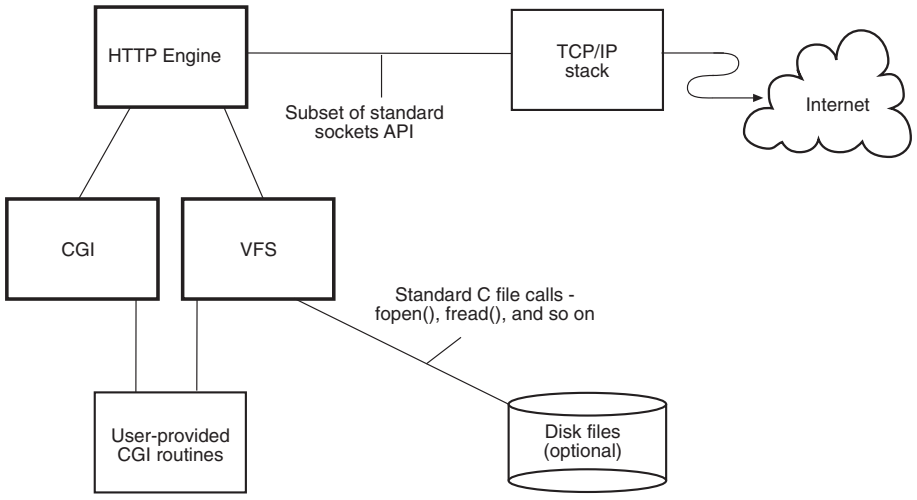


Figure 2-1 Server components

2.1.1 HTTP engine

The HTTP engine code performs the following tasks:

- parses incoming HTTP requests from a newly connected socket, and builds outgoing HTTP headers for replies
- builds and frees the `httpd` objects (the per-connection dynamic data structures)
- handles *server-side includes* (SSIs) embedded in HTML files
- diverts HTTP `POST` and `GET` form replies to CGI code
- keeps clickable bitmap `.map` file information encoded in small tables, and resolves HTTP requests based on clicked bitmaps.

The HTTP code creates an `httpd` structure (defined in `httpd.h`) and adds it to a master list for each new connection. A single regular timer tick loops through the list and gives each connection CPU time to do some work, which usually involves sending more data to the end user. The maximum amount of data to allow on a single tick can be set in `webport.h` by setting the value of the macro `HTTPMAXSEND`.

The bulk of the HTTP code is in `httpsrv.c`, which you should not need to change.

2.1.2 Virtual File System

The VFS contains all the file input/output routines called by the HTTP and CGI modules. It performs the following functions:

- Scans all file requests to see if they are for an embedded VFS file or not. If not, the request can be forwarded to another file system.
- Handles decompression of compressed files during read operations.
- Contains flags indicating special files, such as CGI or SSI functions.

A single C `#define` is all that is required to remove either the entire VFS or the support for a native file system. However, even if all the HTML files the server will ever use can be made available in a native file system, there are other reasons to use the VFS:

- fast, direct embedded CGI handling
- file compression
- the security of having the file data in ROM.

Note

You can mix the two systems, that is, VFS files can have hyperlinks to native files, and native files can have hyper-links to VFS files.

2.1.3 Common Gateway Interface

In most embedded systems, the main purpose of CGI routines is to use data from HTML forms to configure the system. The CGI on the server is optimized for this purpose.

In general terms, CGI is an HTTP feature that allows the end user to execute a predefined program on the server. Usually, the end user initiates this by getting an HTML page that contains a form from the server. The form specifies a file the browser should reference when it sends back filled-in form data. When the end user has filled in the form, the browser sends a `GET` or `POST` request for the file referenced in the form, with the form data appended to the file name as a string of encoded text.

On a conventional UNIX webserver, 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 processes the form data and returns HTML text to be sent back to the browser.

The ARM Webserver assumes that there is no command shell capability. The file indicated by the `GET` or `POST` reply to a form is looked up in the VFS. If the programmer and HTML form author are cooperating, this file will be a CGI file, having a pointer to a CGI routine to execute instead of file data. You, as part of the porting effort, supply the CGI routine. All the form reply data items are parsed from the HTTP header and stored in a C structure, `struct formdata` in `httpd.h`, which is passed to the CGI routine.

The CGI routine is also passed a pointer to the `httpd` structure, from which it can get the transaction's socket. This routine can do whatever you want with the passed data, and return any HTML text you want. It can also send HTML data into the socket, and it can close the socket. On a system with multitasking, the CGI routine can block, and the logic in the ARM Webserver will not be affected. It can loop indefinitely, sending occasional updates to the browser. On a port to a UNIX-like system, this CGI routine could execute a shell script, allowing any standard CGI script or program to be used with the server.

The exact semantics of the CGI routine are designed to take advantage of generic form reply HTML text that you can modify and embed with the HTML Compiler. See *CGI routines* on page 3-18 for details.

2.1.4 System interfaces

System interfaces are the APIs that make system resources available to the server. These interfaces can be divided into two categories:

- memory access (`malloc()` and `free()`)
- network access (sockets).

On systems with native file systems, the file input/output calls are also found in the *network access* category. On many embedded systems, all the required APIs already exist, and you only need to include the system header files (for example, `memory.h` and `sockets.h`) in `webport.h`.

On systems where the existing memory and/or TCP/IP 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 server's required routines. A list of these routines and their exact semantics is given in *Provide the system routines* on page 3-7.

2.2 Embedded data and executable files in the Virtual File System

This section is designed to expand on the overview of the VFS (see *Virtual File System* on page 2-4). It explains, in detail, the aspects of the VFS you may want to modify:

- *File compression*
- *Layering VFS on a pre-existing file system* on page 2-8.

2.2.1 File compression

HTML files are, by default, compressed by the HTML Compiler before their data is encoded into C files. This simple compression is based on frequently occurring text in the HTML files and is called *tag compression* (as opposed to generic compression applied to HTML files) since it is based on HTML tags and patterns. Tag compression is useful because it is:

- designed to decompress quickly
- implemented with very little code
- extendable.

The resulting compressed data can easily undergo further compression using standard techniques.

Tag compression works by taking a list of text patterns which are expected to occur in the HTML files, and replacing these patterns in the HTML files with 1-byte codes. The text patterns are often HTML tags, on which the term tag compression is based. The 1-byte codes are simply a 7-bit index into 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.

The HTML decompression code can be omitted by removing the following define from your build:

```
#define HTML_COMPRESSION 1
```

If this is done, you should ensure that the HTML Compiler command lines also have the tag compression feature disabled.

You can optimize tag compression by inserting frequently used strings from your own HTML pages into the tags table. The list of tags is in the file `htcmptab.h`. Since this file contains actual C data, it should not be included in more than one C source file. Also, since it is included in the HTML Compiler, the HTML Compiler must be recompiled with the same `htcmptab.h` file with which the server is compiled.

Note

Because only seven bits of index are available, the table is limited to 128 entries.

2.2.2 Layering VFS on a pre-existing file system

The ARM Webserver can use either the VFS or a native file system. It can also use both. In the server sources, all the file system calls are made to the VFS routines listed below. These routines are coded to search the VFS for the passed file names or pointers first, and default to a native file system if the file is not found in the VFS.

The VFS can be excluded from the server by adding to `webport.h` the line:

```
#define HT_NOVFS 1 /* ifdef out the VFS */
```

This redefines the VFS routines (see Table 2-1) to map directly to the standard routines.

Support for a native file system is enabled by the define:

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

Conversely, omitting this define from your build will remove native file system support. If this feature is enabled, the ARM Webserver will expect your embedded system to support the standard file input/output routines listed in Table 2-1.

Normally, you will never have to worry about the interface to the VFS, but if you decide to modify it, you will notice that the VFS calls are similar in function to standard C library calls. Setting the `HT_NOVFS` `#define` above actually defines macros that 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. Table 2-1 lists the VFS calls and their standard equivalents.

Table 2-1 VFS calls and their standard equivalents

VFS	Standard
`vfopen()`	`fopen();`
`vfread()`	`fread();`
`vfseek()`	`fseek();`
`vfclose()`	`fclose();`
`vgetc()`	`getc();`

2.3 Users, authentication, and security

The default security mechanism supported by the server is the standard HTTP *Basic*, *uuencoded* (a simple encryption algorithm) user and password mechanism. When the end user attempts to access a secured page (usually a form that can configure sensitive aspects of the device), the browser prompts for the entry of a user name and password. The correct information must be entered before the server will provide the secured page.

Internally, the following happens:

1. The browser makes a conventional page request for the secured page.
2. The browser receives a reply from the server indicating that authorization is required.
3. The browser queries the end user for the password information.
4. The password information is uuencoded.
5. The browser requests the page again with an *authentication* field and the uuencoded password information appended to the HTTP request header.
6. The server *uudecodes* (decrypts) the password information.
7. If the user name and password are deemed valid, the server provides the page.

2.3.1 Background information

Many embedded systems were not originally designed to have lists of user names and passwords. They assumed that anyone who could physically access the device was allowed to reconfigure it, change paper trays, set IP addresses, and so on. Conversely, HTTP and webservers evolved on UNIX machines, where users with passwords are file owners. To password-protect an HTML page on a device without user lists requires the addition of a user and password facility.

Many manufacturers of these devices have already faced a similar problem when they added *Simple Network Management Protocol* (SNMP) agents to their products. SNMP `GET` and `SET` operations require community strings which are fundamentally similar to passwords. One approach used by manufacturers of such SNMP enabled products, when supporting webserver management, is to accept any user name as valid, as long as the password matches an SNMP `SET` community string.

2.3.2 Higher levels of security

While these mechanisms are secure enough for most real-world applications, they do not provide the level of security offered by public and private key encryption systems. For example, an intruder with a packet monitor and a uudecoding program could monitor a legitimate user entering a password, decode the password, and later use it to access sensitive pages on the embedded device.

Recently, a more sophisticated authentication scheme, MD5, or *Digest* authentication, has been standardized. If MD5 authentication is required, you must #define `HT_MD5AUTH` in the `webport.h` file. The type of authentication to be used is set on a per-page basis in the input file for the HTML compiler (see *Command lines* on page 4-3 for more information).

2.3.3 Requirements for embedded applications

For typical embedded applications, the system requirements are fairly straightforward. You must:

- indicate which files are protected
- provide a mechanism for checking password information.

The implementation details of this process are discussed in *User and password lookup routine* on page 3-11.

2.4 System requirements

The ability to run TCP/IP is the main system requirement for porting the ARM Webserver. If your system runs TCP/IP, then it almost certainly has the requisite hardware and system software for a webserver. Here is a brief list of specific requirements, each of which is further explained in this section:

- *TCP/IP and sockets*
- *Static memory* on page 2-12
- *Dynamic memory* on page 2-13
- *Clock tick* on page 2-14.

2.4.1 TCP/IP and sockets

Browsers communicate with the ARM Webserver by way of TCP/IP, and the most common API for TCP/IP in embedded systems is sockets. The webserver code assumes that your embedded system already has TCP/IP. If it does not, ARM can provide TCP/IP as a separate product. The code also assumes that the embedded system has some kind of API to access the TCP/IP connection services. If your API is socket-oriented, the code in the ARM Webserver package maps directly to your socket API, removing the bulk of your porting work.

If your API is not socket-oriented, you need to provide the ability to perform a TCP/IP `LISTEN` to accept incoming connections, and to read and write to TCP/IP connections. Details of this procedure are provided in *Initialization routine* on page 3-10.

2.4.2 Static memory

As with all embedded system code, the ARM Webserver takes up both code and data space. On embedded systems, the code is usually stored in ROM, and can be moved to RAM at boot time. The exact amount of code space required will vary, depending on:

- which webserver features you enable through `#define`
- your compiler
- your processor.

Table 2-2 and Table 2-3 show the sizes (from the linker) of the major modules in the *Widget* demonstration compilation (see Appendix A *Building the Demonstration Program*).

Table 2-2 ARM Webserver basic requirements in bytes

Webserver file	Thumb Code and read-only data	Read-write data	Zero-init data	Thumb ROM	RAM
`httpcgi.o`	1332	4	16	1336	20
`httpsrv.o`	3616	252	0	3868	252
`htmllib.o`	68	0	0	68	0
`httpport.o`	3196	4	232	3200	236
`vfsfiles.o`	3616	252	0	3868	252
`vfsutil.o`	532	100	0	632	100
Totals	12360	612	248	12972	860

Table 2-3 VFS basic requirements in bytes

VFS file	Thumb Code and read-only data	Read-write data	Zero-init data	Thumb ROM	RAM
`formdata.o`	244	0	0	244	0
`gifdata.o`	13788	0	0	13788	0
`htmldata.o`	6244	0	0	6244	0
Totals	20276	0	0	20276	0

Note

These figures are subject to change without notice.

Conditions of the above sizes are:

- VFS enabled
- native file system disabled
- tag compression enabled
- authentication disabled
- debug code disabled
- default buffer sizes
- ARM ADS 1.0.1 compiler with space optimizations
- memory requirements exclude C runtime libraries, board support and application-specific code.

Embedded HTML and graphics data is generated from the `.html`, `.gif`, and other source files provided with the demonstration program.

With most server installations, the VFS uses more memory that the webserver itself. All the HTML and graphics data is embedded there. The size of all the `.html` and `.gif` files, and the VFS structure overhead in the demonstration program is 20276 bytes.

2.4.3 Dynamic memory

Every time an HTML connection is established, the ARM Webserver will need to allocate memory to build the structures used to manage the connection, and buffer the connection data. The exact amount will vary, depending on:

- the buffer sizes set in `webport.h`
- whether or not the connection requires file I/O
- whether the connection accesses a file with an SSI.

In general, about 4KB of space per connection should be assumed. Most of this is used for data buffers, and can be cut considerably by reducing the buffer size defines.

In addition to the per-connection requirement, you should be aware of the number of simultaneous connections that are likely to occur. This depends on a variety of factors, listed here in order of importance:

- the number of browsers using the server at once
- the design of the browser software
- the TCP/IP stack in use by the browser
- the content of the current HTML page (graphics, frames, and forms).

In general, you should allow for one connection for each browser in use, and add one more for each graphic, frame, or form in the current HTML page. In embedded systems that only use the ARM Webserver for system monitoring and configuration, it might be practical to:

- Limit access to one browser at a time. This can be enforced based on the IP address in the TCP/IP listen and accept port code.
- Put no more than one form or graphic in each HTML page.

Doing both of these would allow the server to run with only two connections.

Dynamic memory is allocated by calls to a routine called `npalloc()`, and released by calls to `npfree()`. You must provide both of these routines. Specifications for these are found in *Provide the system routines* on page 3-7.

2.4.4 Clock tick

Not all HTTP requests can be completely resolved by the initial call to the function `http_connection()`. For example, a `GET` on a file larger than the TCP/IP window size may block, because large files might not fit into the available socket buffers. Therefore, the server needs to periodically get some CPU cycles to perform individual blocks of work, each block involving the transfer of a smaller, manageable part of the overall file. The final block of work involves cleaning up when the request has been completed.

Chapter 3
Porting Step-by-Step

This chapter outlines what you need to do, step-by-step, to get the server working in your embedded system. It contains the following sections:

3.1 Setting up your source tree

The source files for the ARM Webserver software and the HTML Compiler share some include files. Two of these include files, `webport.h` and `htcmptab.h`, will probably be modified during the course of the porting process, so you must take care to ensure that the webserver sources and the HTML Compiler sources are using the same files:

- *Source and include files*
- *Port files* on page 3-4
- *HTML compression tags table* on page 3-4
- *HTML compiler output* on page 3-4.

3.1.1 Source and include files

The source and include files are listed in the following sections.

Note

Do not change these files directly. Changing them makes it harder for ARM to provide technical support, and might make them incompatible with future server upgrades.

Source files

The source files are:

webserve\httpsrv.c The main HTTP code.

webserve\httpcgi.c The CGI (forms) handling logic.

webserve\htauth.c The authentication and authorization code sources.

webserve\htmllib.c Assorted support routines used by the webserver.

webserve\htusrcgi.c

Stub routines that can be expanded to implement CGI and SSI functionality if VFS is not in use (not required if VFS is used).

webserve\formdata.c

Generic form return HTML data.

vfs\vfsfiles.c The main VFS sources.

vfs\vfsutil.c Assorted support routines used by the VFS.

vfs\vfssync.c Synchronize VFS with a backing store such as FLASH memory.

Include files

The main include files for the above sources should not need to be changed either. These are:

`webserve\httpd.h`

The main webserver declarations.

`webserve\cgi.h`

The CGI declarations.

`webserve\htmllib.h`

Declarations for support routines.

`webserve\htusrcgi.h`

Declarations for routines in `htusrcgi.c`.

`vfs\vfsfiles.h`

The VFS declarations.

`vfs\vfsport.h`

Declarations required to port the VFS to other platforms.

HTML compiler source

The last source file that should remain unchanged is the source for the HTML Compiler. It is a single large `.c` file that calls only standard C library routines. Although it is compiled as an application for the development system and is not linked to the actual ARM Webserver system, it shares include files with the server code, and must therefore be maintained in parallel.

`htmlcomp\htmlcomp.c`

HTML Compiler source code.

Note

It is especially important to rebuild the HTML compiler if the `htcmptab.h` is modified. This file is described in *Port files* on page 3-4.

3.1.2 Port files

Port files are the files that you are expected to provide or modify from those provided in the demonstration program. These files are:

your_project\webport.h

Port definitions file. Included by all ARM Webserver C files.

your_project\httpport.c

Provides the webserver with access to, for example, sockets and timers. This also contains CGI and SSI routines that your port requires.

your_project\htmlusrd.h

Is included by the htmldata.c file produced by the HMTL compiler. It is provided to allow you to add code to the file containing VFS file structures. The default content of this file is a definition of the function vfs_setup(), used to initialize the webserver VFS and to return a pointer to the VFS data structure.

3.1.3 HTML compression tags table

You can edit the HTML compression tags table but, if you do, both the HTML Compiler and the ARM Webserver files must be rebuilt.

your_project\htcmptab.h

Compression tags table.

3.1.4 HTML compiler output

The HTML Compiler creates .c and .h files with default names. The names might be changed by modifying the HTML Compiler input file, but they are included here for completeness:

your_project\htmldata.c

Default filename for VFS data, SSI, and CGI stubs.

your_project\htmldata.h

Declarations and prototypes for htmldata.c.

3.2 Start with HTML sources

The first thing your webserver needs is something to serve. You must have at least one HTML page to determine whether your webserver is operational. If some or all the pages are to go in the VFS, you should first set up your makefiles and HTML compiling in an organized manner.

You can start with one or more pages. They can be created using any HTML authoring tool, or built manually with a simple text editor. If you want to begin coding right away and do not have your own pages, you can use pages from the demonstration program (see *Demonstration program* on page 1-3). HTML links do not have to be resolvable for you to prototype a server, but any graphics files your pages reference should be included.

3.2.1 Index.htm

Generally, you can choose the names of your pages (although it is recommended that HTML pages end in `.htm` or `.html`). The *home* or main page, however, should be named `index.htm`. Traditionally, this is the name of the page that is at the root of the system of links in a webserver. Whenever you connect to the root of a new webserver (for example, by typing in a URL like `www.company.com`), the page you first see is probably named `index.htm` or `index.html`. If you already have a set of HTML pages, it is likely that one of them is named `index.htm`.

The reason for naming the root page `index.htm` is that the server assigns this name by default if no file name is specified in the request from the browser. If using the ARM Webserver primarily for configuration and status monitoring, the end user will probably access your device by typing the IP address or host name in the URL window of a browser, for example:

```
http://192.9.200.99/
```

The browser then sends a `GET` request for a file, but the filename will be `/`. When this happens, the server is returns the `index.htm` file.

3.3 HTML Compiler

If you plan to use the VFS, the next step is to compile the HTML files into the data structures and strings that will be the actual VFS files. If you are working solely on a native file system, you should skip this step.

3.3.1 Building the HTML Compiler

The first step is to get the HTML Compiler working on your development system. The compiler source is a single C source file, with a few standard includes, and a few includes from the ARM Webserver package. It can be built either by using the supplied makefile, or by building the `htmlcomp.mcp` file with ADS 1.0.1.

3.3.2 Running the HTML Compiler

The exact use of the HTML Compiler is detailed in Chapter 4 *Using the HTML Compiler*. To build an initial prototype webserver with only a few pages, you can use the compiler's default settings. To run the compiler:

1. Create an *input* file with the names of your HTML files. This is a simple text file with one file name per line. You can create it with any plain text editor, such as vi, Notepad, or ADS 1.0.1. You must also include the names of any `.gif`, `.jpeg`, `.map`, or Java bytecode files you want in your VFS. Any kind of file can be included here. If the HTML Compiler does not understand the type (as indicated by the file extension), it will simply encode a binary image of the file.

2. Run the HTML Compiler. Use the `-i` option to give it the name of your input file. If your input file name is `htmllist.vfs`, for example, the correct syntax would be:

   ```
   armsd -armul -exec htmlcomp.axf -i htmllist.vfs
   ```

   The HTML Compiler, in this case, will create a file named `htmldata.c`, which 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 *Provide the system routines* on page 3-7. If the file is too big (for example, it may slow the compiler or exceed the maximum file size for your editor), refer to *Usage* on page 4-3 for information on the `-o` option that can be used to provide multiple, smaller files.

3.4 Provide the system routines

At this stage in the port procedure, you must provide the *glue* routines to initialize the ARM Webserver, and give it access to the networking and memory resources. The sections below describe the routines needed, and provide examples and suggestions:

- *Dynamic memory*
- *TCP/IP and sockets*
- *String library* on page 3-8
- *Timer tick routine* on page 3-8.

In all cases, examples are available in the demonstration program.

3.4.1 Dynamic memory

All the dynamic memory on the ARM Webserver is allocated by calls to `npalloc()`, and released by calls to `npfree()`. The syntax for these is exactly the same as the standard C library calls `malloc()` and `free()`, with the exception that the memory returned by `npalloc()` is assumed to be pre-initialized to all zeros. In this respect, `npalloc()` is like `calloc()`.

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

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

If your system does not support `calloc()` and `free()`, you will need to implement them. A description of how these functions work and sample code is available in *The C Programming Language* by Kernighan and Ritchie.

3.4.2 TCP/IP and sockets

The only socket calls made directly from the HTTP engine are `recv()`, `send()`, `close()`, and `errno()`. These are coded as `sys_recv()`, `sys_send()`, `sys_closesocket()`, and `sys_errno()` in the C sources. Because the calling semantics of the `sys_` commands are identical to the originals, the defines in `webport.h` map directly to the ARM TCP/IP sockets interface, as shown in Example 3-1 on page 3-8.

Caution

All sockets for the ARM Webserver must be opened in nonblocking mode, as the logic around the `sys_recv()` call assumes that it returns immediately whether data is ready or not. The ARM Webserver code assumes that `sys_recv()` returns an error if no data is available, and that in this situation, `sys_errno()` returns the value `SYS_EWOULDBLOCK`.

Example 3-1

```
#include “sockets.h”
/* required includes will vary from system to system */

#define sys_recv(sock, buf, len, flags) t_recv(sock, buf, len, flags)
#define sys_send(sock, buf, len, flags) t_send(sock, buf, len, flags)
#define sys_closesocket(sock)           t_socketclose(sock)
#define sys_errno(sock)                 t_errno(sock)
#define SYS_EWOULDBLOCK                 EWOULDBLOCK
```

3.4.3 String library

The ARM Webserver expects the C compiler library to provide the routines it uses for string manipulation. Most of these are standard routines defined in ANSI C:

- `sprintf()`
- `strcat()`
- `strchr()`
- `strcmp()`
- `strcpy()`
- `stricmp()`
- `stristr()`
- `strncmp()`
- `strnicmp()`
- `strstr()`.

All these functions, except `stricmp()`, `stristr()`, and `strnicmp()`, are already available in the standard runtime libraries. These functions are available in the `misclib\strilib.c` file.

3.4.4 Timer tick routine

The timer tick routine is a function that needs to be called periodically to perform further work on HTTP connections that could not previously complete. This can happen when a large HTML file is sent on a TCP/IP connection with limited resources. The whole file cannot be sent at once because the networking layer does not have enough buffer space to store it while the network moves it.

The ARM Webserver does a reasonable amount of work each time it assumes control, then returns to the system. In these cases, the `httpd` structure (and its submembers) will be left filled in with state data. This allows the work to resume when the timer tick routine is next called.

Both web performance and system efficiency must be considered when determining the frequency with which the timer should be called. For embedded systems that use the web pages for status monitoring and configuration, ten times per second is suggested.

The timer tick routine to be called is `http_loop()`. It is called with no parameters. It returns the number of connections that had work pending. The demonstration program ignores this return. However, in a system where CPU cycles are a critical resource, this could be used to reduce the calling frequency when little work is pending (that is, `http_loop() == 0`), and increase it when several connections are pending.

3.5 Initialization routine

The ARM Webserver code itself needs no internal setup. The C compiler will initialize all the internal lists and structures to the appropriate values. However, there are one or two initialization activities that must occur on the target system before it will invoke the webserver:

- the initiation of a socket `listen` for webserver operation
- optionally, the initialization of a user-extended VFS files system.

In the demonstration program, these activities are both addressed in a single function, the initialization routine `http_init()`, which can be found in the `httpport.c` file.

To set up the TCP/IP `listen`, you only need to copy the code from the demonstration program `http_init()` function into your own file.

A mechanism is needed to sense when an incoming TCP/IP connection is accepted, and tell the webserver. In the example port, a simple polling routine checks the TCP/IP `listen` every clock tick to see if a connection has formed. If it has, the socket is passed to the webserver by way of `http_connection()`. The syntax for this call is:

```
void http_connection(long sock);
```

The webserver logic will call `sys_closesocket()` to close the socket when it is no longer needed.

In a more sophisticated system (that is, one with UNIX-like signals) the overhead of polling can usually be avoided by signaling when the TCP/IP connection is ready, and then calling the `http_connection()` function. Refer to your system documentation for details.

3.6 User and password lookup routine

Embedded systems in business environments usually have some configurable features that must be accessible to authorized users only. For example, not every employee should be able to change the IP address of a networked printer. HTTP supports this type of protection, using the UNIX-like user and password principle. Any HTML page can be flagged as part of a protected *realm* (a list of pages). Accessing pages in a realm requires the user to enter a valid user name and password at the browser. On an embedded system, the form to set the IP address would usually have this type of protection.

The following steps take place when a user first tries to access a protected form:

1. The browser performs a standard HTTP `GET` request on the HTML page containing the form.
2. The server realizes that the page is protected, and scans the HTTP header for an appropriate *authorization* field, which should contain an encoded user name and password.
3. If this field is missing, which is usually the case on the first request, the server sends an `HTTP error 402 (authorization failure)` back to the browser.
4. The browser displays a popup window to the user, asking for the name and password.
5. The user enters this information, and the browser resends the HTTP `GET` request, this time with the authorization field.
6. The server decodes and checks the name and password, and, if everything checks out, returns the protected HTML page.

To use this facility on the ARM Webserver, you must do two things:

- flag the protected files, so the webserver knows when to require authorization
- provide a routine to verify user names and passwords.

Files can be flagged as protected by adding either the `-a` or `-5` option to their entry in the HTML Compiler input file. The routine to verify user names can perform any type of verification you desire. The syntax for the routine is shown in Example 3-1 on page 3-12.

Example 3-1

```
/* user authorization routine */

int
user_ok(char *user,           /* decoded name of user from browser's HTTP header*/
        char *password,       /* decoded password from browser */
        struct vfs_file *vfp) /* protected file's entry */
{
....
```

The user name and password are passed in plain text because they have already been decoded. If organized realms are desired, the user code is responsible for initializing and maintaining them. The `vfs_file` structure has a `name` member to assist with this.

Note

The name of this routine must be `user_ok()`, and it must be provided if any files are flagged as protected.

For most embedded systems, a single user name and password for system privileges is stored in *Non-Volatile Random Access Memory* (NVRAM). Checking the passed parameters against these and returning nonzero (if access is permitted) or zero is usually sufficient.

3.7 SSI routines

There are two types of SSI commands supported by the ARM Webserver:

- `#include` commands
- `#exec` commands.

They are described in the following sections:

- *Including other files within a page*
- *Dynamic page content*
- *SSI exec routine* on page 3-14
- *html_exec* on page 3-14
- *SSI include routines* on page 3-16
- *Displaying C variables using #include* on page 3-17.

3.7.1 Including other files within a page

If your web pages all have the same header and trailer components, they can be stored more efficiently by separating out the common text areas into individual files, such as `header.htm` and `trailer.htm`. This allows you to keep just one copy of the standard header and trailer, and include them in each of your web pages with a command similar to:

```
<!--#include file="header.htm"-->
<P>This is the body of the webpage</P>
<!--#include file="trailer.htm"-->
```

When the ARM Webserver sends this document to the browser, it will replace the entire HTML comment (`<!--` to `-->`) with the contents of the corresponding file.

In addition to offering a significant space saving, this also allows you to easily change the look of your web pages by simply editing the header page. For example, you can change the background color or text font in the header page, and this affects the appearance of all pages of your system.

3.7.2 Dynamic page content

You may also want to have dynamic data within your web page, and there are multiple ways of achieving this. You can either use the CGI functionality of the webserver to generate the entire page dynamically (see *CGI routines* on page 3-18), or you can use the `#exec` or `#include` SSI commands.

3.7.3 SSI exec routine

The SSI *exec* feature is useful when you want to include some information that may change in a web page. An example of this is the port status page in the demonstration program (see *Demonstration program* on page 1-3). The port status can change each time it is read. Simply typing the numbers into an HTML file does not work. Current information must be inserted into the HTML file as it is read. This is why SSI exec is used.

The syntax of SSI exec in an HTML file is:

```
<!--#exec cmd_argument="text_string"-->
```

When a webserver with SSI support encounters this tag in the file, it passes the string in quotes to the system for execution. The execution is expected to result in some HTML text that can be inserted in the data stream to replace the text of the SSI tag.

3.7.4 html_exec

The SSI exec functions on the ARM Webserver are all directed to a single user-supplied function, `html_exec()`.

Syntax

```
int html_exec(struct httpd *hp, char *args)
```

where:

hp Is a pointer to the ARM Webserver state structure for the current connection.

args Points to the string specified as the `cmd_argument` in the SSI exec tag.

Return value

Returns one of the following:

0 The command was understood by the command parser. This does not necessarily mean the command succeeded. It only means that no internal parse error occurred.

nonzero The command was not understood. This means there is a definite problem, and a server error code is returned to the browser.

Usage

Usually `html_exec()` uses `sys_send()` to insert the data, as shown in Example 3-1

Example 3-1

```
/*
 * html_exec() - execute a command from an HTML file on the
 * local system
 *
 * This example parses just one HTML SSI command, 'portstat',
 * which is used to fill in checkboxes within a FORM page. If
 * the current status for a port is 'PORT_OK', the corresponding
 * checkbox within the form will be checked.
 */
int html_exec(struct httpd *hp, char *args)
{
    int port;

    /* check this is a 'portstat' command */
    if(strncmp(args, "portstat", 8) != 0)
    {
        /* programmer error! */
        dtrap();
        return -1;
    }

    /* extract port number */
    port = atoi( nextarg(args) );

    /* check port number is in valid range */
    if(port < 1 || port > 8)
    {
        /* programmer error! */
        dtrap();
        return -1;
    }

    /* if this port is enabled, send 'CHECKED' string */
    if( portstat[port-1] == PORT_OK )
        sys_send(hp->sock, "CHECKED", 7, 0);

    return 0;
}
```

The `html_exec()` routine is expected to parse the string passed as *args*, and to insert some text into the socket given by `hp->sock`.

3.7.5 SSI include routines

The ARM Webserver supports another method of using SSI to return dynamic data. The syntax in HTML is identical to standard SSI includes:

```
<!--#include file="ipaddr"-->
```

In this case, however, the named file `ipaddr` is a special VFS *executable* file. It has no data, just a pointer to a user-supplied routine to be called whenever it is invoked. The routine, similar to `html_exec()` above, should just send some text to the socket and return. In Example 3-2, the routine formats the current IP address of the system into text and uses `sys_send()` to write it to the socket.

Example 3-2

```
/*
 * ht_ipaddr() - SSI dynamic include routine
 *
 * Returns 0 if everything was OK. Otherwise, returns a non-zero error.
 *
 * This example converts the primary interface's IP address to a text string
 * and sends it to the browser.
 */
int ht_ipaddr(const struct vfs_file *vfp,  /* pointer to vfs file */
              struct httpd *hp,            /* HTTP connection */
              char *args)                  /* args, text separated */
                                           /* by spaces */
{
    char * cp = print_ipad(nvparms.ifs[0].ipaddr );
                                           /* format IP address */
    sys_send(hp->sock, cp, strlen(cp), 0); /* send to socket */

    return 0;
}
```

The advantage of this approach over using `html_exec()` is that the VFS executable file and the C code routine *stub* can be generated by the HTML Compiler. For compiler usage details, see Chapter 4 *Using the HTML Compiler*.

The name of the C function for your SSI will vary depending on the parameters given to the HTML Compiler. In this case, the function name is `ht_ipaddr()`, which is the function for the `ipaddr` example in the demonstration program.

Syntax

```
int ht_ipaddr(struct vfs_file *vfp, struct httpd *hp, char *args)
```

where:

vfp Is the VFS file entry for the included file (in this case, "ipaddr"). This might not be needed, but is sometimes useful.

hp Is the connection on which the files are being sent out.

This is required because hp->sock is the socket ID to be used with sys_send().

args Is the text in quotes in the SSI tag, as in "ipaddr" (see *SSI include routines* on page 3-16). In more complex applications, this string can be used to pass parameters.

For example, a multihomed host can use "ipaddr iface1" and "ipaddr iface2" to return the IP addresses of different interfaces. All calls still go to the routine indicated by the VFS entry for ipaddr, and the routine can use the *args* parameter to determine the interface whose IP address was being requested.

3.7.6 Displaying C variables using #include

A common task for webservers that manage embedded devices is the displaying of dynamic data. In the demonstration program, examples of this are the various counters displayed in the hub status page, such as the number of frames sent, number of errors, and so on. These values can change every time the page is redisplayed. Therefore, you must have access to the value maintained in the device software, format it into HTML, and insert it into the web page.

One technique for accomplishing this is described in *SSI include routines* on page 3-16. If the values you want to insert into your web pages correspond directly to C variables within your program, you can use cvar files within the VFS, which is an easier technique. This allows the HTML Compiler to generate the necessary code to handle formatting of data and sending it to the browser. See *Usage* on page 4-3 for details on how to make the HTML Compiler generate cvar SSIs.

3.8 CGI routines

As explained in *Common Gateway Interface* on page 2-5, the CGI capabilities of a conventional webserver are usually oriented toward executing a separate process and returning the results. Because many embedded systems have limited multi-process capabilities, but good C code capabilities, the ARM Webserver CGI is oriented toward calling a C function. This C function can perform a variety of tasks (within system limits), but the calling interface is designed with the assumption that it will be processing the results of an HTML form:

- *CGI routine inputs*
- *CGI routine return values* on page 3-19
- *Using the HTML Compiler to automatically generate CGI functions* on page 3-21.

3.8.1 CGI routine inputs

Example 3-3 shows the semantics of the CGI calls:

Example 3-3

```
int setip_cgi(struct httpd *hp,          /* connection on which GET or POST came in */
              struct httpform *form,     /* a list of name-value pairs */
              char **filetext            /* optional text or file to return to the */
                                         /* Web browser */
                      )
```

The input to these routines is fairly simple, assuming you are familiar with HTML forms. Before you attempt to code a CGI routine, you should be familiar with the use of name/value pairs from a form. 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; /* the VFS file name */
       int    nv_ct;   /* number of entries in name_val table */
       struct name_val nameval[1];
                       /* the first of an array of name-value */
                       /* entries */
};
```


There are actually 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 by way of `npfree()` after the CGI routine returns, so 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 to the file name, which can be useful if you want to write a CGI routine that is invoked by more than one form. The next member is the number of name/value structures that follow. The rest of the structure is an array of the name/value structures. Although this array is declared (in the structure definition) to contain only one element, the actual structure passed to your CGI routine has been dynamically allocated by `npalloc()`, and the `nameval[]` array contains `nv_ct` entries.

The name/value structures are filled in the order they were received from the browser. Experience shows this has always been the order in which they appeared in the form. However, the HTTP specifications allows them to be in any order. You should check the names before parsing the values.

The CGI routine may block while it is parsing data, but caution should be taken so that calls to `http_loop()` do not cease. This would stop your webserver from servicing other blocked connections until the CGI routine returns. The ARM Webserver logic keeps a flag in the `httpd` structure, which lets it know that the connection is blocked in a CGI routine, so it will not call the CGI routine recursively for the same connection and overflow your stack.

3.8.2 CGI routine return values

The CGI routine return values (a bitmask and a string) are complex. Once the browser has sent a form to the server, it is expecting an HTML page in response. This form reply page can be anything from a simple *thanks for filling in our form* type of message, to the most complex page in the system. The ARM Webserver provides a simple generic form response page that you can use without having to write any code or include any additional HTML pages. This allows you to have many forms and still keep memory usage down. However, some CGI routines may need more control over the content of their form replies than the generic reply page provides. These have the options of building and sending their own HTML text replies, or passing the name of a reply file back to the webserver code.

The return value is a bitmask of the following bit values:

```
FP_OK    /* form processing was OK if set, else system error */
FP_FILE  /* filetext points to a file reply */
FP_TEXT  /* filetext points to an error or status text */
```



```
FP_ERRHD /* insert an error header line in the reply page */
FP_OKHD  /* insert an "OK" header line in the reply page */
FP_DONE  /* CGI routine did everything */
```

The hp parameter is passed to the CGI function so it can use hp->sock to write its own form reply. In this case, it should return the predefined value FP_DONE. This overrides all the other return bit values and causes the ARM Webserver to simply close the connection and free resources.

If FP_DONE is not the sole bit returned, then FP_OK should be set. This bit indicates that the CGI routine did *not* experience any catastrophic failure. If this bit is not set, an HTTP *server error* packet is returned, and the end user at the browser will not know exactly what has gone wrong. The FP_OK bit does not imply that the form data was correct or parsable. Bad form data is indicated by returning the FP_ERRHD bit (see below).

Another way to ensure that the CGI routine controls its reply is to modify it to set the passed filetext parameter to a file name, set the FP_FILE bit in the return value, and return. The ARM Webserver will send the named file to the browser as a reply.

Most forms do not require such detail. Generally, the webserver CGI routines will return a bit mask consisting of:

- FP_OK
- possibly FP_TEXT
- either FP_ERRHD or FP_OKHD.

If FP_ERRHD is set, a generic *form error* reply page is prepared. If FP_OKHD is set, a generic *form OK* page is prepared. If neither is set, a very plain form reply page is prepared, which implies neither error nor success. If FP_TEXT is set, then filetext is assumed to point to a string that is inserted in the reply page. This text can be inserted regardless of the state of FP_ERRHD and FP_OKHD.

3.8.3 Using the HTML Compiler to automatically generate CGI functions

The previous sections describe, in some detail, how CGI functions are called, and the values they are expected to return. In practice, CGI functions are mostly used for parsing the values that the end user has entered into a form. Therefore, the HTML Compiler has extra functionality that makes this process much easier.

The HTML Compiler can be instructed to generate local variable declarations and function calls from within the CGI routine to parse values from the form. You still have to add some code to the CGI function to perform range checking, and to actually use the values obtained, but the overall workload is greatly reduced. Refer to *Usage* on page 4-3 for more details.

It is recommended that you refer to the CGI routines in the demonstration program (see *Demonstration program* on page 1-3), and experiment with your own CGI routines. Clear form replies can be produced without writing a lot of code or HTML pages.

Chapter 4
Using the HTML Compiler

This chapter describes how to use the HTML Compiler. This is a program that takes your web pages and compiles them into C structures which become VFS files in your embedded server. It contains the following sections:

4.1 About the HTML Compiler

The HTML Compiler is a program that creates C structures from your web page files. These C structures, in turn, become VFS files in your embedded server. The web page files can be any file you would normally put on a webserver. Files of extension `.html` (or `.htm`) and `.gif` are most common, but the compiler can accept any binary or text file.

In addition to simply encoding binary files into C tables, the HTML Compiler supports some optional features that are useful in building even a modest webserver. These include:

- tag compression of HTML files
- setting authorization flags on files
- creating the routine stubs and prototypes for CGI and SSI executable files.

For details on how to build the HTML Compiler, see *Building the HTML Compiler* on page 3-6.

4.2 Usage

The HTML Compiler is designed to be invoked from a *Makefile* as part of an automated code building system. This allows your web page files to be listed as dependencies for your firmware image. The invoking syntax from a DOS or UNIX command line is:

```
armsd -armul -exec htmlcomp.axf -i htmllist.vfs [-options]
```

The input file, `htmllist.vfs` in this case, is a text file containing a list of files that become part of the VFS of the embedded ARM Webserver. The format is one file name per line, possibly followed by options for that file compilation.

The runtime options in the command line can change default values for each of the optional features of the HTML Compiler. Runtime options can also:

- rename output *C code* files
- change VFS file names
- create multiple output files.

Many runtime options can be set on the command line which invokes the HTML Compiler. Some can also be set on a line in the input file. When set on the command line, an option applies to all files in the input file. An option set in the input file affects only the web page file named on that line.

4.2.1 Command lines

The following are the options that can be used on the command line, along with a brief description of what each one does, and what the defaults are:

`-a`	Require *Basic* (uuencoded) authentication on `GET`s.
`-5`	Require *Digest* (MD5) authentication on `GET`s.
`-c`	Omit HTML file tag compression. Default is to compress.
`-h` *`filename`*	Rename generated include file. Default is `htmldata.h`
`-f`	Omit `vfs_files` array. Default is to create it.
`-m`	Omit map files structure. Default is to create it.
`-n` *`vfsname`*	Change name of `vfs_files` array. Default is `vfs_files[]`.
`-o` *`outfile`*	Rename output file to something other than `htmldata.c`.
`-v`	Toggle verbose mode. Default is ON.

4.2.2 Input lines

As well as `-o`, `-a`, `-5`, `-c`, and `-m`, the following options can also appear on lines in the `.vfs` compiler input file:

`-s` Generates an SSI executable function stub.

`-x` Generates a CGI function stub.

`-cvar` Generates an SSI C variable entry.

Note

When used this way, these options apply *only* to the file named on that line.

4.3 Sample input file

This section shows part of the input file from the demonstration program. After Example 4-1, several of the options are explained in more detail, in the sections:

- *The -a option* on page 4-6
- *The -o option* on page 4-6
- *The -s option* on page 4-6
- *Embedding C variables in a web page (-cvar option)* on page 4-6
- *The -x, -form and -endform options* on page 4-8
- *The setip.cgi option* on page 4-8
- *Generating stub routines* on page 4-10.

4.3.1 Code sample

Example 4-1 shows part of the input file from the demonstration program.

Example 4-1

```
btnmap.map
hlpmask.htm
setip.htm -a
index.htm
body.htm
inworks.htm
helphlp.htm
hlpipadd.htm
statmenu.htm
ptstats.htm
helpbtn.gif -o gifdata.c
nplogot.gif -o gifdata.c
hub4907.gif -o gifdata.c
ipmask -s ht_ipmask
defgw -s ht_defgw
ipaddr -cvar ip_addr nvparms.ifs[0].ipaddr HT_IP_ADDRESS
setip.cgi -x setip_cgi
-form
    bool      VJcompress
    int       ppp_trace
    u_long    ppp_tmo
    unshort   ppp_keep
-endform
setport.cgi -x setport_cgi
```

4.3.2 The -a option

The third line of the code sample illustrates the `-a` option:

```
setip.htm -a
```

When the ARM Webserver is used, the `setip.htm` file requires Basic authentication. Replacing the `-a` with `-5` forces the use of Digest authentication, as described in *Users, authentication, and security* on page 2-9.

4.3.3 The -o option

The three `.gif` file lines illustrate the `-o` option:

```
helpbtn.gif -o gifdata.c
nplogot.gif -o gifdata.c
hub4907.gif -o gifdata.c
```

This specifies where to send the C data output from the compiler. In this case, the binary data for the three files is placed in the file `gifdata.c`, rather than the default file `htmldata.c`.

4.3.4 The -s option

The files with `-s` options generate C code stubs for SSI routines:

```
ipmask -s ht_ipmask
defgw -s ht_defgw
```

Any real file with these names is ignored. The tags following the `-s` are the names that are given to the SSI stub routines. See *SSI include routines* on page 3-16 for more information about SSI `#include` routines.

4.3.5 Embedding C variables in a web page (-cvar option)

The file with the `-cvar` option illustrates how to embed C variables in a web page:

```
ipaddr -cvar ip_addr nvparms.ifs[0].ipaddr HT_IP_ADDRESS
```

If a web page contains a line such as:

```
<!--#include "ipaddr"-->
```

the ARM Webserver sends the contents of the variable *`nvparms.ifs[0].ipaddr`*, formatted as a dotted-quad IP address, to the browser.

The cvar example above, used to send the IP address of the primary network interface, contains five fields that direct the code generation. The fields in the above example are:

ipaddr The name of the SSI file created in the VFS.

-cvar The option that causes the HTML Compiler to generate full mapping of a C variable to the HTML output.

ip_addr The C variable type to use. The valid variable types are:

short	A signed 16-bit value (%d).
unshort	An unsigned 16-bit value (%u).
u_short	An unsigned 16-bit value (%u).
long	A signed 32-bit value (%ld).
int	A signed 32-bit value (%d).
u_long	An unsigned 32-bit value (%lu).
unsigned	An unsigned 32-bit value (%u).
char*	A null-terminated string (%s).
ip_addr	A *dotted quad* IP address.
bool	A Boolean value. If nonzero, the output is the literal string CHECKED for use in checkboxes. Produces no output if the value is zero.

nvparms.ifs[0].ipaddr

The name of the C language variable you want to appear in the HTML output. The variable is cast to the type below, so you must make sure they are compatible.

HT_IP_ADDRESS

A name for a #define that is generated in the HTML Compiler output files to identify this variable. This string must be unique, and must not clash with any other #define in the system.

The C code generated in the default output file to handle cvar values is a single routine named ht_ssi_cvar(). It is inside an undefined #ifdef section. You must copy it to a local *per-port* file (for example, httpport.c) to compile it. In this way, any changes you make to ht_ssi_cvar() are not lost when you re-run the compiler.

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. See *Displaying C variables using #include* on page 3-17 for more information about SSI #include variables.

4.3.6 The -x, -form and -endform options

The -x option is similar to -s, except it generates a CGI routine stub instead of an SSI routine stub:

```
setip.cgi -x setip_cgi
```

When used on its own, the -x option simply produces an empty CGI routine stub, as described in *CGI routines* on page 3-18. You must then finish the routine to provide the necessary functionality. However, if your CGI function is going to be parsing the results of a form GET or POST response (which is usually the case), the HTML Compiler can do most of the parsing work for you. To do this, the -x option is supplemented by two further directives:

- -form
- -endform.

These are placed as a pair around a series of data lines following a -x line in the htmllist.vfs input file:

```
-form
    bool      VJcompress
    int       ppp_trace
    u_long    ppp_tmo
    unshort   ppp_keep
-endform
```

Any number of lines can appear between the pair, each defining and naming a local C variable to be declared and set from the passed form data. Each data line has two fields:

C type A list of usable C types is the same as for the -cvar option, defined above.

Name A name is used for both the C variable *and* the HTML name/value pair name, so it must be of legal syntax for both.

4.3.7 The setip.cgi option

The setip.cgi entry from Example 4-1 on page 4-5 generates a VFS file named setip.cgi and maps it to a code routine named setip_cgi():

```
setip.cgi -x setip_cgi
```

When the end user submits a form with an action of setip.cgi, the ARM Webserver code parses the name/value pairs from the form into a table and passes them to the generated routine setip_cgi().

The HTML Compiler automatically generates the following code inside the setip_cgi() routine, as shown in Example 4-2 on page 4-9.

Example 4-2

```
/* setip_cgi() CGI routine */
/* Returns bitmask (see section <Undefined Cross-Reference>) */
{
int
setip_cgi(struct httpd *hp,         /* http connection */
          struct httpform *form,    /* filled in form data from browser */
          char ** filetext          /* filename or text for return */
{
int     err;                        /* numeric parse errors */
int     VJcompress;                 /* to be set from form */
int     ppp_trace;                  /* to be set from form */
u_long  ppp_tmo;                    /* to be set from form */
unshort ppp_keep;                   /* to be set from form */

    /* Auto-generated code: */

    VJcompress = get_form_bool(form, "VJcompress");
    err = get_form_int(form, "ppp_trace", (int*)&ppp_trace);
    if(err)
    {
            *filetext = "bad ppp_trace";
            return (FP_OK|FP_TEXT|FP_ERRHD);
    }
    err = get_form_long(form, "ppp_tmo", (long*)&ppp_tmo);
    if(err)
    {
        *filetext = "bad ppp_tmo";
        return (FP_OK|FP_TEXT|FP_ERRHD);
    }
    err = get_form_short(form, "ppp_keep", (short*)&ppp_keep);
    if(err)
    {
        *filetext = "bad ppp_keep";
        return (FP_OK|FP_TEXT|FP_ERRHD);
    }
    return (FP_OK|FP_OKHD); /* OK return, no text */
}
```

At this point, you must add code to perform range checking on the values in the local variables created by the HTML Compiler, and to use the values (for example, setting new operating parameters for the TCP/IP stack or recording user preferences).

As with the `-cvar` option, you must move the generated code out of your output file, and you must link in `httplib.c` in to provide the required routines.

4.3.8 Generating stub routines

Generating the VFS file entries and function stubs for the code which is executed by SSI and CGI commands is a time-consuming task, with great potential for error. Therefore, the HTML Compiler does it for you. The last eleven lines of Example 4-1 on page 4-5 illustrate this.

The first of these eleven lines, the `ipmask` file, is an SSI entry that, when included in an SSI command such as:

```
<!--#include file="ipmask"-->
```

inserts the currently-set IP mask into the HTML document.

To perform this, a file named `ipmask` must exist in the VFS, and this VFS entry must have a pointer to a routine that sends the current IP mask to the data stream. By adding the `-s` option to the `ipmask` input file line, the HTML Compiler is informed that `ipmask` is an SSI executable file, rather than a data file. The compiler then:

- generates the function prototype
- generates a stub of the routine named on the input file line
- makes a VFS entry with a pointer to the SSI routine.

The generated function prototype is placed in the include file output from the compiler (`htmldata.h` by default). The actual routine stubs are placed in the default output C code file (`htmldata.c` by default), in a section which is `#ifdef`ed out.

Note

You must copy the stub routines into a port-dependent file and edit them there. Otherwise, they are overwritten the next time you run the compiler.

In the demonstration program, these routines have been moved to `httpport.c`.

The CGI routines and `cvar` formatting are handled in a manner similar to the SSI routines. Prototypes are placed in the default output include file, and routine stubs are generated in the same `#ifdef`ed out section of the default output C file. The function parameters for CGI and SSI routines are different, so be careful not to confuse the two options in the input file.

Appendix A
Building the Demonstration Program

This appendix details the requirements, installation procedure, and steps required to build the demonstration program. Instructions are provided for using both ADS for Windows and ADS for command-line environments.

This chapter contains the following sections:

A.1 Requirements

The following products are required to build the demonstration program:

- ARM TCP/IP software (AS301)
- ARM Webserver (AS303)
- ARM Integrator/AP, fitted with a suitable core module, for example, Integrator/CM7TDMI, with Ethernet Kit
- ARM Ethernet kit for Integrator/AP
- ARM Developer Suite, 1.0.1
- ARM Multi-ICE version 1.4.

Note

The Integrator/AP must have a system controller FPGA that is Rev A, Version 53 or newer.

The Integrator/CM7TDMI must have an FGPA that is Rev A, Version 60, or newer.

A.1.1 FPGA product information

You can find out information on the FPGA release used on your Integrator/AP or core module using the ARM bootPROM firmware on the Integrator:

1. From the boot monitor prompt, type `x` to enter board-specific command mode:

   ```
   > x
   ```

2. Display the hardware configuration information by typing `dh` at the prompt:

   ```
   > dh
   ```

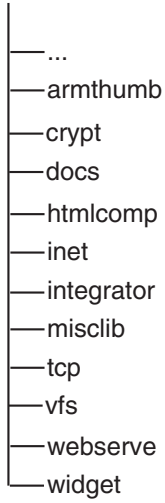
Upgrade information

If you need to upgrade your integrator/AP and Integrator/CM7TDMI FPGAs, please contact `support@arm.com`.

A.2 Installation procedure

The following steps give an overview of the installation procedure:

1. Check that you can use ADS to compile and run programs on the ARM Integrator board.
2. Install the ARM TCP/IP software by following the detailed instructions in the *Porting TCP/IP Programmer's Guide*. This is provided with the ARM TCP/IP software.
3. Unpack the ARM Webserver software into the same directory you used for the ARM TCP/IP sources. You should now have a directory structure similar to the following:



Note

There might be other directories for PPP and other ARM networking protocols that you have purchased.

A.3 Building using ADS for Windows

This section details the steps necessary to build the HTML Compiler and the *widget* demonstration application using ADS in a Windows environment.

A.3.1 Build the HTML Compiler

1. Copy the `htmlcomp.mcp` and `htmlcomp.bat` files from the `htmlcomp` directory into the project directory:

   ```
   C:\...> cd widget
   C:\...\widget> copy ..\htmlcomp\htmlcomp.mcp
   C:\...\widget> copy ..\htmlcomp\htmlcomp.bat
   ```

2. Select **Open** from the CodeWarrior **File** menu, and open `htlmcomp.mcp` from the `widget` directory.

3. Select **Make** from the **Project** menu.

   The HTML Compiler should compile and link without errors or warnings.

A.3.2 Compile the sample HTML files

1. At the command line, run `htmlcomp.bat` in the `widget` directory:

   ```
   C:\...\widget> htmlcomp.bat
   ```

   This invokes armsd to run the HTML compiler. You should see output similar to that shown in Example A-1 on page A-5.

A.3.3 Build the Widget application

1. Select **Open** from the CodeWarrior **File** menu, and open `widget.mcp` from the `widget` directory.

2. Select the appropriate target (for example, `EtherSuperloop`) from the drop-down list in the `widget.mcp` project window in CodeWarrior.

3. Select **Make** from the **Project** menu.

   Widget compiles and links without errors or warnings.

Example A-1 Sample htmlcomp output

```
------------ Starting section 'GIF images'
Building "C" vfile data array for helpbtn.gif    1197 bytes {12399, 5713}
Building "C" vfile data array for nplogot.gif    1781 bytes {14180, 7494}
Building "C" vfile data array for hub4907.gif    6864 bytes {21044,14358}
Building "C" vfile data array for btnmap.gif     3943 bytes {24987,18301}
------------ Section file size: 13785


------------ Starting section 'server-side include (SSI) routines'
------------ Section file size:     0


------------ Starting section 'CGI routines'
------------ Section file size:     0

Building MAP structure from btnmap.map


Total file size: 24987 (compressed to 18301)

Program terminated normally at PC = 0x0000f520 (_sys_exit + 0x10)
+0010 0x0000f520: 0xef123456  V4.. :    swi      0x123456
Quitting
Press any key to continue . . .
```

A.3.4 Run the Widget application

1. Edit the `ether.nv` file and set valid IP addressing options (see the *Porting TCP/IP Programmer's Guide*).
2. Ensure that the MultiIce server has been started and configured.
3. Select **Run** from the CodeWarrior **Project** menu.
4. Use your preferred web browser with a URL like `http://10.0.2.9/`, where the address used is the one that you specified in the `ether.nv` file.

A.4 Building using ADS from the command line

To build the HTML Compiler and the Widget application using ADS in a command-line environment (such as Solaris or a Windows Command prompt), you must do the following:

1. Change to the project directory, `widget`.
2. Run one of the following:

   `make -s` For Solaris.

   `nmake /s` For a Windows Command prompt.

   The Widget project should build without errors or warnings.
3. Edit the `ether.nv` file and set valid IP addressing options (see the *Porting TCP/IP Programmer's Guide*).
4. Run the `widget.axf` file on the ARM integrator board.
5. Use your preferred web browser with a URL like `http://10.0.2.9/`, where the address used is the one that you specified in the `ether.nv` file.

Glossary

ADS	ARM Developer Suite.
API	Applications Program Interface.
BSD	Berkeley System Distribution.
CGI	Common Gateway Interface.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
ICE	In-Circuit emulator.
MD5	Message Digest 5.
NVRAM	Non-Volatile Random Access Memory.
SSI	Server-Side Include.
TCP/IP	Transmission Control Protocol/Internet Protocol.
URL	Uniform Resource Locator.
VFS	Virtual File System.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

G

H

I

L

M

N

P

R

S

T

U

V

W

Symbols

