USNet®
Web Server
User's Guide

Version 1.3
December 2004

U S SOFTWARE®
EMBEDDED EXCELLENCE

Copyright and Trademark Information

Lantronix, Inc.
15353 Barranca Parkway
Irvine, CA 92618
(949)453-3990
Fax (949) 453-3995

For Support Contact:
Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxinfo.com
www.smxinfo.com

Documentation Conventions

Computer output and code examples: `Courier`, usually in a separate paragraph.

Function names and command names: ***Bold italic***, usually followed by parentheses, as in ***main()*** function.

Variables: Courier 11 italic (*`mt_busy`*).

File names: Times bold (the file **usrclk.asm**), usually in lower case.

Key names: Initial capital, in angle brackets, as in press <Enter>.

Menu names and selections, dialog box names, screen titles, window titles: Times bold, as in **File** menu**.**

Notes: Indicate important information.

Cautions: Indicate potential damage to hardware or data.

Documentation History

Revision Number	Date	Notes
1.0 (Original)	December 1997	
1.1	May 1998	
1.2	August 1998	
1.3	December 2004	Merged and eliminated Advanced Customization manual.

Contents

1. Introduction 1
User's Guide Overview 1
Web Server Terminology 2
Recommended Reading 3
Other U S Software Documents 3
On the Internet 3
Books 4

2. Getting Started 5
Web Server Overview 5
Web Server Requirements 6
Building the Web Server 7
Building for USNet 7
Building for UNIX 7
Building for Another TCP/IP Stack 8
Example Web Server 10
Building the Example Web Server for your Target 11
Connecting to the Example Web Server 11

3. Using the Web Server 13
User Server Functions 13
Bwrite() 14
GetEntry() 15
HTTPserver() 16
HTTPservinit() 16
Neof() 17
HTTP Server Request Structure 18
Modules and Handlers 20
Module Function Descriptions 21
MODchkaccess() 21
MODchkauth() 22
MODchkloc() 23
MODchktype() 23
MODgetuser() 24
MODlog() 25
MODtranslate() 25
Request Structure 26
Using Usbldpg 27
Server Configuration File 27
MIME Information 33
AddType Command 35
Page Configuration File 36
Variable Configuration File 38

CGI Function Programming Interface 39
System Support Routines 40
CGI Routines 45
CGI Environment Variables 52
USMETA Programming Interface 55
#echo 56
#exec 57
#include 58
#memory 59
#system 59

Index 61

1. Introduction

User's Guide Overview

This guide describes U S Software's USNet® Web Server. The included files and the functions that these files provide are detailed in the **readme.txt** file.

This is the organization of the guide:

Chapter	Contents
1. Introduction	Introduces the reader to the Web Server User's Guide, Web Server terminology, and recommended reading.
2. Getting Started	Provides an overview of the Web Server, instructions for building the Web Server, and information about the provided examples of the Web Server.
3. Using the Web Server	Provides information on design paradigms, configuring the Web Server, and building your web pages. Describes user server functions, CGI routines, and HTML META commands.

Web Server Terminology

CGI — Common Gateway Interface. CGI reads parameters from forms on the displayed web page to the server, so the server can display different pages depending on the user's actions.

DNS — Domain Name System, a mechanism that allows the IP address of a system in a TCP/IP network to be determined based on a name assigned to the system, or vice versa.

HTML META commands — Commands embedded in the HTML that return predefined system information to the user.

HTTP — Hypertext Transfer Protocol, a simple application- level protocol used to access hypermedia documents. The protocol is stateless and generic, which allows it to be used for many tasks.

ISMAP — An HTML tag which returns position coordinates within the page image.

MIME — Multipurpose Internet Mail Extensions, which defines how to encode and decode multipart messages and non-ASCII character sets.

POP — Post Office Protocol, a minor variation of SMTP that allows a client to retrieve mail from a remote server mailbox.

SMTP — Simple Mail Transfer Protocol, a protocol for transferring mail.

SVA — Server Variable Access, a mechanism for accessing static global variables within an embedded application via HTML.

TCP/IP — Transmission Control Protocol/Internet Protocol, a software protocol for communication between computers.

Recommended Reading

Other U S Software Documents

USNet User's Manual

On the Internet

RFCs (request for comments) are documents that are available over the Internet via anonymous FTP. The following references will provide more information on topics relevant to the Web Server:

Topic	RFC Numbers
SMTP	821, 822, 1869, and 2045
POP	1725
MIME	2045 through 2049
HTTP	2068
DNS	1034, 1982, 2065, 1876, 1101

Here is an abbreviated example FTP session:

```
% ftp ftp.rfc-editor.org
.
Name: anonymous
Password: <your email address>
.
ftp> cd in-notes
.
ftp> get rfc1122.txt
.
ftp> quit
```

Books

Foundations of WWW Programming with HTML & CGI
IDG Books
ISBN 1-56884-703-3

CGI Programming in C and Perl
Thomas Boutell
Addison Wesly
ISBN 0-201-42219-0

CGI Developers Guide
Eugene Eric Kim
Sams Net
ISBN 1-57521-087-8

There are many books on web page design. This one is very good for low-level protocols, and has cross-references to RFCs:

Internet Protocols Handbook
Dave Roberts
Coriolis Group Books
ISBN 1-883577-88-8

2. Getting Started

Web Server Overview

The USNet® Web Server provides an HTML server framework with default modules, handlers, a server configuration file, and the **usbldpg** utility to compile HTML. It also includes CGI system support routines and the USMETA programming interface. The developer does not have to create their own Web Server API, and the Web Server is customizable.

The USNet Web Server supports any MIME file type that can be manipulated or displayed by your web browser. This includes audio and Java. The MIME types determine how the browser processes the information.

All source code discussed in this chapter is supplied with the USNet Web Server unless stated otherwise.

The USNet Web Server has a modular design, and can be easily modified to suit your application. Because existing web technology is page-oriented rather than object-oriented, full pages transfer from the server to the client. This limits the speed that data can be updated on the browser.

These are the general steps for creating and inserting web pages into the embedded Web Server:

1. Design and prototype your website using a standard web design tool (see *Recommended Reading* in Chapter 1).
2. Test your prototype HTML on any standard web server.
3. Move your prototype to the development system.
4. Change CGI programs to CGI functions (see *CGI Function Programming Interface* in Chapter 3).
5. Configure the Web Server to work with your network by modifying the configuration file (see *Server Configuration File* in Chapter 3).
6. Process your web pages through the **usbldpg** utility to obtain a C file that is compiled into the embedded format (see *Using Usbldpg* in Chapter 3).
7. Compile your application.
8. Test.

Though the USNet Web Server is designed to be user-customizable, it probably will not need customization. If you do want to customize, design information and guidelines for modifications are included in this document.

Web Server Requirements

System Requirements:

For a typical Web Server configuration, a minimum of 6K RAM (data and stack), and 30K ROM. Since the Web Server is modular these sizes may vary depending on the application, processor, and compiler.

NOTE: The Web Server uses the program stack to hold temporary data, so make sure there is at least a 5K stack in your application.

Tools required to build the Web Server:

USNet Web Server source, a compiler/linker for your target platform, and an editor.

Optional Tools:

A test Web Server for page design.

You can also use a web page design tool. Be sure that your tool produces only HTML without propriety extensions. Microsoft FrontPage contains proprietary extensions and will not work with the Web Server.

Building the Web Server

Instructions are provided for building the Web Server with USNet, for UNIX, and with another TCP/IP stack.

Building for USNet

After you install USNet:

1. Install USNet on your development system (see *Installing USNet* in the *USNet User's Guide*).
2. Use Opus **make** to build the sample USNet programs on your target, then test them. This is to verify that your target hardware is working properly before you incorporate the Web Server.
3. Install USNet IAP into the **IAPSRC** directory in the USNet source directory tree. The **install.bat** file provided on the distribution disk will copy the USNet IAP product files into the proper directories in the USNet development directories.
4. Edit **config.mak** in the root directory of USNet to include the USIAP library. This is accomplished by uncommenting the following line:

   ```
   #%set USIAP=usiap uscgi
   ```

   and commenting the line:

   ```
   %set USIAP=
   ```

   The makefile should now read:

   ```
   %set USIAP=usiap uscgi
   #set USIAP=
   ```

5. Build HTTEST from the root directory of USNet by typing:

   ```
   MAKE HTTEST
   ```

See also: *Example Web Server,* in this chapter, for more information on configuring the example Web Server for your target environment.

Building for UNIX

When building the source code on a UNIX platform:

1. Be sure you have the following lines in the **httpd.h** file:

   ```
   #undef USNET
   #undef LIKE
   #define UNIX
   ```

   These literals are defined or undefined within the first 10 lines of the file.

2. To make the web page compiler, **usbldpg**, change to the **usbldpg** directory and type:

```
make –f  makefile.unx
```

3. To precompile the web pages, change to the websrc directory and type:

```
..\usbldpg\usbldpg build.cfg ; cp htpgtbl.* ..
```

4. To build the Web Server library and the HTTEST sample program, change to the main directory and type:

```
make –f makefile.unx
```

NOTE: One issue you may notice when building the source code on a UNIX platform is that DOS is not case sensitive, and you may find some capitalized file names. The easiest way to fix this is to 'zip' the files on your DOS/WIN95 system, and then 'unzip' thefiles on your UNIX host using the –L option. The –L option will make file names lowercase.

See also: *Example Web Server,* in this chapter, for more information on configuring the example Web Server to your target environment.

Building for Another TCP/IP Stack

Install the USNet Web Server using the batch file provided on the disk. Be sure to specify the `–s` flag to indicate you will not be using USNet as your TCP/IP stack. The syntax is:

```
install –s <destination_dir>
```

Building a Web Server with a TCP/IP stack other than USNet requires a TCP/IP stack that supports BSD sockets. UNIX operating systems include a BSD socket library for TCP/IP.

To build the Web Server with another TCP/IP stack:

1. To accommodate your TCP/IP stack, you will need to make several modifications to **httpd.h,** found in the directory where you installed the Web Server. Be sure you have the following lines in the file:

```
#undef USNET
#undef LIKE
```

These literals are defined or undefined within the first 10 lines of the file.

2. You may want to define a literal that refers to your TCP/IP stack to enclose specific information about that stack. For example:

```
#define XYZNET /*Def for XYZ TCP/IP stack */
#ifdef XYZNET
/* Specify path to sockets header file */
#include <c:\XYZNET\INC\socket.h>
/* USNet uses Nprintf, printf in XYZNET */
#define Nprintf printf
#endif
```

3. The file **httpd.h** will need to include the sockets header file for your TCP/IP stack. For example, USNet has **socket.h**, while Linux uses **sys/socket.h**.

4. It is also possible that routine names will not exactly match those in your TCP/IP stack. Redefine function names as needed in **httpd.h**. As an example, please refer to the example on the previous page and to **httpd.h** to see what changes were made to build the Web Server under Linux.

5. The makefile, found in the directory where you installed the Web Server, will also require some modifications. Modify the linker command line to link in your TCP/IP library and any other support libraries required by your TCP/IP stack, compiler, and/or processor.

6. Once these changes have been made, build the sample Web Server by typing:

```
make HTTEST
```

See also: *Example Web Server,* in this chapter, for more information on configuring the example Web Server to your target environment.

Example Web Server

HTTEST is provided as a sample Web Server. Some of the terms listed below might be new (for definitions, see *Web Server Terminology* in Chapter 1). They will be discussed throughout the manual. The example is placed here to show the powerful features available in the Web Server.

There are six examples in the sample USNet Web Server, HTTEST. Each example demonstrates a different feature of the Web Server. These examples are links off of a starting page.

Example 0 transfers binary data to the browser in the form of GIF and JPEG pictures, and a JAVA applet.

Example 1 shows the CGI variables that are available.

Example 2 shows a CGI function that uses the '`GET`' method. The data passed back to the server is on the request line. This is used for transferring a small amount of data.

Example 3 shows a CGI function that uses the '`POST`' method. The data is in the body of the HTTP request. This is used for transferring a large amount of data.

Example 4 shows an ISMAP CGI function. Coordinates of an image are passed using the (*`argc`*, *`argv`*) parameters of a CGI function.

Example 5 shows how META commands can retrieve server data. Examples 2 and 3 put values into variables, and Example 5 reads those variables using META commands.

Example 6 shows all the different types of META commands.

Building the Example Web Server for your Target

Edit the **buildpg.cfg** file, found in the **websrc** directory. The following lines might need to be modified to match your target configuration:

```
# Change BindAddress to be the IP address of your target
BindAddress                   206.251.94.188

# Change ServerAdmin to be the email address of someone who
#     administers the target
ServerAdmin                   admin@yourcompany.com

# Change ServerName to the name associated with the IP
#     address of your target
ServerName              Target.yourcompany.com
```

These configuration variables are not used by the Web Server or test programs, but are available for use in your applications.

You may want to familiarize yourself with the other configuration files in the Web Server. More information on these files is given in Chapter 3. New pages are added to the server by specifying the pages in the file **pages.cfg**. If you want to access a variable via a META command, those variables are specified in the file **vartable.cfg**.

Connecting to the Example Web Server

To connect to your Web Server from a browser such as Netscape Navigator or Internet Explorer, enter the following in the open dialog box:

`http://`*`xxx.xxx.xxx.xxx`*

Where *`xxx.xxx.xxx.xxx`* is the IP address (*`BindAddress`* in **buildpg.cfg**) of the target system running the Web Server.

3. Using the Web Server

User Server Functions

These functions are described in this section:

Bwrite()	Performs a buffered write to the network.
GetEntry()	Finds and returns the *ENTRY* structure used to access the web page.
HTTPserver()	Initalizes and runs the Web Server.
HTTPservinit()	Initalizes the Web Server and allocates space for all the structures.
Neof()	Tests for the EOF indicator for the network stream.

Bwrite()

Performs a buffered write to the network.

```
int Bwrite(struct SERV_REC *recp,uchar *buf,ulong len)
```

recp a pointer to the request structure

buf a pointer to the output buffer

len the length of the buffer

Bwrite() writes out the buffer to the network. The output is buffered to minimize network traffic. To flush the buffer, use NULL for *buf*, or *len* of zero.

Return Value

<0 Error

0 or >0 Success

Example

```
Rslt = Bwrite(reqp,buf,len);     /* write buffer */
Rslt = Bwrite(reqp,NULL,0);      /* flush buffer */
```

GetEntry()

Finds and returns the *ENTRY* structure if the web page is found. The *ENTRY* structure is used to access the web page.

```
ENTRY *GetEntry(REQUEST_REQ *reqp,char *file,char *path)
```

reqp a pointer to the request structure

file the name of the file, i.e., **index.html**

path the absolute path after translation

The ***GetEntry()*** function searches the directory specified by *path*for the page *file*.. If the directory or file doesn't exist, a NULL is returned.

This is the ENTRY structure:

```
struct entry {
      char        *name;
      char        *path;
      short       type;
      char        *mime;
      char        *encoding;
      char        *lang;
      void        *offset;
      ulong       clen;
      ulong       ulen;
      ulong       groups;
      ulong       hits;
}
typedef struct entry ENTRY;
```

Return Value

Pointer to ENTRY structure if found

NULL if not found

Example

```
ENTRY *ep = GetEntry(reqp, "index.html",NULL);
```

HTTPserver()

Initalizes and runs the Web Server.

```
int HTTPserver(void)
```

This function initializes the Web Server data structures and executes the main loop that processes incoming requests for the Web Server.

Return Value

<0 Error

Example

```
main()
{
   int  rslt;
   if(Ninit()<0)
            return -1;
   if(Portinit("*")<0)
            return -2;
   rslt = HTTPserver();
   Nterm();
}
```

HTTPservinit()

Initalizes the Web Server and allocates space for all the structures.

```
struct SERV_REC *HTTPservinit(struct SERV_REC *servp)
```

servp the server information for the Web Server

Use the ***HTTPservinit()*** function to initialize server information such as port or IP. The function is called only once per server.

Return Value

`struct SERV_REC` Filled-out server information

Neof()

Tests for the EOF indicator for the network stream.

`int Neof(int stream)`

stream the network file descriptor

Neof() tests the end-of-file indicator for the network stream pointed to by *stream*, returning non-zero if it is set.

Return Value:

0	More data available
!0	End of data

HTTP Server Request Structure

The structure of the HTTP server is very modular, so modules can be added and removed at any time. This allows for additions of new features and control of code size without extensive changes.

The request structure is the heart of the server. The request structure is passed through a sequence of functions which process the request. By having a request filter through different modules, the processing of that request can be tailored to each application. It also allows for user-written processing without affecting other parts of the HTTP server, which reduces debugging.

The processing of the request structure occurs in the ***doreq()*** function. The ***doreq()*** function is called from ***UserLoop()***.

`int` **`doreq`** `(REQ_STRUCT *`*`reqp`*`)`

`reqp` a pointer to the request structure

The pseudocode for ***doreq()*** is:

```
request processing
translate paths
check the URL
check the MIME type
check access
get user ID
authorize the user
handle the request
log the request
```

See also: *Request Structure,* later in this document

The following figure shows the process that each request to the embedded web server goes through.

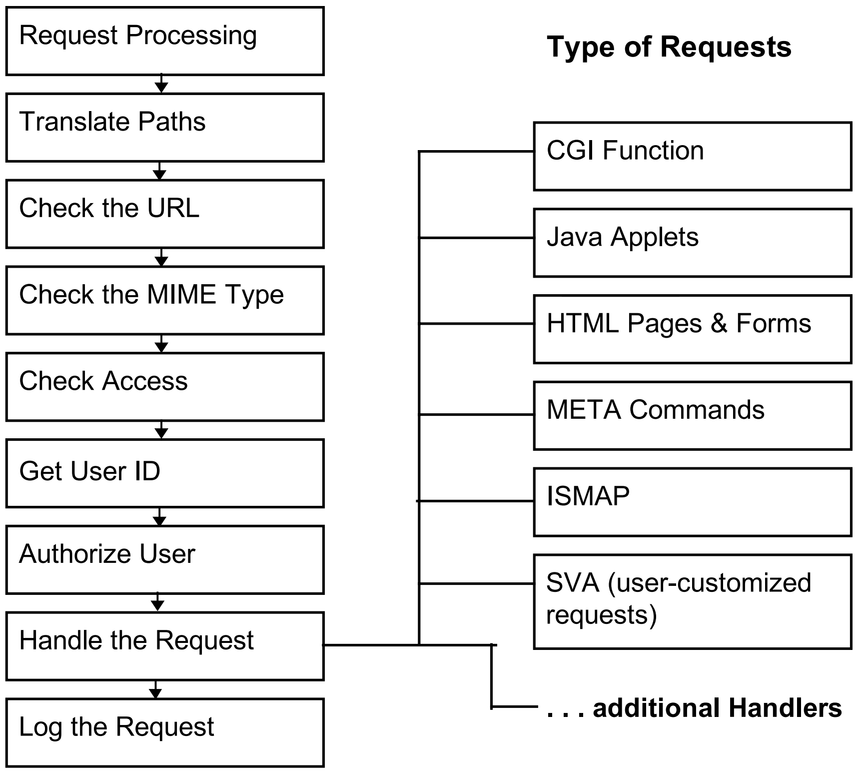

Figure 3-1: Process for Request to the Embedded Web Server

Return Value

<0 Error

Example

See the previous section describing ***UserLoop()***.

Modules and Handlers

The structure of the HTTP server is very modular, so modules can be added and removed at any time. This allows for additions of new features and control of code size without extensive changes. New plug-in modules and increased functionality will be added in the future.

All data is passed through the modules by the request structure. The Web Server provides a framework and default modules for your use, and is designed so the user can customize it. To customize the modules, you must modify or replace the existing modules, using the existing modules as templates.

Each module has a function and modifies only certain parts of the request structure. Only the ***MODtranslate()*** and ***MODchkloc()*** functions are required; all others are optional. The module functions are described in alphabetical order, but are used in this sequence:

MODtranslate()	Parses and translates the URL.
MODchktype()	Determines the type and encoding of the document.
MODchkloc()	Checks for the existence of the file.
MODchkaccess()	Checks access privileges of the document.
MODgetuser()	Performs user authorization.
MODchkauth()	Finds the user in a database or file, and does the final authorization.
MODlog()	Logs errors and access.

Once the request has been processed by all the modules, the final display is the responsibility of the handler function. Each type of page has an associated handler. Each handler processes the page and sends the output to the browser. You can also add your own specialized handlers if needed for customization.

When the default web page type is set to 'text' (in buildpg.cfg), only the text handler is necessary. Additional handlers enhance the Web Server by allowing it to handle different page types.

These typical handlers are included with the USNet Web Server:

`HNDtext`	Handles the standard HTML pages and text.
`HNDcgi`	Sets up the CGI environment and calls the function.
`HNDasis`	Sends the file to the browser without any processing.
`HNDmeta`	Handles server-side HTML parsing.
`HNDussnmp`	Comes with the U S Software SNMP package.

Module Function Descriptions

MODchkaccess()

Gets access privileges of the document.

`int  MODchkaccess(struct request_rec *rec)`

rec pointer to the `request_rec` structure

This optional function checks the group flags or a directory file to determine the access permissions (security) for this page. Access parameters and page permissions are defined in access and page configuration files **access.cfg** and **pages.cfg**. This module sets the access group flag using the information specified in the access configuration file. ***MODchkauth()*** must be written so that the correct username/password returns a flag to match this access group flag.

The default ***MODchkaccess()*** module sets up two types of access checking:

None	No checking done (anyone can access)
Group	Checks a group flag associated with a user

The developer may implement other forms of access checking by modifying or replacing ***MODchkaccess().***

See also: *Request Structure*, in this booklet.
Using Usbldpg and *Page Configuration File*, in the *USNet Web Server User's Guide*.

Return Value

< 0 Error

Otherwise modifies the structure.

Example

This is a pseudocode example for the authentication procecure:

```
MODchkaccess()
/* Checks access restiction of a given web page */
Check request structure for page protection
if (not protected) return 0
if protected
  initialize access information in request structure
  /* specifically, set access group flag */
  return 0
```

MODchkauth()

User-implemented routine to verify user authentication information.

```
int  MODchkauth(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

MODchkauth() is an optional routine that checks the authentication parameters obtained by ***MODgetuser()*** against a user-defined lookup. The default routine supplied with IAP sets the group to match the one specified in the access configuration file, if a preset username and password are entered. This routine must be modified by the developer to implement a site-specific lookup mechanism.

MODchkauth() does two types of access checking:

None	No checking done (anyone can access)
Group	Checks that the individual is within the group

If the developer has set up alternate checking methods in ***MODchkaccess()***, they must be implemented here.

See also: *Request Structure*, in this booklet.
Using Usbldpg and *Page Configuration File*, in the *USNet Web Server User's Guide*.

Return Value

< 0	Error

Otherwise modifies the structure.

Example

MODchkauth()

```
/* Largely user-defined routine to authenticate user info */
if (no access restriction) return 0
match username/password to user-defined lookup
/* Default routine has a hard-coded username and password.
   When these are matched, a hard-coded group flag is
   returned. This group flag matches the one in the access
   configuration file, which was read into the request
   structure in MODchkaccess(). */
if (no match) return 401
if (match) return 0
```

MODchkloc()

Checks for the existence of the file.

```
int  MODchkloc(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

This required module finds the document and sets up a pointer to an embedded structure. If the page is not found, a result of `404` (not found) is returned to the requesting host. The developer may modify this module to find pages in a file system instead of in an embedded structure.

See also:*Request Structure*, in this booklet

Return Value

< 0	Error

Otherwise modifies the structure.

Example

See the file **modchklo.c** in your source code.

MODchktype()

Determines the type and encoding of the document.

```
int  MODchktype(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

This optional function checks the embedded type flags or the extension to determine the correct handler. This routine is appropriate when there is a file system in your embedded target.

See also:*Request Structure*, in this booklet

Return Value

< 0	Error

Otherwise modifies the structure.

Example

See the file **modchkty.c** in your source code.

MODgetuser()

Performs user authorization.

```
int  MODgetuser(struct request_rec *rec)
```

rec pointer to the `request_rec` structure

MODgetuser() is an optional routine that gets authentication information from an end user. The routine extracts the username and password (commonly entered in a pop-up dialog from a browser) from the HTTP headers. This information is stored in the request structure and subsequently processed by ***MODchkauth()***. This routine decodes authentication information using either the basic or digest authentication schemes. Support for any other authentication scheme must be added by the developer.

See also: *Request Structure*, in this booklet
RFC 2069 and chapter 11 of RFC 2068

Return Value

< 0 Error

Otherwise modifies the structure.

Example

MODgetuser()

```
/* Checks user authorization information */
if (no access restriction) return 0
if (no “Authorization” in HTTP header)
  add “WWW-Authenticate” to HTTP header
  return 401 (Unauthorized)
  /* A browser receiving “WWW-Authenticate” will commonly
     pop up a username/password dialog. Entered parameters
     are sent to server as new request with “Authorization”
     in HTTP header. */

if (“Authorization” in header)
  if (not basic or digest authentication) return 401
  decode username and password from HTTP headers
  store username and password in request structure
  return 0
```

MODlog()

Logs errors and requests.

```
int  MODlog(struct request_rec *rec)
```

`rec` pointer to the `request_rec` structure

MODlog() is an optional function that must be impelemented by the developer. This routine could log all requests and errors to a buffer, to a monitor, or to a file if a file system is present.

See also:*Request Structure*, in this booklet

Return Value

< 0 Error

Otherwise modifies the structure.

Example

See the file **httputil.c** in your source code.

MODtranslate()

Parses and translates the URL.

```
int  MODtranslate(struct request_rec *rec)
```

`rec` pointer to the `request_rec` structure

MODtranslate() is a required module that parses the URL and translates its contents to a form usable by the Web Server. The path, file, and query information are parsed from the URL, and stored in the URI structure within the request structure. This information is used in the handler modules to take the appropriate action, such as displaying a page or executing a CGI function. This module supports HTML and CGI translation.

See also:*Request Structure*, in this booklet

Return Value

< 0 Error

Otherwise modifies the structure.

Example

See the file **modtrans.c** in your source code.

Request Structure

The request structure is the heart of the server. As an HTTP request is filtered through the modules, the request structure is filled in.

Since the structure is broken into stages, the user can customize each of the modules with little impact on the rest of the code. This also allows for future enhancements to be added easily.

The request structure is defined in the include file, **httpd.h**. An example of the `request_rec` structure is provided below:

```
struct request_rec {
    short           rslt;       /* result status */
    SERV_REC        *servp;     /* ptr to server rec */
    int             reqfd;      /* req sock descriptor */
    char            *ptr;       /* ptr for strng manip */
    int             blen;       /* buf len left to read*/
    int             slen;       /* sz of sockadd struct*/
    struct sockaddr saddr;      /* sock addr structure */
    short           close;      /* keepalive flag */
    short           protonum;   /* protocol number */
    char            *protover;  /* protocol version */
    short           type;       /* type of HTTP req */
    char            *method;    /* request method */
    short           hostport;   /* listen port */
    char            *reqline;   /* request line */
    char            *status;    /* ptr to status line */
    char            *scheme;    /* GET, POST, (unused) */
    char            *hostname;  /* where from */
    URI             uri;        /* text info */
    short           headcnt;    /* num of HTTP headers*/
    struct headers *headers;    /* HTTP headers */
    short           rplycnt;    /* num to HTTP reply */
    struct headers *rplyheads;/* reply headers */
    unsigned char *body;        /* ptr to body of POST */
    short           bodylen;    /* how big? */
    struct entry   *fileinfo;   /* after page is found,
                                   ptr to the entry */
    char            *mime;      /* mime type */
    char            *encoding;/* the encoding */
    char            *lang;      /* the language */
    char            *accepth;
    char            *connecth;
    char            *from;

    struct cookie   *cookie;    /* cookie info */
    int            (*handler)(struct request_rec *req);
    ACCESS          *access;    /* access structure */
    unsigned long   ldat;       /* undefined data */
    void            *data1;     /* now undefined ptr */
    void            *data2;     /* another undef ptr */
    char                *buff;     /* gen purpose buffer */
};
```

Using Usbldpg

The **usbldpg** utility builds the web pages from your configuration files. To do this, it reads these files in this order:

- The server configuration file, named **buildpg.cfg**
- The MIME types file, named **mime.typ**
- The page configuration file, named **pages.cfg**
- The variable configuration file, named **vartable.cfg**

Usbldpg then takes the pages and turns them into C code, generating:

htpgtbl.c	headers and tables, plus the server configuration and pages in binary format
htpgtbl.dat	an included C file that contains source data for the web pages

These files are then compiled into your application.

Server Configuration File

USWeb server's configuration is similar to the NCSA and Apache* web servers. **Usbldpg** uses the configuration file to build your web pages. There are five different areas of the server configuration, which can be seen in the example file on the next page:

- Other configuration files
- Application system information
- Server information
- Directory and file system information
- MIME information

This is an example of a typical **buildpg.cfg** file:

```
# This configuration file is read by the usbldpg utility #

# other configuration files
BuildDocRoot        .\
PageConfig          pages.cfg
VarConfig           vartable.cfg
TypesConfig         mime.typ

# application system information
Processor           68EN302
HWdate              3 April 1951
HWversion           Release 35.1
HWconfig            WOM (Write Only Memory)
SWdate              11 Aug 1955
SWversion           1309.7.32
SWconfig            swodniW ultra light
TotalMem            32
SysMem              25
FreeMem             7

# server information
BindAddress         206.29.173.23
DefaultType         text/html
Port                80
ServerAdmin         admin@yourserver.company.com
ServerName          yourserver.company.com

# directory and file system information
Alias               /pages/     /
Alias               /other/     /
DirectoryIndex      index.html
Readme              ReadMe

# mime information
AddEncoding         x-zip       zip
AddEncoding         x-gzip      gz
AddType             application/x-us-snmp   smp
AddType             application/x-us-prog   uso
AddType             application/x-us-include      usi
```

Other Configuration Files

These variables provide information on where needed files are located. These files are described in detail later in this chapter.

Table 3-1: Other Configuration Files

Value	Description	Example
`PageConfig`	The name of the page configuration file. See also: *Page Configuration File*, in this chapter.	**pages.cfg**
`VarConfig`	The name of the variable configuration page. Each entry in the file has the format of: *`Searchname,type,size,varname`* An example is: `VAR1,short,sizeof(short),variable1` See also: *Variable Configuration File*, in this chapter.	**vartable.cfg**
`TypesConfig`	The name of the file that contains the file extension to MIME type mapping. See also: *MIME Information*, in this chapter.	**mime.typ**

Application System Information

Application system information contains values that define more about the embedded system. The values are returned to the user when a META command is embedded into the HTML. These values can also be filled in at initialization time by the application. The values must be a string or a number, as specified in the following table, but they are not case-sensitive and can be in any format.

Table 3-2: Application System Information Variables

Value	Description	Example
`BindAddress`	Binds the listen connection to this address (eight 16-bit hex numbers).	0000:0000:0000:0000: 0000:0000:C0A8:0101 (same as 192.168.1.1)
`Port`	The listen port.	80
`ServerAdmin`	The server administrator's e-mail address.	admin@*yourserver.* *company*.com
`ServerName`	The host name of the HTTP server.	*yourserver.* *company*.com
`access_log`	There are two different formats, depending on the logging method: • E-mail address -- the log is stored in RAM until it is mailed to this address. • File name -- the log information is saved to a file.	admin@*yourserver.* *company*.com

Server Information

These variables set the server and network environment.

Table 3-3: Server Information Variables

Value	Description	Example
BuildDocRoot	Defines the path used by **Usbldpg** to preprocess the pages.	**./pages**
DocRoot	On File System this would be the root where the search would start.	**C:/mypages**
DefaultType	If the system does not know what type a file is when handling a message, it will use this type.	**text/html**
Readme	Default name in directory for more information.	**ReadMe**
DirectoryIndex	Default file when no file is specified.	**index.html**
Alias	Changes the URL path, for instance, from **/here/file** to **/there/file** (the physical path would be **C:/mypages/there/file**).	**/here/ /there/**
ScriptAlias	Remaps the URL to a physical directory, and notifies the server that the file being accessed is code.	**/cgi-bin/ /**
ErrorAlias	If an error occurs, the output to the browser is changed from the standard error to this new page.	404 **notfound.html**

Directory and File System Information

These variables provide information on where needed files are located.

Table 3-4: Directory and File System Information Variables

Value	Type	Description	Example
`Processor`	string	Defines the processor type.	68EN302
`HWdate`	string	Defines the hardware build data.	3 April 1951
`HWversion`	string	Defines the hardware version.	Release 35.1
`HWconfig`	string	Contains any special hardware configuration information.	WOM (Write Only Memory)
`SWdate`	string	Defines the software build date.	11 Aug 1955
`SWversion`	string	Defines the software version.	1309.7.32.8
`SWconfig`	string	Contains any special software configuration information.	swodniW ultra light
`TotalMem`	number	The total size of memory, in kilobytes.	32
`SysMem`	number	The amount of memory used by the system. Because the application defines what 'system' is, this could be anything.	25
`FreeMem`	number	The amount of free memory, in kilobytes.	7

MIME Information

MIME file types are defined by suffix (extension), and the MIME type controls how the server or browser will treat the defined files:

- If the file is server-specific, the MIME type tells the server how to handle it.
- If it is a browser file, the server adds the content type(s) to the header information for the browser's use.
- The MIME information also defines how to decode the data, and the **usbldpg** program uses it for the encoding scheme.

There are two ways of defining MIME types for the USNet Web Server: In the **mime.typ** file, or with the ***AddType*** command. The **mime.typ** file included in the USNet Web Server distribution contains most of the standard definitions. The ***AddType*** command adds definitions to the server configuration file, allowing you to keep your **mime.typ** file general.

MIME Types File

This file lists the types of files the server is capable of sending. You can define multiple extensions for one file type.

This is an example portion of a **mime.typ** file:

```
# This is a comment. I love comments.

application/mac-binhex40        hqx
application/msword              doc
application/octet-stream        bin dms lha lzh exe class
application/pdf                 pdf
application/postscript          ai eps ps
application/powerpoint          ppt
application/rtf                 rtf
application/x-compress          Z
```

```
application/x-cpio               cpio
application/x-csh                csh
application/x-director           dcr dir dxr
application/x-gtar               gtar
application/x-gzip               gz
application/x-httpd-cgi          cgi
application/x-tar                tar
application/x-tcl                tcl
application/x-wais-source        src
application/zip                  zip
audio/basic                      au snd
audio/mpeg                       mpga mp2
audio/x-aiff                     aif aiff aifc
audio/x-wav                      wav
image/gif                        gif
image/jpeg                       jpeg jpg jpe
image/tiff                       tiff tif
message/external-body
message/news
multipart/alternative
multipart/appledouble
multipart/digest
multipart/mixed
multipart/parallel
text/html                        html htm
text/plain                       txt
text/x-sgml                      sgml sgm
video/mpeg                       mpeg mpg mpe
video/quicktime                  qt mov
video/x-msvideo                  avi
```

AddType Command

Adds an additional MIME type to the Web Server.

```
AddType  application/type        extension
```

type	the type of file
extension	the extension for the file type

AddType helps define the file type when parsing. The new type goes into the server configuration file (not the **mime.typ** file) and functions like a command. Use ***AddType*** to add specialized MIME types to the Web Server rather than to your **mime.typ** file, thus keeping your **mime.typ** file general.

Example

```
AddType application/x-us-meta    usm
```

Page Configuration File

The page configuration file defines what local pages should be included in embedded web sites. Each page is defined by a line with a format of:

`Buildname,webname,accessname,flags[,maxsize, mime]`

Buildname — the name of the source file on your development system or the name of the CGI routine within the application program.

webname — the URL name.

accessname — a string used to associate authentication parameters with a web page. This variable is used by ***Modchkaccess()***. The authentication parameters associated with *accessname* are specified in **access.cfg**.

flags — define the processing this page needs -- the flags are defined by `0xFFTT`, where `FF` are bit flags and `TT` is a type number.

The flags are defined as:

0x01	RAM/ROM, if set move page to RAM and access it from RAM
0x02	If bit is set, the URL is executable (i.e., CGI function)
0x04	Undefined

The type is:

0,1	TEXT and HTML
2	CGI Function
3	ASIS, just send it out without parsing
4	USMETA, a HTML file with META commands
5	USSNMP, a UUUSMP file with META commands
255	QUIT, exit the server

maxsize — optional numeric variable used to reserve memory (a specified number of bytes) for the web page.

mime — rarely-used optional alpha variable that overrides the MIME definitions from the **mime.typ** file and ***AddType.***

This is a example of a typical **pages.cfg** file:

```
# format is
# build file name or link name
# page name
# accessname: string to define access parameters
# flags bits TYPE 0-7, ROM/RAM = 0x0100, DATA/LINK = 0x0200,
#  0,1 = TEXT
#  2 = CGI
#  3 = ASIS
#  4 = META
#  5 = USSNMP
#  255 = ABORT
# [maxsize] optional (0-9)
# [mime] optional (alpha)

# pages
index.htm,index.html,0,0
linktest.htm,linktest.htm,0,0
imagepag.htm,imagepag.htm,0,0
example3.htm,example3.htm,0,0
example4.htm,example4.htm,0,0
example5.htm,example5.htm,0,0
example6.htm,example6.htm,0,0
mailit.htm,mailit.htm,0,0

#images
example5.gif,example5.gif,0,3
image.jpg,image.jpg,0,3
lava_l.gif,lava_l.gif,0,3

#cgi functions
query_cgi,cgi-bin/query,0,0x0202
post_query_cgi,cgi-bin/post-query,0,0x0202
prntenv_cgi,cgi-bin/prntenv,0,0x0202
mailit_cgi,cgi-bin/mailit,0,0x0202
rainbow.cls,RainbowText.class,0,0x0003
```

Variable Configuration File

The variable configuration file defines the variables in the application that need to be accessed from the web pages. The file translates text strings into variables for access, and creates a table. The web pages can access the variables directly using META commands. You can use this to allow an end-user to access a variable within the application.

The format is:

```
search_name, type, sizeof(type), pointer to it
```

This is an example of a **vartable.cfg** file:

```
NAME,char*,sizeof(name_var),name_var
SEX,char*,sizeof(sex_var),sex_var
AGE,short,sizeof(short),age_var
BROWSER,char*,sizeof(browser_var),browser_var
COLOR,char*,sizeof(color_var),color_var
```

Example

```
<BODY>
The widget count is <!— USMETA VAR=”WIDGETCNT”—><BR>
</BODY>
```

If `WIDGETCNT` is equal to 5, this would print:

```
The widget count is 5
```

CGI Function Programming Interface

The heart of the interactive web is the Common Gateway Interface (CGI). The server needs to display different pages depending on the user's actions. CGI reads parameters from forms on the displayed web page to the server. The data is in the format of:

```
name1=value1, name2=value2
```

The USNet Web Server supplies all needed support routines to manipulate CGI data. The HTTP server uses the standard CGI programming interface, but with a twist. The main difference is that the embedded HTTP server uses subroutines instead of programs.

ISMAP is supported via `argc` and `argv` passed into the CGI function. A mouse click would be passed in as `argv[1]` being x and `argv[2]` being y.

In UNIX the CGI programs are called like:

```
int main(int argc, char *argv[])
```

In the embedded world it would be:

```
int subname(int argc,char *argv[],REQ_STRUCT *reqp)
```

This section includes descriptions of the CGI routines and the CGI system support routines.

System Support Routines

These routines are support routines for the application engineer to use for CGI functions such as exchanging information with the network. They are similar to standard CGI support routines, but tailored to the embedded environment.

These routines are described in this section:

findvar()	Searches the variable structure for a specified string.
getvar()	Searches the request structure for a variable.
Ngetenv()	Searches the environment structure for a specified string.
send_file()	Writes a file to the network.

findvar()

Searches the variable structure for a specified string.

`VARENTRY *findvar(REQSTRUCT *reqp, char *name)`

`reqp` a pointer to the request structure

`name` a pointer to the specified string

The ***findvar()*** function searches the variable structure for a string that matches the string pointed to by *`name`*. It is typically used for changing the variable structure. This allows *`name`* to be reassigned to a different pointer. This routine could be used to write a larger buffer for a pointer associated with the name.

See also:***getvar()***
Request Structure, in the *HTTP Server Request Structure* section.

Return Value

A pointer to the `VARENTRY` structure if found, `NULL` if not found.

Example

```
/* This program demonstrates the GET CGI routines */
/* the HTML is given a filename that is to be sent */

    typedef struct {
        char name[128];
        char val[128];
    } entry;

    static entry entries[10];

    int demo_cgi(int argc,char *argv[],REQUEST_REQ*reqp)

    {
          char *str, fname;
          int *pmaxetn;
          ENTRY *ep;
          VARENTRY *vp;
          str = Ngetenv(reqp,"METHOD");  /* get the
                                                    METHOD=XXXX */
          if(strccmp(str,"GET") != 0) {
                  /* compare str to "GET" case-insensitive */
             str = Ngetenv(reqp,"QUERY_STRING");
             if (str == NULL) {
                PRINTF();
                return 0;
             }
          } else if (strccmp() == 0) {
             char buff[8192];
             (reqp,buff,8192);
             str = buff;
          } else {
             PRINTF(reqp,"BAD METHOD");     /* bad method */
             return 0;
```

```
      }
      pmaxetn = (int*)getvar(reqp,"ENTRYSZ");
                                   /* get a pointer to integer */
      for(x=0;cl[0] != '\0';x++) {
                          /* this section decodes the string
                                   into an array for easy use */
         getword(entries[x].val,cl,'&');
                       /* get the whole "name=value" string */
         plustospace(entries[x].val);
                                      /* change any '+' to ' ' */
         unescape_url(entries[x].val);
                                            /* remove any nasties */
         getword(entries[x].name,entries[x].val,'=');
             /* split the entry into "name" and value" */
         if(x==*pmaxetn)                    /* check if at max */
            break;
      }
   m=x;
   setvar(reqp,"THISENTRY",entries,0);
                       /* save the array to be used later */

                          /* usually the entries are in the
                             same order, but just in case */
   for(x=0;x<m;x++) {                 /* loop through array */
      if(strcmp(entries[x].name,"SENDFILE") == 0){
         fname = entries[x].value;
         break;
      }
   }
   if(x==m) {
      PRINTF(reqp,"not found\n");
      return 0;
   }
   ep = GetEntry(reqp,fname,0);
   send_file(reqp, ep);
   vp = findvar(reqp, "THATVAR");

   if(vp == NULL) {
      PRINTF(reqp,"not found\n");
      return 0;
   }
   PRINTF(reqp,"name >%s, data pointer >%x\n",vp->name,
          vp->data);
   Bwrite(reqp,vp->data,vp->size);
   return 1;
}
```

getvar()

Searches the request structure for a variable.

```
char * getvar(REQSTRUCT *reqp, char *name)
```

`reqp` a pointer to the request structure

`name` a pointer to the specified variable

The ***getvar()*** function searches the request structure for a variable that matches the variable pointed to by `name`. This function is used to access application variables from the CGI routine. The variable accessed is the same as if done from an HTML META command.

See also: ***findvar()***
Request Structure, in the *HTTP Server Request Server* section

Return Value

Returns the pointer needed to access the variable specified by `name`, so the variable's value can be changed

Example

This is included in the example for ***findvar()***.

Ngetenv()

Searches the environment structure for a specified string.

```
char* Ngetenv(REQSTRUCT*reqp, char* str)
```

`reqp` a pointer to the request structure

`str` a pointer to the specified string

The ***Ngetenv()*** function searches the environment structure for a string that matches the string pointed to by *str*.

See also: *Request Structure*, in the *HTTP Server Request Structure* section

Return Value

A pointer to the value in the environment, or `NULL` if there is no match.

Example

This is included in the example for ***findvar()***.

send_file()

Writes a file to the network.

```
int send_file(REQSTRUCT *reqp, ENTRY *ep)
```

reqp a pointer to the request structure

ep pointer to the `ENTRY` structure, where `ENTRY` is a structure that contains a file or page description

The ***send_file()*** function writes the file in the `ENTRY *ep` to the network. This is a way to send out a file without processing it.

See also: ***GetEntry()*** description, in this chapter, for a definition of the `ENTRY` structure
Request Structure, in the *HTTP Server Request Structure* section

Return Value

< 0	Error
0 or > 0	Success

Example

This is included in the example for ***findvar()***.

CGI Routines

These routines are described in this section:

escape_shell_cmd()	Converts all 'nasty control characters' to '\x'.
getword()	Parses a string.
Nmakeword()	Parses a string and returns a pointer to the word that was matched.
plustospace()	Converts all '+' to spaces.
unescape_url()	Searches for %xx and terminates the string.
x2c()	Converts two hex values into an unsigned 8-bit value.

escape_shell_cmd()

Converts all ‘nasty control characters’ to ‘\x’.

```
void escape_shell_cmd(char *cmd)
```

cmd the string to convert

The ***escape_shell_cmd()*** routine converts unwanted characters (which might blow up shells, be security holes, etc.) in the specified string to ‘safe’ characters. The 'nasty control characters' which are processed are:

```
& ; ` ' " | * ? ~~ < > ^ ( ) [ ] { } $ \ 0x0A
```

Return Value

None

Example

```
char *buf = "grep foo > x";
escape_shell_cmd (buf);
  /* After execution of escape_shell_cmd(),
     buf is "grep foo \> x". */
```

getword()

Parses a string.

```
void getword(char *word, char *line, char stop)
```

word	a pointer to buffer space
line	the beginning of the string
stop	the ending character

Getword() parses the string pointed to by *line* until the *stop* char is matched or there is an end-of-string or end-of-line. ***Getword()*** returns the contents of the buffer pointed to by *word*, and adjusts *line* to point to the next character after the *stop* character.

See also:***Nmakeword()***

Return Value

The contents of the buffer (the line up to the stop character) pointed to by *word*.

Example

This example determines whether to do GET or POST, and shows a GET routine and a POST routine. It includes ***getword()***, ***plustospace()***, and ***unescape_url()***.

```
#include httpd.h

extern int getcgi(int,char**,REQ_STRUCT*);
extern int postcgi(int,char**,REQ_STRUCT*);
#ifdef UNIX
int main(int argc,char *argv[])
#else
int cgiroutine(int argc,char *argv[], REQ_STRUCT *reqp)
#endif
{
   char *method = GETENV("REQUEST_METHOD");
   if(strcmp(method,"GET") == 0){
      return getcgi(argc,argv,reqp);
   }
   if(strcmp(method,"POST") == 0) {
      return postcgi(argc,argv,reqp)
   }
   return -1;                          /* bad request */
}

int getcgi(int argc,char* argv[],REQ_STRUCT *reqp);
{
   char *query;
   int m,x;
   query = GETENV("QUERY_STRING");
   if(query == NULL) {
      PRINTF(reqp,"No query information to decode.\n");
      EXIT(1);
   }
```

```
    for(x=0;query[0] != '\0';x++) {
        getword(entries[x].val,query,'&');
            /* get the whole name=value string */
        plustospace(entries[x].val);      /* convert '+' to ' ' */
        unescape_url(entries[x].val);
                            /* remove any nasty chars that might
                               blow up the system */
        getword(entries[x].name,entries[x].val,'=');
                                    /* separate name from value */
    }

    m=x;
    PRINTF(reqp,"<H1>Query Results</H1>");
    PRINTF(reqp,"You submitted the following name/value
             pairs:<p>%c",10);
    PRINTF(reqp,"<ul>%c",10);
    for(x=0; x < m; x++)
        PRINTF(reqp,"<li> <code>%s = %s</code>%c",
            entries[x].name, entries[x].val,10);
    PRINTF(reqp,"</ul>%c",10);
    return 0;
}

int postcgi(int argc,char* argv[],REQ_STRUCT *reqp);
{
    char *body;
    int m,x,qlen;
    qlen = atoi(GETENV("CONTENT_LENGTH"));
                                    /* needed to buffer the input */
    body = getbody(reqp);
    for(x=0;!Neof(reqp);x++) {            /* read until no more */
        entries[x].val = Nmakeword(reqp,'&',&cl);
                        /* read input stream for full name=value */
        plustospace(entries[x].val);      /* convert '+' to ' ' */
        unescape_url(entries[x].val);
        entries[x].name = getword(entries[x].val,'=');
    }

    m=x;
    PRINTF(reqp,"<H1>Query Results</H1>");
    PRINTF(reqp,"You submitted the following name/value pairs:
        <p>%c",10);
    PRINTF(reqp,"<ul>%c",10);
    for(x=0; x <= m; x++)
        PRINTF(reqp,"<li> <code>%s = %s</code>%c",
            entries[x].name,entries[x].val,10);
    PRINTF(reqp,"</ul>%c",10);
    query = GETENV("QUERY_STRING");
    if(query == NULL) {
        PRINTF(reqp,"No query information to decode.\n");
        EXIT(1);
    }
```

```
    for(x=0;query[0] != '\0';x++) {
        getword(entries[x].val,query,'&');   /* get the whole
                                                 name=value string */
        plustospace(entries[x].val);      /* convert '+' to ' ' */
        unescape_url(entries[x].val);     /* remove any nasty chars
                                    that might blow up the system */
        getword(entries[x].name,entries[x].val,'=');/* separate
                                          name from value */
    }

    m=x;
    PRINTF(reqp,"<H1>Query Results</H1>");
    PRINTF(reqp,"You submitted the following name/value
        pairs:<p>%c",10);
    PRINTF(reqp,"<ul>%c",10);
    for(x=0; x < m; x++)
        PRINTF(reqp,"<li> <code>%s = %s</code>%c",
            entries[x].name,entries[x].val,10);
    PRINTF(reqp,"</ul>%c",10);
    return 0;
}
```

Nmakeword()

Parses a string.

`char * Nmakeword(char *line, char *stop)`

line the beginning of the string

stop the ending character

Nmakeword() is like ***getword()*** but it returns a pointer to the word that was matched.

It parses the string pointed to by `line` until the `stop` char is matched or there is an end-of-string or end-of-line. ***Nmakeword()*** returns a pointer to the `word`, and `line` is adjusted to point to the next character after the `stop` character.

See also:***getword()***

Return Value

A pointer to the word that was matched.

Example

See the file **cgiutil.c** in your source code for an example.

plustospace()

Converts all '+' to spaces

`void plustospace(char *str)`

str the string to convert

Return Value

None

Example

This is included in the examples for ***getword()***.

unescape_url()

Searches for `%xx` and terminates the string.

```
void unescape_url(char *url)
```

url the URL to convert

The ***unescape_url()*** routine converts hex numbers to characters.

Return Value

None

Example

This is included in the example for ***getword()***.

x2c()

Converts two hex values into an unsigned 8-bit value.

```
char x2c(char *what)
```

what the hexadecimal value to convert

The conversion is to characters or integers, depending on the hexadecimal number specified.

Return Value

The converted value.

Example

```
char *str="AB";
char num;
num = x2c(str);      /* num = 0xab */
```

See the file **cgiutil.c** in your source code for another example.

CGI Environment Variables

When programming CGI, all the data about the world around you is passed by environment variables. Each environment variable has a different meaning.

Table 3-5: CGI Environment Variables

Variable	Description
SERVER_SOFTWARE	The name and version of the information server software answering the request (and running the gateway). Format: name/version
SERVER_NAME	The server's hostname, DNS alias, or IP address as it would appear in self-referencing URLs.
GATEWAY_INTERFACE	The revision of the CGI specification to which this server complies. Format: CGI/revision
SERVER_PROTOCOL	The name and revision of the information protocol this request came in with. Format: protocol/revision
SERVER_PORT	The port number to which the request was sent.
REQUEST_METHOD	The method with which the request was made. For HTTP, this is "GET", "HEAD", "POST", etc.

Table continued on next page.

Table 3-5: CGI Environment Variables (section 2 of 3)

PATH_INFO	The extra path information, as given by the client. In other words, scripts can be accessed by their virtual pathname, followed by extra information at the end of this path. The extra information is sent as PATH_INFO. If this information comes from a URL, the server should decode the information before it is passed to the CGI script.
PATH_TRANSLATED	The server provides a translated version of PATH_INFO, which takes the path and does any virtual-to-physical mapping to it.
SCRIPT_NAME	A virtual path to the script being executed, used for self-referencing URLs.
QUERY_STRING	The information which follows the ? in the URL which referenced this script. This is the query information, and it should not be decoded in any way. This variable should always be set when there is query information, regardless of command line decoding.
REMOTE_ADDR	The IP address of the remote host making the request.
AUTH_TYPE	If the server supports user authentication, and the script is protected, this is the protocol-specific authentication method used to validate the user.
REMOTE_USER	If the server supports user authentication, and the script is protected, this is the username they have authenticated as.

Table continued on next page.

Table 3-5: CGI Environment Variables (section 3 of 3)

REMOTE_IDENT	If the HTTP server supports RFC 931 identification, then this variable will be set to the remote user name retrieved from the server. Usage of this variable should be limited to logging only.
CONTENT_TYPE	For queries that have attached information, such as HTTP POST and PUT, this is the content type of the data.
CONTENT_LENGTH	The length of the content as given by the client.
HTTP_ACCEPT	The MIME types which the client will accept, as given by HTTP headers. Other protocols may need to get this information elsewhere. Commas as per the HTTP spec should separate each item in this list. Format: type/subtype, type/subtype
HTTP_USER_AGENT	The browser the client is using to send the request. General format: software/version library/version
DATE_GMT	The current date and time in Greenwich mean time.
DATE_LOCAL	The current date and time in the local time zone for the server.
DOCUMENT_NAME	The name of the file using this variable. Contains only the file name, not the location.
DOCUMENT_URI	The path to the file using this variable relative to the page root directory. Contains the directory location and the file name. For example: /parsed_docs/myfile.shtml
LAST_MODIFIED	The last modification date of the file using this variable.

USMETA Programming Interface

META commands are used to access predefined application system variables in the **vartable.cfg** file. They allow HTML access of the variables, which can be viewed while the application is running. You must define these variables and update them when necessary.

See also: *Variable Configuration File*, in this chapter

META commands are parsed by the server, and are stored as comments in the body of the HTML page. The commands have this format:

```
<!—#command arg=”value”—>
```

Each command accepts different arguments. For example, this command includes a separate file within the page:

```
<!—#include virtual+”../includes/header.txt”—>
```

If the server cannot parse the command in the comment because of an error, it returns the unparsed comment to the browser.

The power of META commands is the ability to not only have access to the variable, but to format the variable.

For example, if you wanted to access an IP address, you can have it printed out in ether hex or decimal:

```
hex: <!—#ECHO FORMAT=”%x” VAR=”ipaddress”—>
dec:  <!—#ECHO FORMAT=”%d” VAR=”ipaddress”—>
```

This is a command to print a string:

```
<!—ECHO FORMAT=”this is it %s” VAR=”astring”—>
```

It would print out “this is a web page” if `astring` contains “`web page`”.

These HTML META tags are described in this section:

#echo Prints a statement to the browser screen.

#exec Runs a CGI function.

#include Inserts the contents of a file.

#memory Prints the memory size, in kilobytes.

#system Prints information about the system.

#echo

Prints a statement to the browser screen.

The ***#echo*** command includes the value of one of the environment variables defined for CGI programs (see *CGI Environment Variables*) or uses SVA (Server Variable Access) to include one of the variables defined in the **vartable.cfg** file (see *Variable Configuration File*). By echoing a variable to the browser, the web page can dynamically update the page.

The only argument is `var`, whose value is the name of the variable you want to output.

Example 1

```
<!—#echo var==”HTTP_USER_AGENT”—>
```

Example 2

```
<HTML>

<HEAD>

<TITLE> Meta Commands Examples </TITLE>

</HEAD><BODY>

This is an example of meta commands.

<!--#include file=”header.txt”-->  <!-- this would read the file
‘header.txt’ and send it out, then continue sending out this file-->

The number of widgets is <!-- #echo var=”WIDGCNT”-->
  <!-- would look like The number of widgets is 5 -->

Total Memory is <!-- #memory total-->
  <!-- Total Memory is 512K -->

</BODY>

</HTML>
```

#exec

Runs a CGI function.

The valid arguments are:

cgi	runs the CGI function you specify and includes its output in the page.

The #exec META tag is useful when a web page should contain dynamically generated information that is best localized in a CGI function. For simple text insertions, #echo combined with server variables should be less complicated to implement.

The CGI function is implemented as discussed in the “CGI Function Programming Interface” section earlier in this document, and can process arguments. The server does not check to make sure your CGI program produces an output.

Example

```
<!—#exec cgi=”cgi-bin/fill_in”—>
```

#include

Inserts the contents of a file.

```
<!—#include file=”filename”—>
<!--#include virtual=”path”—>
```

The ***#include*** command accepts either of the following arguments:

`file` gives a relative reference to the file you want to include. The path is relative to the directory containing the file that uses the ***#include*** command. You cannot use absolute paths with this argument. To keep your non-public directories secure, a page cannot use relative paths that traverse upward through the directory structure (that is, it cannot use paths that contain ../).

> *`filename`* the filename in the physical file structure on the server machine

`virtual` gives the path to a file relative to the page root directory for the server. The double dash after the “!” is necessary.

> *`path`* the file path as seen from the outside by those accessing the server

The ***#include*** command inserts the contents of the file you specify at the location of the ***#include*** command. The user must have read access to the file that gets included. If the file that is included has a file extension or location that causes it to be parsed by the server, that file can in turn include other files.

Make sure files you include contain only tags that are appropriate in the context of the files that include them. For example, don’t use the <HTML>, <HEAD>, or <BODY> tags (or their end tags) in a file that will be included in another file that already contains these tags.

Examples

```
<!—#include file=”include.txt”—>
<!—-#include virtual=”/doc/cust/include.txt”—>
```

See also:The second example for the ***#echo*** command.

#memory

Prints the memory size, in kilobytes.

The ***#memory*** command accepts these variables:

`total` returns the total amount of memory in the system.

`system` returns the amount of memory used by 'the system'.
Because this is defined by the application, 'the system' is user-defined.

`free` returns the amount of free memory.

This command returns information from the server configuration file's `TotalMem`, `SysMem`, or `FreeMem` field, where the application has earlier set these global variables.

See also: *Server Configuration File*, in this chapter

Examples

```
<!—#memory total—>

<!—#memory system—>

<!—#memory free—>
```

See also: The second example for the ***#echo*** command.

#system

Prints information about the system.

The variables are stored in:

`processor` returns the system processor string:

HWdate	hardware date
HWversion	hardware version
HWconfig	hardware configuration
SWdate	software date
SWversion	software version
SWconfig	software configuration

Example

```
<!-- #system HWdate=>
```

Index

#

#echo META command
description 56
example 56
#exec META command
description 57
example 57
#include META command
description 58
examples 58
#memory META command
description 59
examples 59
#system META command
description 59

A

access, restricting 21
AddType command
description 35
example 35
authentication of user 22, 24

B

browser
printing to 56
building
example Web Server for target 11
for UNIX 7
with another TCP/IP stack 8
buildpg.cfg file
editing 11
example 28
Bwrite() user server function 14

C

CGI
definition 2, 39
CGI environment variables 52
CGI function programming interface 39
CGI programs
running 57
CGI routines 45
escape_shell_cmd() 46
getword() 47
Nmakeword() 50
plustospace() 50
summary list 45
unescape_url() 51
x2c() 51
CGI support routines
calling 39
findvar() 41
general description 39
getvar() 43
Ngetenv() 43
send_file() 44
summary list 40
configuring 27
connecting from browser 11

D

data structures
request_rec 26
DNS
definition 2
documents
determining encoding of 23
determining type 23
finding 23

E

embedded web server
including pages 36
request process 19
encoding, determining type 23
ENTRY structure
definition 15
finding and returning 15
EOF for network stream 17
errors, logging 25
escape_shell_cmd() CGI routine
description 46
example Web Servers 10

Index

F

files
  writing to network 44
files, checking for 23
findvar() CGI support routine
  description 41
  example 42

G

GetEntry() user server function 15
getvar() CGI support routine
  description 43
getword() CGI routine
  description 47
  example 47

H

handlers
  included, summary list 20
HTML META commands
  definition 2
HTTP server
  modules 20
  structure 18
HTTP, definition 2
HTTPserver() user server function 16

I

initializing 16
ISMAP 39
  definition 2

L

logging errors and access 25

M

memory
  printing size of 59
META commands
  #echo 56
  #exec 57
  #include 58
  #memory 59
  #system 59
  arguments accepted 55
  format of 55
  general description 55
  summary list 55
MIME types
  adding to server 35
MIME types file 33
  example 33
MIME, definition 2
MODchkaccess() module/function 21
mouse click 39

N

Neof() user server function 17
network
  buffered write to 14
  writing files to 44
network stream
  finding EOF 17
Ngetenv() CGI support routine
  description 43
  example 41
Nmakeword() CGI routine
  description 50

P

page configuration file
  description 36
  example 37
pages.cfg file
  example 37
parsing URLs 25
plustospace() CGI routine
  description 50
  example 47
POP, definition 2

R

request structure
  description 18
  example 26
  loop 18
  searching for variables 43
request to web server
  process 19
request_rec structure 26
requirements 6

running 16

S

send_file() CGI support routine
description 44
example 42
server configuration
application system information 30
directory and file system variables 32
MIME types 33
other files 29
page configuration file 36
server information variables 31
server configuration file
contents 27
example 28
SMTP, definition 2
strings
parsing 47, 50
searching for 43
terminating 51
structures
allocating space for 16
ENTRY 15, 44
request 18
SVA
and #echo command 56
definition 2
system information
printing 59

T

TCP/IP, definition 2
terminology 2
translating URLs 25

U

unescape_url() CGI routine
description 51
example 47
UNIX
building web server for 7
URL
translating and parsing 25
usbldpg utility
files generated 27
files read 27
using 5
user authentication 22, 24
user server functions
Bwrite() 14
GetEntry() 15
HTTPserver() 16
Neof() 17
summary list 13
USMETA programming interface 55

V

variable configuration file
description 38
example 38
variable structure
searching for string 41
vartable.cfg file
example 38

W

web pages
inserting into web server 5
steps for creating 5
web server modules
description 20
web server modules/functions
MODchkaccess() 21
sequence of use 20
summary list 20
write, buffered 14

X

x2c() CGI routine
description 51