

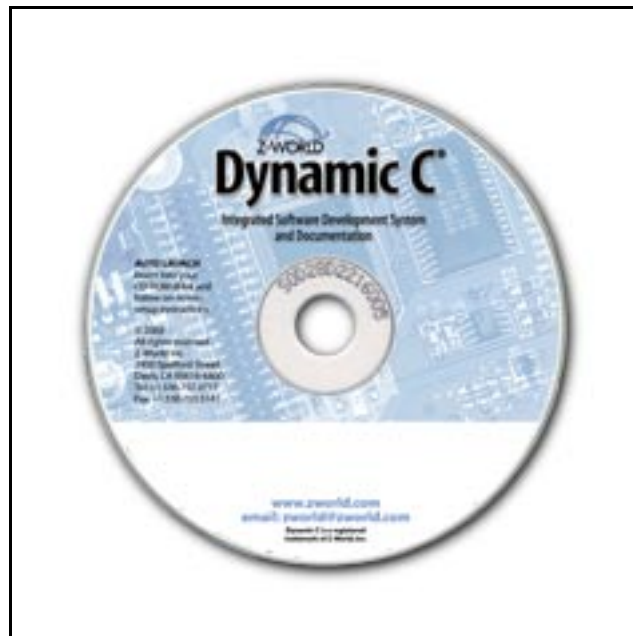


Dynamic C TCP/IP

User's Manual

Volume 2

050223 • 019-0144-B



This manual (or an even more up-to-date revision) is available for free download at the Z-World website: www.zworld.com.

Dynamic C TCP/IP User's Manual

Volume 2

Part Number 019-0144-B • Printed in U.S.A.

©2005 Z-World Inc. • All rights reserved.

Z-World reserves the right to make changes and improvements to its products without providing notice.

Trademarks

Dynamic C is a registered trademark of Z-World Inc.

Windows® is a registered trademark of Microsoft Corporation

Z-World, Inc.

2900 Spafford Street
Davis, California 95616-6800
USA

Telephone: 530.757.3737
Fax: 530.757.3792 or 530.753.5141
www.zworld.com

Table of Contents

1	Introduction.....	1	sspec_addform	60
2	Web-Enabling Your Application.....	3	sspec_adddfsfile.....	61
	2.1 Designing Your Application.....	3	sspec_addfunction	62
	2.2 The Smallest Web Server in the WWW ...	4	sspec_addfv	63
	2.3 Web Server Architecture	6	sspec_addrrootfile	64
	Application Block	7	sspec_addrule.....	65
	HTTP Block	8	sspec_adduser	67
	HTTP Block Subcomponents	9	sspec_addvariable.....	68
	Zserver Block	10	sspec_addxmemfile	69
	2.4 Architecture of a Toy Application.....	11	sspec_addxmemvar.....	70
	2.5 A Simple but Realistic Application.....	13	sspec_aliasspec	71
	2.6 Adding Access Controls.....	16	sspec_automount	72
	2.7 A Full-Featured Application.....	22	sspec_cd.....	74
	2.8 Living Without RabbitWeb and FAT	26	sspec_checkaccess	75
3	Server Utility Library.....	31	sspec_checkpermissions	76
	3.1 Data Structures for Zserver.lib	31	sspec_close	77
	MIMETypeMap Structure	32	sspec_delete	78
	ServerSpec Structure	33	sspec_dirlist	79
	ServerAuth Structure	35	sspec_fatregister	82
	ServerPermissions Structure	35	sspec_fatregistered.....	83
	RuleEntry Structure	36	sspec_findfv	84
	ServerContext Structure	36	sspec_findname	85
	SSpecStat Structure	37	sspec_findfsname.....	86
	sspec_fatinfo Structure	37	sspec_findnextfile.....	87
	FormVar Structure	38	sspec_getfileloc	88
	SSpecFileHandle Structure	38	sspec_getfiletype	89
	3.2 Constants Used in Zserver.lib.....	38	sspec_getformtitle.....	90
	ServerSpec Type Field	38	sspec_getfunction	91
	ServerSpec Vartype Field	39	sspec_getfvdesc	92
	ServerPermissions Servermask Field	39	sspec_getfventrytype	93
	Configuration Macros	39	sspec_getfvlen	94
	Macros for Control Data Initialization	41	sspec_getfvname.....	95
	3.3 File Compression Support	44	sspec_getfvnum	96
	3.4 HTML Forms.....	45	sspec_getfvopt	97
	3.5 API Functions.....	45	sspec_getfvoptlistlen	98
	sauth_adduser	46	sspec_getfvreadonly	99
	sauth_authenticate	47	sspec_getfvspec	100
	sauth_getpassword.....	48	sspec_getlength.....	101
	sauth_getserver	48	sspec_getMIMEtype	102
	sauth_getuserid	49	sspec_getname	103
	sauth_getusermask.....	50	sspec_getpermissions.....	104
	sauth_getusername.....	51	sspec_getpreformfunction	105
	sauth_getwriteaccess	52	sspec_getrealm.....	106
	sauth_removeuser	53	sspec_getservermask	107
	sauth_setpassword	54	sspec_gettype.....	108
	sauth_setserver.....	55	sspec_getuserid	109
	sauth_setusermask	56	sspec_getusername	110
	sauth_setwriteaccess.....	57	sspec_getvaraddr	111
	sspec_access	58	sspec_getvarkind	112
	sspec_addCGI.....	59	sspec_getvartype.....	113
			sspec_getxvaraddr	113
			sspec_mkdir	114
			sspec_needsauthentication	115
			sspec_open	116
			sspec_pwd.....	119
			sspec_read.....	120
			sspec_readfile	121
			sspec_readvariable	122
			sspec_remove.....	123
			sspec_removeuser	124
			sspec_removeuser	125
			sspec_resizerootfile	126
			sspec_restore.....	127

sspec_rmdir	128	http_getCond	217
sspec_save	129	http_getContentDisposition	218
sspec_seek	130	http_getContentLength	219
sspec_setformepilog	131	http_getContentType	220
sspec_setformfunction	132	http_getcontext	221
sspec_setformprolog	133	http_getContext	222
sspec_setformtitle	134	http_getData	223
sspec_setfvcheck	135	http_getDataLength	224
sspec_setfvdesc	136	http_getField	225
sspec_setfventrytype	137	http_getHTTPMethod	226
sspec_setfvfloatrange	138	http_getHTTPVersion	227
sspec_setfvlen	139	http_getRemainingLength	228
sspec_setfvname	140	http_getSocket	229
sspec_setfvoptlist	141	http_getState	230
sspec_setfvrange	142	http_getTransferEncoding	231
sspec_setfvreadonly	143	http_getURL	232
sspec_setpermissions	144	http_getUserState	233
sspec_setpreformfunction	145	http_handler	234
sspec_setrealm	146	http_idle	235
sspec_setsavedata	147	http_init	236
sspec_setuser	148	http_nextfverr	237
sspec_stat	149	http_parseform	238
sspec_tell	151	http_safe	239
sspec_write	152	http_scanpost	240
4 HTTP Server	153	http_set_anonymous	241
4.1 HTTP Server Data Structures	154	http_setauthentication	242
HttpState	154	http_setCond	243
4.2 Configuration Macros	157	http_setcookie	244
Sending Customized HTTP Headers to		http_set_path	245
the Client	159	http_setState	246
Saving Custom Headers from the Client		http_shutdown	247
160	160	http_skipCGI	248
4.3 Authentication Methods	161	http_sock_bytesready	249
4.4 Setting the Time Zone	162	http_sock_fastread	250
4.5 Sample Programs	163	http_sock_fastwrite	251
Serving Static Web Pages	163	http_sock_gets	252
Dynamic Web Pages Without HTML		http_sock_mode	253
Forms	165	http_sock_tbleft	254
Web Pages With HTML Forms	170	http_sock_write	255
HTML Forms Using Zserver.lib	176	http_sock_xfastread	256
4.6 HTTP File Upload	182	http_sock_xfastwrite	257
What is a CGI Function and Why is It		http_status	258
Useful?	182	http_switchCGI	259
How Do I Use the New CGI Facility?		http_urldecode	260
183	183	http_write	261
4.7 API Functions for HTTP Servers	202	shtml_addfunction	263
cgi_continue	202	shtml_addvariable	264
cgi_redirectto	203	shtml_delfunction	265
cgi_sendstring	204	shtml_delvariable	266
http_abortCGI	205	5 FTP Client	267
http_addfile	206	5.1 Configuration Macros	267
http_contentencode	207	5.2 API Functions	268
http_date_str	208	ftp_client_setup	268
http_defaultCGI	209	ftp_client_tick	269
http_delfile	211	ftp_client_filesize	270
http_finderrbuf	212	ftp_client_xfer	271
http_findname	213	ftp_data_handler	272
http_finishCGI	214	ftp_last_code	274
http_getAction	215	5.3 Sample FTP Transfer	275
		6 FTP Server	277
		6.1 Configuration Macros	278
		6.2 File Handlers	280

Replacing the Default Handlers	280	pop3_getmail	328
File Handlers Specification	280	pop3_tick	329
ftp_dflt_open	281	9.5 Sample Receiving of E-mail.....	330
ftp_dflt_getfilesize	282	Sample Conversation	331
ftp_dflt_read	283	10 Telnet	333
ftp_dflt_write	284	10.1 Telnet (Dynamic C 7.05 and later)	333
ftp_dflt_close	285	Setup	333
ftp_dflt_list	286	API Functions (Dynamic C 7.05 and	
ftp_dflt_cd	287	later)	334
ftp_dflt_pwd	288	vserial_close	334
ftp_dflt_mdtm	289	vserial_init	334
ftp_dflt_delete.....	290	vserial_keepalive	335
6.3 API Functions	291	vserial_listen	336
ftp_dflt_is_auth.....	291	vserial_open.....	337
ftp_init	292	vserial_tick.....	338
ftp_load_filenames	293	10.2 Telnet (pre-Dynamic C 7.05).....	339
ftp_save_filenames	294	Configuration Macros	339
ftp_set_anonymous	295	API Functions	339
ftp_shutdown	296	telnet_init	339
ftp_tick	297	telnet_tick	340
6.4 Sample FTP Server.....	298	telnet_close	340
6.5 Getting Through a Firewall	299	An Example Telnet Server	341
6.6 FTP Server Commands.....	299	An Example Telnet Client	342
6.7 Reply Codes to FTP Commands	301	11 General Purpose Console.....	343
7 TFTP Client	303	11.1 Zconsole Features	343
7.1 BOOTP/DHCP	303	File System Requirement	343
7.2 Data Structure for TFTP.....	304	TCP/IP and Zconsole	343
7.3 API Functions	304	11.2 Login Name and Password.....	344
tftp_init	305	11.3 Zconsole Commands and Messages ...	344
tftp_initx	306	Zconsole Command Data Structure	344
tftp_tick	307	11.4 Zconsole Command Array.....	346
tftp_tickx.....	308	Zconsole Commands	346
tftp_exec	309	Zconsole Error Messages	353
8 SMTP Mail Client.....	311	11.5 Zconsole I/O Interface	356
8.1 Sample Conversation.....	311	How to Include an I/O Method	356
8.2 SMTP Authentication.....	312	Predefined I/O Methods	356
8.3 Sample Sending of an E-mail.....	313	Multiple I/O Streams	357
8.4 Configuration Macros.....	314	11.6 Zconsole Execution	358
8.5 API Functions	316	File System Initialization	358
smtp_data_handler.....	316	Serial Buffers	358
smtp_mailtick	318	Using TCP/IP	358
smtp_sendmail	319	Required Zconsole Functions	359
smtp_sendmailxmem	320	console_init.....	359
smtp_setauth	321	console_tick.....	359
smtp_setserver	322	Useful Zconsole Function	360
smtp_setserver_ip	322	con_backup	360
smtp_status	323	con_backup_bytes	360
9 POP3 Client	325	con_backup_reserve	361
9.1 Configuration.....	325	con_chk_timeout	361
9.2 Steps to Receive E-mail.....	326	con_load_backup	362
9.3 Call-Back Function.....	326	con_reset_io.....	362
Normal call-back	326	con_set_backup_lx	363
POP_PARSE_EXTRA call-back	326	con_set_files_lx.....	363
9.4 API Functions	327	con_set_user_idle	364
pop3_init.....	327	con_set_timeout.....	364
		con_set_user_timeout	365

console_disable	365
console_enable	366
Zconsole Execution Choices	367
11.7 Backup System.....	368
Data Structure for Backup System	368
Array Definition for Backup System	369
11.8 Zconsole Macros	370
11.9 Sample Program.....	373
Notice to Users	377
Index	379

1. Introduction

The TCP/IP User's Manual is intended for embedded system designers and support professionals who are using a Rabbit-based controller board. Most of the information contained here is meant for use with Ethernet-enabled boards, but using only serial communication is also an option. Knowledge of networks and TCP/IP (Transmission Control Protocol/Internet Protocol) is assumed. For an overview of these two topics a separate manual is provided, *An Introduction to TCP/IP*. A basic understanding of HTML (HyperText Markup Language) is also assumed. For information on this subject, there are numerous sources on the Web and in any major book store.

The Dynamic C implementation of TCP/IP comprises several libraries. The main library is `DCRTCP.LIB`. As of Dynamic C 7.05, this library is a light wrapper around `DNS.LIB`, `IP.LIB`, `NET.LIB`, `TCP.LIB` and `UDP.LIB`. These libraries implement DNS (Domain Name Server), IP, TCP, and UDP (User Datagram Protocol). This, along with the libraries `ARP.LIB`, `ICMP.LIB`, `IGMP.LIB` and `PPP.LIB` are the transport and network layers of the TCP/IP protocol stack.

The Dynamic C libraries:

- `BOOTP.LIB`
- `FTP_SERVER.LIB`
- `FTP_CLIENT.LIB`
- `HTTP.LIB`
- `POP3.LIB`
- `SMNP.LIB`
- `SMTP.LIB`
- `TFTP.LIB`
- `VSERIAL.LIB`

implement application-layer protocols. Except for `BOOTP`, which is described in volume 1 of the manual, these protocols are described in volume 2.

All user-callable functions are listed and described in their appropriate chapter. Example programs throughout both volumes of the manual illustrate the use of all the different protocols. The sample code also provides templates for creating servers and clients of various types.

To address embedded system design needs, additional functionality has been included in Dynamic C's implementation of TCP/IP. There are step-by-step instructions on how to create HTML forms, allowing remote access and manipulation of information. There is also a serial-based console that can be used with TCP/IP to open up legacy systems for additional control and monitoring. The console may also be used for configuration when a serial port is available. The console and HTML forms are discussed in volume 2.

Multiple interfaces are supported starting with Dynamic C version 7.30.

2. Web-Enabling Your Application

This chapter, and the next two, describe how to add web browser control to your application. Web-enabling is a logical and appealing choice for adding a user interface to your application, since the necessary hardware (an Ethernet or serial port) is available on all Z-World core modules and SBCs. Most users of your application will be familiar with at least one web browser (Netscape, Mozilla, Internet Explorer, Opera), with its graphical user interface, so they will be ready to start controlling your application with minimal training.

This chapter provides an overview of the steps you will need to take to web-enable an application. Knowledge of browsers, and something of their capability, is assumed. With this knowledge, you can understand the concepts described in this chapter. The following chapters go into more detail about the specific libraries; but for simple programs, you may be able to use just the information in this chapter along with the sample code to write a working application.

Dynamic C provides libraries that implement most of the functions required to implement a web server, more formally known as an HTTP (HyperText Transfer Protocol) server. (The browser is formally called an HTTP client). You only need to write code specific to your application, such as dealing with I/Os and the Rabbit peripheral devices, and possibly some code to help the HTTP server generate the appropriate responses back to the user's web browser. In addition, there is a small amount of "boilerplate" that needs to be written to include and configure the HTTP server and any ancillary libraries such as the TCP/IP suite and filesystems.

2.1 Designing Your Application

Should you decide to web-enable your application, you probably already have some idea of the format and layout of the web pages that will be presented to the browser. Unless the application only returns information and does not allow any updates (such as a data logger), you will probably need to lay out some *forms*. Forms, in web parlance, allow the browser's user to fill in some information then submit it to the server. The server, your application, then performs the requested actions and sends a confirmation back to the browser. This is the most common means for implementing control of the server as opposed to merely querying it.

There are several other things to consider. Answers to the following list of questions will determine the pieces of software that need to be gathered into your application, and how they link together.

- Does access to some or all resources need to be limited to a select set of users?
- If so, how confident does your application need to be that the user's credentials are valid?
- Do you need to be able to upload large amounts of data (over, say, 250 bytes)?
- Do you want to be able to update the web pages themselves, or maybe even the entire application firmware?
- Is the application small, medium, or large?
- Do you want to use this same (web) interface to configure all aspects of the application including, for example, the network settings? In other words, is the web interface going to be the *only* interface once the unit leaves the factory?

The first and second questions relate to *user authentication* and *access control*. The next two questions relate to the *HTTP upload* facility. The last two questions concern the overall design of your application; in particular, a large application may necessitate more storage than is usually available for a given Z-World product, and may require a sophisticated filesystem to manage the large number of resources.

Since the terms small, medium and large are rather vague, we shall define them by example. A small application would be limited to less than 10 different web pages, and up to about 30 different “controls” (buttons to press, dials to twiddle, options to select etc.). A large application may have upwards of 100 pages, and more than 10KB of configurable data. A medium application sits, as you might expect, near the middle of these.

Note that we are not considering the size of the application other than the web interface part. For example, you may have a sophisticated G-code interpreter and motion control system, where the web interface is limited to simply enabling/disabling the actuators and showing an error log to maintenance personnel. For the purposes of our discussion, this would be a small application.

The next section describes a “smaller-than-small” application, that is, a toy, which we use to show the bare essentials of a web-enabled application.

2.2 The Smallest Web Server in the WWW

Before moving on to real applications, the following sample code shows how to create the simplest possible web server. It does nothing but show “Hello WWW” on the browser. There are two files needed for this. The first is the Dynamic C code to be loaded to the target board (which must support TCP/IP). The second is the web page content itself, written in a syntax known as HTML (HyperText Markup Language). The second file is effectively included in the program, using the `#ximport` directive.

```
// toy_http.c
#define TCPCONFIG 1
#include "dcrtcp.lib"
#include "http.lib"
#include "helloworld.html" helloworld_html
SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html")
SSPEC_MIMETABLE_END
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/helloworld.html", helloworld_html)
SSPEC_RESOURCETABLE_END
void main() {
    sock_init();
    http_init();
    for (;;) http_handler();
}
```

The second file, named `helloworld.html` is coded as follows:

```
<HTML>
<HEAD><TITLE>Hello, WWW</TITLE></HEAD>
<BODY><H1>Hello, WWW</H1></BODY></HTML>
```

That's all there is to it. However, there is actually a lot of activity going on beneath the covers. For a start, the `#use "dcrtcp.lib"` directive and the `TCPCONFIG` macro definition bring in the TCP/IP networking suite and configure it. Unless you have a private test network, you probably have to modify the default setting - how to do that is beyond the scope of this chapter; it is described in volume 1 of the manual. The `#use "http.lib"` statement is required in order to bring in the web server. The next lines down to the start of the `main()` function are setting up tables that are consulted by the HTTP server and other libraries in order to "do the right thing." Finally, the `main()` function calls the necessary runtime initialization of the network and the HTTP server. It then calls the HTTP server in an endless loop, which drives the entire system into motion.

The `.html` file is ASCII text, in HTML syntax, which is transferred back to the browser when it is requested. Apart from the server adding some header lines, the `.html` file is transferred verbatim. This markup is merely telling the browser to display "Hello, WWW" as a 1st level heading, i.e., big bold text. This is specified by the second line. The first line adds a title to the page, which most browsers display in the window bar.

To see this web page on screen, the user needs to tell their browser what to get. If doing it manually, they would need to enter something like "http://10.10.6.100/helloworld.html" in the browser's URL entry field. The browser strips off the `http://10.10.6.100` part of it, and sends the rest to the specified host address (10.10.6.100) using a TCP connection to port 80 (interpreted from the `http://` part). The server gets the `/helloworld.html` part, which it knows about since it has a page of that name, and returns the contents of that file as a response. The browser interprets the HTML it receives, and generates a nice visual rendition of the contents.

2.3 Web Server Architecture

Before describing a real application, it is useful to know how such an application is organized. The following diagram shows all of the relevant components of a web-enabled application. There may seem to be a large number of components, however keep in mind that not all components need to be used by your application.

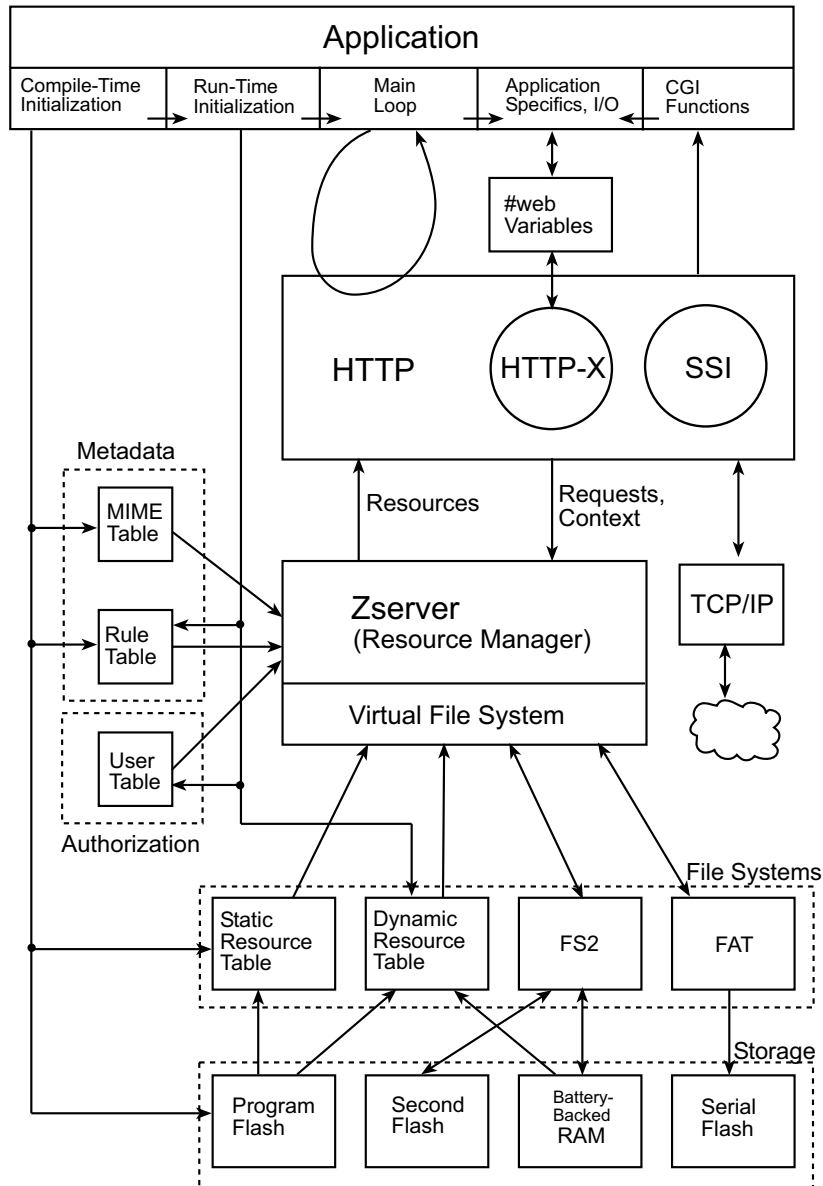


Figure 2.1. Components in a web-enabled application.

2.3.1 Application Block

At the top of this diagram is a block, called “Application,” consisting of five sub-blocks. The Application block represents the code that you have to create. Everything below this is provided by the libraries, although you will need to specify some parts of the interface to these components. This will be described in detail in the following sections.

The application block is subdivided into 5 parts:

1. **Compile-time initialization.** This includes things like selection of the appropriate library modules; initialization of static (constant) data structures and tables; selecting default network configuration; and inclusion of static resources (external files) via the `#ximport` or `#zimport` directives. The arrows leading from the “Compile-Time Initialization” sub-block indicate the tables that may be set up at compile time; namely:
 - The MIME type mapping table. This mandatory table indicates to the browser how the content is to be presented to the user. This is necessary for the browser, and needs to be specified by the server, however the server does not need to be particularly aware of the details.
 - The rule table. This is only necessary if a filesystem is in use. It is used by the resource manager to apply access permissions to the resources contained in a filesystem. This is necessary because not all filesystems can associate file ownership and access rights with individual files.
 - The static resource table. This is the classic method of defining resources in Dynamic C. This table is optional, since all necessary resources may be loaded in a filesystem, or in the dynamic resource table. Most applications will contain at least a few static resources, as an initial default or fallback, or for data that will never change such as a logo image.
 - Program flash. This really represents the loading of resource files into program memory via the `#ximport` directive. There will almost always need to be a few `#ximport` files, but this can be limited to a few kilobytes total.
2. **Runtime initialization.** Your `main()` function needs to call some specific library functions, once only, when it starts:
 - `sock_init()`. This is always mandatory. It initializes the TCP/IP networking system.
 - `sspec_automount()`. This is optional. It initializes the available filesystems (FS2 and/or FAT) for use by the resource manager, Zserver.
 - `http_init()`. This is mandatory. It initializes the HTTP server.
 - Various functions for setting up a user ID table, the rule table and/or the dynamic resource table. These are optional, but would be used in the majority of applications. The user ID table can only be initialized at run time, unlike the other tables that may, at least partially, be initialized at compile-time.
3. **Main loop.** The final code in the `main()` function continuously calls `http_handler()` and possibly other functions. This is mandatory, since it allows the HTTP server to process requests from the network. Other functions may be specific to your application. For example, you may need to poll an I/O device in order to obtain best performance.

4. Application specifics and I/O. This is really *your* part of the application or, if you like, the “back end” to the HTTP server. There are a number of ways that your application can communicate with the HTTP server. (These are not all shown on the diagram since it would add needless complexity.) Your application can directly call functions in the HTTP server, in the resource manager (Zserver), in TCP/IP, and just about anywhere else. One very clean and powerful interface is provided via #web variables. This is a new add-on module in Dynamic C 8.50.
5. CGI functions. CGI stands for “Common Gateway Interface,” however this acronym has a more specific use in Dynamic C—it refers to a C function that is called by the HTTP server to generate some dynamic content for the browser. This is the only truly optional block. Many applications can be written without resorting to CGI functions; however, there are some cases where the power and flexibility of a CGI will be required. Prior to Dynamic C 8.50, writing a robust CGI was the most difficult part of the entire process. From 8.50, there is a new style of CGI writing that simplifies the process, and reduces the chances of error. The old style of CGI is still supported for backwards compatibility.

2.3.2 HTTP Block

Let us now progress to the HTTP server itself. In the diagram, this is the block with two circles inside. The server is responsible for fielding requests from the outside world. Each request is analyzed to determine the resource that is being requested, the user who is making the request, and whether the user is authorized to obtain that resource. If the resource is available, the user is known and has the proper permissions, then the resource is transmitted back to the browser.

Following the above steps in more detail, we have:

1. Analyze the request: obtain the resource name. Part of the information provided by the browser is a request header that contains a URL (Uniform Resource Locator). The URL is simply the name of the resource to retrieve. URLs typically look like a file name in a Unix-style filesystem, that is, component directory and file names separated by slash (/) characters.
2. Obtain the user ID. The browser has the option of sending the username and password of its user. If it does not do this, then the userid is “anonymous.” If this is not good enough, then the browser can always try again when it is denied a protected resource. On receipt of user credentials (name and password), the HTTP server consults the resource manager (which in turn looks up the rule table) to see if the user’s credentials are OK. If they are, then the resource manager also determines the group(s) of which this user is a member. Thereafter, all access and permission checking is based on the group, not the individual user.¹
3. Return the resource. Having verified the group access rights (if necessary), the resource is transmitted back to the user. The resource may be an HTML or image file obtained from program memory or a filesystem, or it may be a script file that is processed “on the fly” to generate markup language. It may even represent a CGI function that will be called to generate all the necessary response. Note that a complete response requires a small amount of header information to be prefixed to the actual resource. The HTTP server usually takes care of this, however CGIs sometimes need to generate the header themselves.

1. This is a necessary optimization. There may be hundreds of individual users; however, the majority of these would be considered to be in a single “class,” with that class giving equal access to all its members. Considering the class, i.e., group, as the entity that is requesting a resource reduces the amount of information that needs to be stored.

Referring to the diagram in Figure 2.1, you can see that there are several arrows leading in and out of the HTTP server block. These represent lines of communication, and the arrow heads indicate the usual direction of data flow or, for function calls, “who calls whom.”

2.3.3 HTTP Block Subcomponents

The inner circles represent subcomponents of the server. The first of these, RabbitWeb, is a new add-on module available starting with Dynamic C 8.50. RabbitWeb is an extension to C language syntax to simplify presentation of C language objects (variables, structures) to a browser. RabbitWeb allows you to write web pages in a special scripting language. The script makes it easy to generate HTTP, which is the format expected by the browser. In addition, the script allows the contents of your C language objects to be turned into HTML fragments for presentation by the browser.

The small block named “#web Variables,” between the Application block and the RabbitWeb circle, indicates that the #web variables are the means of communication between your application and the server. Since #web variables are really just ordinary C variables, arrays or structures, they are extremely easy to manipulate by your application. Since they also have the property of being *registered* with the web server, the server has easy access too. (Registering an object with the web server is discussed in the document titled, “RabbitWeb,” available on our website.)

The second circle in the HTTP server block represents the classic way of generating dynamic content. SSI (Server Side Includes) is also a scripting language. It is not nearly as easy to use SSI as it is to use RabbitWeb; however, an SSI can generate the same content as an RabbitWeb script. It is just that you will need to write CGI functions, and such functions can get large and complicated fairly quickly! In fact, SSI has the ability to invoke CGI functions whereas RabbitWeb does not. In addition, SSIs have the ability to include other resources directly in the primary returned resource much like how #include works in ANSI C.

The server also communicates with lower layers in the diagram. On the right hand side is the TCP/IP block. This is the pipeline to the outside world, i.e., the browser. Usually only the server needs to talk directly to TCP/IP (via a TCP socket). Prior to Dynamic C 8.50, it was often necessary for the application’s CGI functions to call TCP/IP functions. This is no longer recommended. Instead, there are functions in the HTTP server that should be called to mediate all networking calls.

2.3.4 Zserver Block

Directly under the HTTP server block is the Zserver, or resource manager, block. This is the “central telephone exchange” of the entire application. It controls access to many of the other blocks in the diagram. In spite of its importance and central placing, you do not usually need to be aware of its inner workings. Zserver has applicability to other types of servers, such as FTP, because it provides a consistent interface to the various different types of resource. As indicated in the diagram, Zserver is architected as a resource manager and a virtual filesystem. The virtual filesystem is basically a notational convenience for accessing all resources using a uniform naming scheme. The external appearance of the virtual filesystem is modelled on the Unix approach. In Unix, all storage devices, and the filesystems contained therein, are accessed from a single starting point known as the root directory, written as a single slash (/) character. Under the root directory may be any number of files and directories. Some of these directories may actually refer to a completely different device and filesystem. The term for such directory entries is *mount-point*.

Note the distinction between this naming convention and the one used by (PC) DOS and similar operating systems. In DOS, you have to explicitly indicate the device by prefixing the file name. For example, C:\index.htm and A:\index.htm are different files, on different devices. On Unix you create two mount points in the root directory; /backup and /production for example. Then, the above mentioned files are known as /backup/index.htm and /production/index.htm. This may seem like a fine distinction, however it matches better with the naming convention used by HTTP, i.e., the URL. It also offers greater flexibility with regards to naming devices.

Zserver does not currently allow arbitrary mount-point names like Unix. Instead, there is an established convention for each filesystem. If FS2 is in use, then there is a mount-point called “/fs2.” If the FAT filesystem is in use, then one or more mount points called “/A,” “/B,” “/C” etc. are created.

Since Zserver is the resource manager, it takes responsibility for mapping the various filesystems and resource types into a single unified API. This API not only takes care of the detailed differences between the various filesystem APIs, but also allows some functions to be emulated that are not natively supported by the underlying filesystem.

In addition to the resource storage and filesystem, the resource manager needs to be able to associate other data with each resource. This other data is divided into two categories, which are listed in the blocks on the left of the diagram.

The two categories are “metadata” and “authorization.” Metadata consists of two tables: the MIME table and the Rule table. The authorization data is currently just a single table of userids. The reason for the split into two categories is this: the metadata is logically associated with individual resources, whereas the authorization data is a mapping from external entities (“users”) to the unit in which authorization is performed, namely user groups. The Rule table has some overlap, since it associates groups with individual resource permissions.

The lowest blocks in the diagram are divided into two groups, with a dashed outline. The upper group is labelled “filesystems,” and the lower “storage.” Both of these groups are indefinitely extensible, meaning that new classes of storage and their organization (filesystems) may be added in future releases of Dynamic C, or by you. The arrows between these groups are indicative of the most common patterns of communication; others may be defined.

2.4 Architecture of a Toy Application

Using the diagram of the previous section as a basis, we now focus on writing a simple web-enabled application. The following diagram is the same as the one above, except that the relevant parts have been visually emphasized. This diagram is essentially the toy application that was described at the start of this chapter. It shows the mandatory components for all web-enabled applications. Later, we introduce the other elements of the diagram to show how a fully featured application is built up.

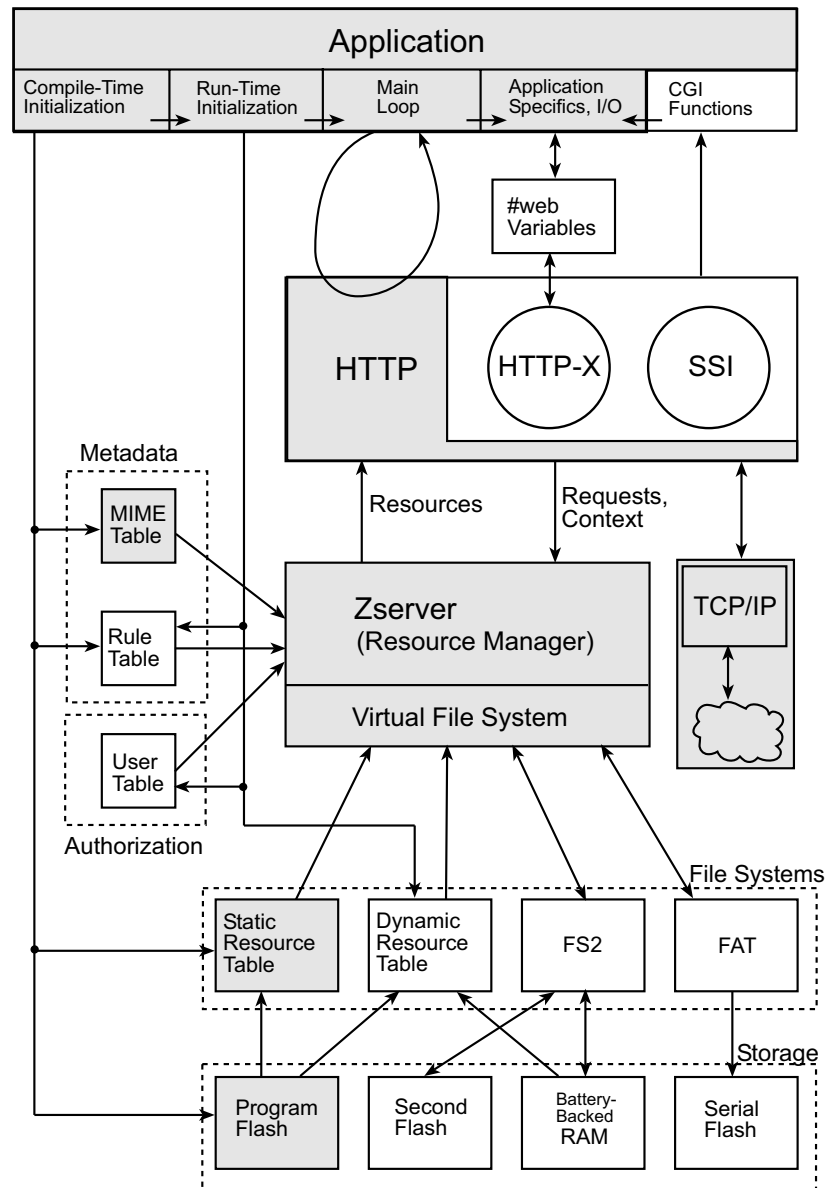


Figure 2.2. Minimum components for a web-enabled application.

Let us work again from left to right in the Application block. To reiterate, the Application block represents the coding that you have to do. First, there is the compile-time initialization. Taking the super-simple example illustrated in Figure 2.2, Dynamic C code is given with the relevant part highlighted in **boldface**.

```
#define TCPCONFIG 1
#use "dcrtcp.lib"
#use "http.lib"
#ximport "helloworld.html" helloworld_html
SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html")
SSPEC_MIMETABLE_END
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/helloworld.html", helloworld_html)
SSPEC_RESOURCETABLE_END
void main() {
    sock_init();
    http_init();
    for (;;) http_handler();
}
```

The first boldface line is the `#ximport` directive. This tells the compiler to include the specified file in the program flash, and make it accessible via the `helloworld_html` constant. In the diagram, the arrow from compile-time initialization to program flash represents this inclusion. In most cases you would be including more than just one file.

The three lines starting with `SSPEC_MIMETABLE_START` are initialization statements for the MIME table. In this case, there is a single mapping from resources that end with “.html” to a MIME type of “text/html.” All MIME types are registered with the relevant standards body, and must be entered correctly so that the browser does not get confused. “text/html” is the registered MIME type for HTML.

The next three lines, starting with `SSPEC_RESOURCETABLE_START`, set up the static resource table. Again, this contains a single entry that associates the resource name “/helloworld.html” with the file that was `#ximported` on the first line. Note that the resource name suffix (.html) matches the first parameter of the `SSPEC_MIME` entry.

Although not directly indicated on the diagram, the other compile-time initialization that is always required is the `#use` of the appropriate libraries. In this case, the first three lines create a default TCP/IP configuration (`TCPCONFIG = 1`) and bring in the networking and HTTP libraries. Note that `http.lib` automatically includes `zserver.lib`.

Back in the Application block of the diagram, we move right and consider the runtime initialization block. This is contained in the `main()` function. `sock_init()` comes first, to initialize the TCP/IP network library and bring up the necessary interface(s). `http_init()` resets the HTTP library to a known state.

The last statement embodies the Main Loop sub-block. This is always required. Typically, only `http_handler()` needs to be called; however, your application can insert calls to its own polling and event handling code. Since this is such a simple example, there is not even any application-specific code.

2.5 A Simple but Realistic Application

To turn the above toy example into something more realistic, we need to add some application specifics, and the ability to customize the resource returned to the browser depending on the relevant state of the application. The following diagram shows the necessary parts.

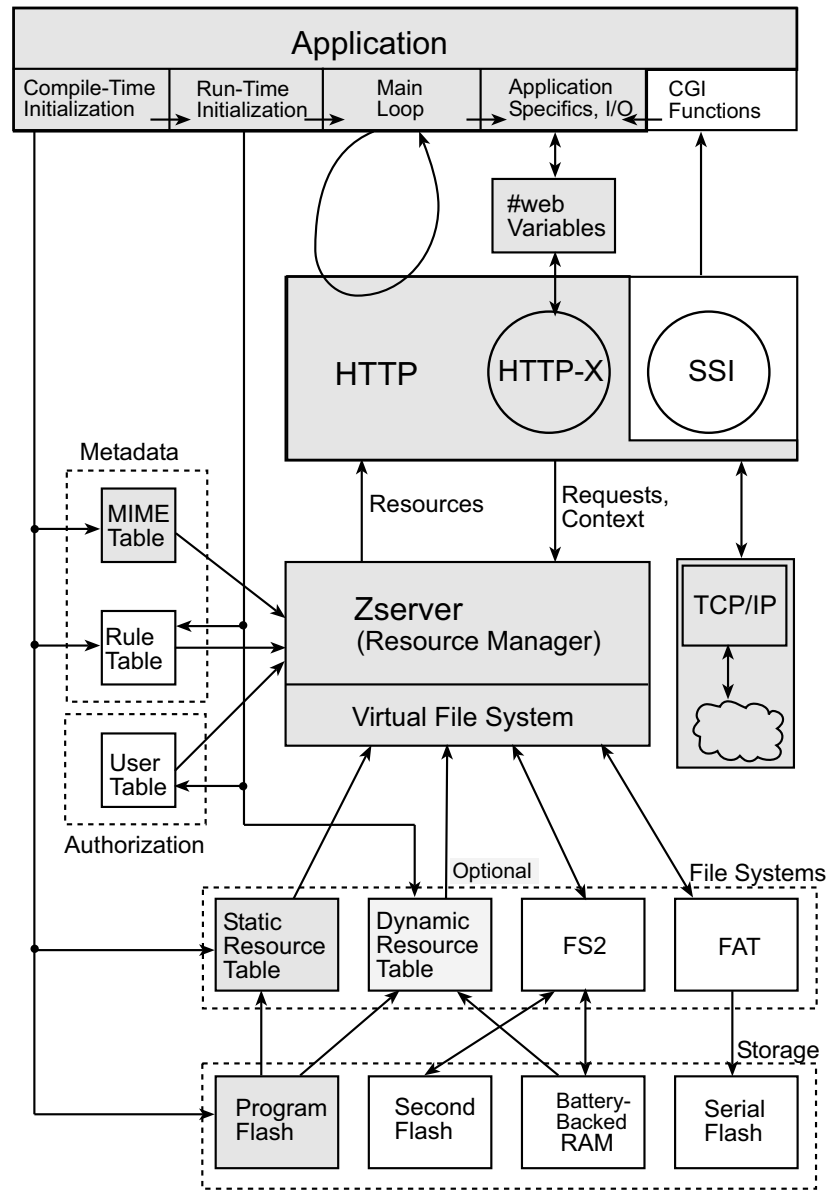


Figure 2.3. Minimum components for a web-enabled application with dynamic content.

The easiest way to introduce dynamic content is to use the RabbitWeb module and the associated scripting language. RabbitWeb is available as a Dynamic C add-on module starting with Dynamic C 8.50. If you do not have RabbitWeb, you can use SSI instead, which is described in Section 4.5.2.1 "SSI Feature." This example, illustrated in Figure 2.3, assumes that you have RabbitWeb.

The document titled, “RabbitWeb, To Web-Enable Embedded Applications” describes RabbitWeb and the scripting language, ZHTML, in detail. The following example is a simplification of the `web.c` sample program in `Samples\tcpip\rabbitweb`.

```
#define TCPCONFIG 1
#define USE_RABBITWEB 1
#use "dcrtcp.lib"
#use "http.lib"
#ximport "my_app.zhtml" my_app_zhtml
SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".html", "text/html", zhtml_handler),
SSPEC_MIMETABLE_END
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", my_app_zhtml)
SSPEC_RESOURCETABLE_END
int io_state;
#web io_state
void my_io_polling(void);
void main()
{
    sock_init();
    http_init();
    for (;;) {
        my_io_polling();
        http_handler();
    }
}
void my_io_polling()
{
    io_state = read_that_io_device();
}
```

The differences between the above code and the toy example in the previous section are in **bold-face**. All the differences relate to the use of RabbitWeb. The first addition is a `#define` of `USE_RABBITWEB`. This is necessary in order to include the necessary library code.

Next, there is a modification to the MIME table. The `SSPEC_MIME_FUNC` macro defines an entry that says that if the resource name ends with “.html” then the MIME type is `text/html` (as before), *and* there is a special scripting function that must be run by the HTTP server. This scripting function is called `zhtml_handler`; it is provided by the HTTP library. ZHTML is the unique embedded scripting language that converts script files into ordinary HTML so the browser can understand it.¹

1. Most applications will want to use a different resource suffix to distinguish between “ordinary” HTML files and script files. The samples provided with dynamic C use `.zhtml` for script files, and `.html` for plain HTML. In this sample, we only have script files, so it is convenient to retain the `.html` suffix. The other reason for this relates to the way the HTTP server handles requests for a directory. If given a URL of “/”, the HTTP server will append “index.html” to determine the actual resource. We take advantage of this default behavior so that this sample would work as expected.

The `int io_state` and `#web` statements define and register a web variable. Such a variable is an ordinary global variable as far as your C program is concerned. In addition, the script is able to access it.

`my_io_polling()` is a function that is part of the Application Specifics sub-block. As the name suggests, it is called regularly to poll some external device so as to keep the `#web` variable up-to-date. The implementation of the `my_io_polling()` function is shown updating the `#web` variable, but we don't specify the actual I/O reading function since that is too, well, application specific.

Now you may be wondering what this scripting language, ZHTML, looks like. The following code shows the contents of the `my_app.zhtml` file:

```
<HTML><HEAD><TITLE>Web Variables</TITLE></HEAD>
<BODY><H1>Web Variables</H1>
<P>The current value of io_state is
  <?z echo($io_state) ?>
</P>
</BODY></HTML>
```

This looks like plain HTML, and it is. The only difference is the existence of special commands flanked by “<?z” and “?>.” In this case, the command simply echos the current value of the web variable that was registered. The value (binary in the global variable) is converted to ASCII text by a default `printf()` conversion, in this case “%d” because the variable is an integer. When the browser gets the results returned by the HTTP server, it will see

```
<HTML><HEAD><TITLE>Web Variables</TITLE></HEAD>
<BODY><H1>Web Variables</H1>
<P>The current value of io_state is
  50
</P>
</BODY></HTML>
```

Where the “50” represents the current variable value—of course, it may be any decimal value that an integer variable could take: -32768 through 32767.

This is still a trivial example, but it is infinitely more real-world than the toy example. We have introduced the concept of dynamic content, which is required for embedded type applications. One thing that has been glossed over is how (and even whether) the variable can be updated from the browser, rather than just within the application. Yes, all `#web` variables may be updated via the browser. This requires use of HTML forms, which is a subject covered in the detailed documentation for RabbitWeb, and in the HTTP library chapter. We will not go over this again here; however, the possibility of remote updating introduces us to the topic of the next section, access control.

2.6 Adding Access Controls

If your application allows updating of the controller state via remote access, and the network connection allows access from locations that are not always under control, then it is important to add some access controls or “security.”

The most common way of doing this is to define a set of users, plus a method of authenticating those users, and attaching a set of “permissions” to each resource. The Dynamic C libraries allow you to do this fairly easily, via two tables. The relevant tables are:

The User Table

The user table contains a list of user IDs (short strings) and authentication information (currently a password string). Each user table entry also contains a group mask. The group mask indicates the user groups to which this user belongs. Up to 16 groups can be defined, and any given user can belong to one or more of these 16 groups. There are two additional masks in each user table entry. The first is a write access mask that indicates which server(s) allow the user to write (modify) its resources. The second mask indicates the server(s) that can recognize the user.

The Rule Table

The rule table is a list of information associated with each resource name, generally called “permissions.” Each resource has the following information:

- The realm (string) that may be used by certain servers (including HTTP).
- The group mask of the user groups that are allowed read-only access.
- The group mask of the user groups that are allowed modify/write access.
- The server(s) that are allowed any access to this resource.
- The authentication method that is recommended.
- The MIME type of the resource.

Resources in the static and dynamic resource tables may be set up to have their own specific permissions, independent of the rule table itself. Resources in a filesystem may be very numerous hence a simple one-to-one table would waste a lot of storage. To solve this problem, the rule table uses a name *prefix* matching algorithm. Using this technique, entire directories of resources need only have one rule table entry provided that all resources therein use the same permissions.

The following diagram shows the application components when access control is added:

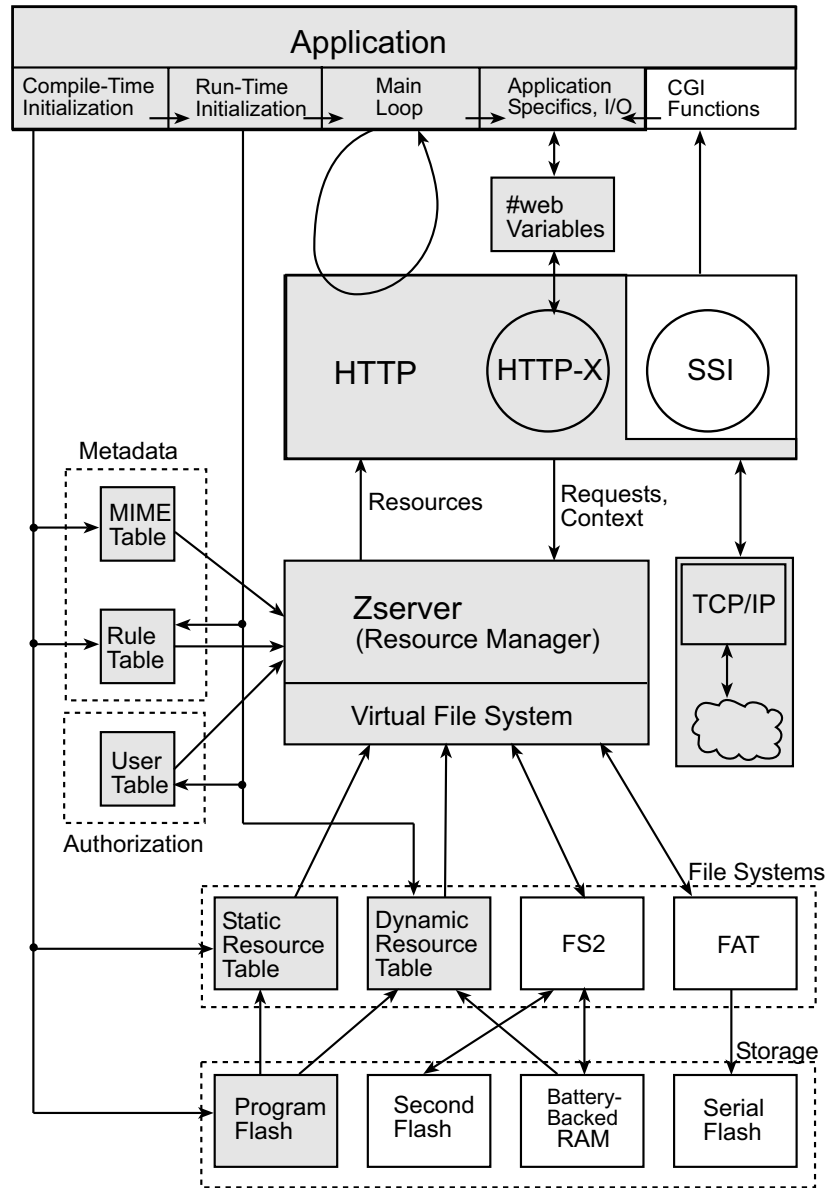


Figure 2.4. Minimal components of a web-enabled application with dynamic content and access control

The main difference between this and the previous diagram is that the Rule Table and User Table blocks have been activated.

The sample program is now expanded to add access control. As before, the changes are in **bold-face**.

```
#define TCPCONFIG 1
#define USE_RABBITWEB 1
#define USE_HTTP_BASIC_AUTHENTICATION 1
#use "dcrtcp.lib"
#use "http.lib"
#web_groups monitor_group, admin_group
#ximport "my_app.zhtml" my_app_zhtml
SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".html", "text/html", zhtml_handler),
SSPEC_MIMETABLE_END
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", my_app_zhtml)
SSPEC_RESOURCETABLE_END
int io_state;
#web io_state auth=basic groups=monitor_group,admin_group
void my_io_polling(void);
void main(){
    sspec_addrule("/index.html", "Pet",
        admin_group|monitor_group, 0, SERVER_HTTP,
        SERVER_AUTH_BASIC, NULL);
    sauth_setusermask(sauth_adduser("admin", "dog",
        SERVER_ANY), admin_group, NULL);
    sauth_setusermask(sauth_adduser("monitor", "cat",
        SERVER_ANY), monitor_group, NULL);
    sock_init();
    http_init();
    for (;;) {
        my_io_polling();
        http_handler();
    }
}
void my_io_polling()
{
    io_state = read_that_io_device();
}
```

The first change is the definition of `USE_HTTP_BASIC_AUTHENTICATION`. This sets up the HTTP server to be able to process this form of authentication. If not defined, then the server is unable to do this; there is little point in setting up any other access controls if the user cannot be verified!

Next, the user groups are defined. In this case, we are defining an “admin” and a “monitor” group. Presumably, the admin group has ability to alter the state of the controller, but the monitor group can only read its current state. The names `admin_group` and `monitor_group` are actually defined to be unsigned integer constants with just one bit set out of 16.

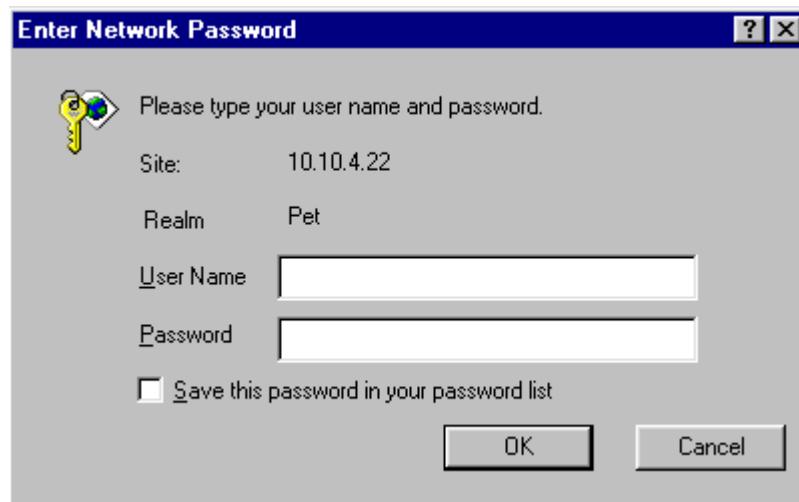
The #web registration of the `io_state` variable is augmented with some access controls. #web variables are not strictly resources—they are included as parts of other resources—however, they can be assigned some access controls of their own. In this example, access to the variable is being set to require “basic authentication,” and the allowable user groups are both of the defined groups, with the proviso that the monitor group is to be allowed read-only access.

The last major change is in the `main()` function, where some runtime initialization needs to be performed. Since the user ID table cannot be statically initialized (i.e., at compile-time), this is a necessary step. The rule table *can* be statically initialized, but in this example we choose to do it at runtime.¹ First, the rule table entry:

```
sspec_addrule("/index.html", "Pet",  
admin_group|monitor_group, 0, SERVER_HTTP,  
SERVER_AUTH_BASIC, NULL);
```

The first parameter specifies the name of the resource to which this rule applies; or rather, the first characters in the resource name. For clarity, the sample shows the full name. In practice, since there is only one resource, it would be acceptable to use just “/” instead of “/index.html.”

The second parameter, “Pet,” is an arbitrary string called the “realm.” This is presented to the browser’s user when prompted for the password, as shown here.



The third and fourth parameters indicate the group(s) that have read and write access to the resource. Both groups are allowed read access, and none write (0). Note that the resource in this case is the `index.html` page, *not* the variables which may or may not be displayed on it. Since this web page (actually a ZHTML script) is in program flash, it is obviously not modifiable.

The `SERVER_HTTP` parameter indicates that this resource is only visible from the HTTP server. This would be more relevant if there was another server, such as FTP, running concurrently.

1. In this example we also choose to use a rule table. This is not strictly necessary since no filesystem is in use. The alternative is to use a different form of initializing the static resource table, namely by using the `SSPEC_RESOURCE_P_XMEMFILE` macro, which allows permission information to be stored in the static table instead of in the rule table. See Section 3.2.5.3.

`SERVER_AUTH_BASIC` indicates that the server should use “basic authentication” when the browser calls for this resource. Note that Zserver does not enforce the method of authentication; it only stores the recommended method in the rule table. Any enforcement of authentication requires the co-operation of the server, since each different type of server may have widely different means of implementing the same type of authentication. Rest assured that the HTTP server (and other servers provided with Dynamic C) always enforce the suggested authentication method. The final NULL parameter allows some arbitrary data to be stored in the rule table entry. This data is available to the server. It is not currently used by any of the servers in Dynamic C, but it may be useful if you implement your own server.

Now, let’s turn to the user ID initialization:

```
sauth_setusermask(sauth_adduser("admin", "dog", SERVER_ANY),
    admin_group, NULL);
```

This is a nested function call. `sauth_adduser()` is called first, to add a user called “admin” with password “dog.” This user is visible to all servers (`SERVER_ANY`).

The result of this function call is a `userID` handle, which is the first parameter to `sauth_setusermask()`. This function explicitly assigns a group mask to the user. You can omit this call; however, the default method of assigning group masks is designed to be backward compatible with old versions of the library, and may not be what you want when using new features. You should always use the `sauth_setusermask()` function for each user ID.

In this example, we have added access control to the code. We do not need to change the ZHTML script, although in reality you would probably want to. Using the script unchanged, when the user tries to retrieve `index.html`, the browser will prompt for a `userid` and password. If one of the valid users is entered, then the page will be displayed. Otherwise, the browser will print an error message saying that access was denied.

Unfortunately, as written above, the sample will not allow us to test the distinction between the two users regarding the ability to modify the `#web` variable. We have shown how to add access control, but not how to actually specify a web form that allows the user to update the variable. It turns out that adding a form is not difficult. A modified script file is shown below. There is quite a lot to HTML forms, so most of the details are documented elsewhere. There are many good HTML reference books available.

```
<HTML><HEAD><TITLE>Web Variables</TITLE></HEAD>
<BODY><H1>Web Variables</H1>
<P>The current value of io_state is
    <?z echo($io_state) ?>
</P>
<?z if (error($io_state)) { ?>
    <P>Sorry, you were not authorized to perform an update.</P>
<?z } ?>
<FORM ACTION="/index.html" METHOD="POST">
    <P>Enter a new value if you dare:</P>
    <INPUT TYPE="text" NAME="io_state" SIZE=5
        VALUE="<?z echo($io_state) ?>">
    <INPUT TYPE="submit" VALUE="Submit">
    <INPUT TYPE="reset" VALUE="Reset">
</FORM>
</BODY></HTML>
```

If you run the above sample with this script, then the user will be able to attempt an update to the `#web` variable, `io_state`. If the user was “monitor,” that is, not able to make an update, then the “Sorry” message will be printed. Recall that the access to `io_state` was set up when the variable was registered with `#web`.

You may be asking how the application notices when the `#web` variable is updated by the browser, not just in the `my_io_polling()` function. This is a good question, since the HTTP server updates the variable just like a normal C variable. The solution to this requires that you specify an “update” callback function in the `#web` variable registration. This is described in detail in the Dynamic C Module document titled, “RabbitWeb: Web-Enabling Your Application;” for the purposes of this section please just remember that it is easy to do.

2.7 A Full-Featured Application

The previous examples have relied on `#import` to store files in the program flash. This is limiting in terms of storage capacity and does not allow for dynamic file updates. Adding the ability to store files in a filesystem that is located somewhere besides the program flash is of high value because it adds storage capacity and allows for dynamic updates.

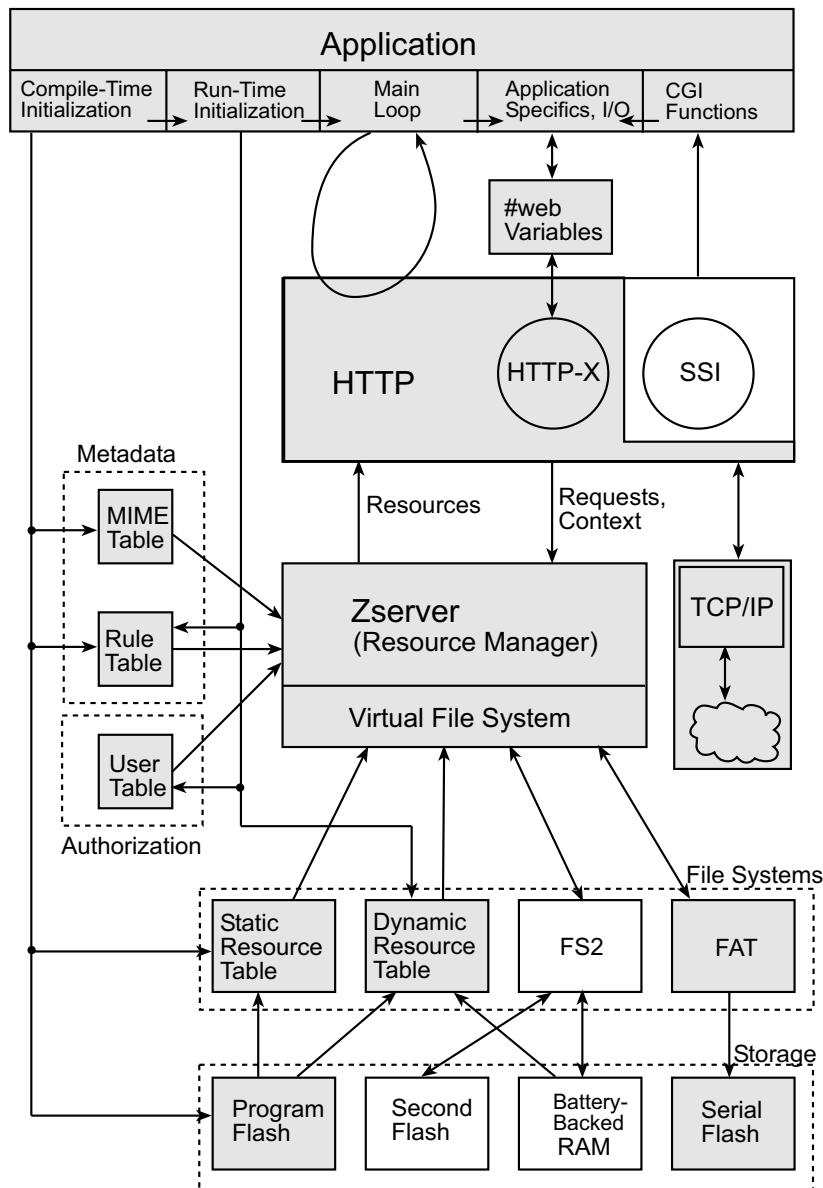


Figure 2.5. Components of a full-featured web-enabled application.

As mentioned previously, Zserver implements a virtual filesystem that can be used by an application for a clean, consistent interface to the various available methods of resource organization. An application can also bypass the resource manager and access a filesystem directly. (Note that there is no arrow in the diagram showing this line of communication.)

Looking at the bottom of the diagram in Figure 2.5 you can see that there are some additional hardware requirements when using FAT or FS2. The FAT needs a serial flash and FS2 needs a second flash or battery-backed RAM.

The sample program is now expanded to use a FAT filesystem and has the ability to upload files to it. As before, the changes are in **boldface**.

```

#define FAT_USE_FORWARDSLASH
#define FAT_BLOCK
#define USE_HTTP_UPLOAD
#define TCPCONFIG 1
#define USE_RABBITWEB 1
#define USE_HTTP_BASIC_AUTHENTICATION 1

#use "sflash_fat.lib"
#use "fat.lib"

#use "dcrtcp.lib"
#use "http.lib"
#web_groups monitor_group, admin_group
#ximport "my_app.zhtml" my_app_zhtml

SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".html", "text/html", zhtml_handler),
    SSPEC_MIME(".cgi", "")
SSPEC_MIMETABLE_END

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", my_app_zhtml),
    SSPEC_RESOURCE_CGI("upload.cgi", http_defaultCGI)
SSPEC_RESOURCETABLE_END

int io_state;
#web io_state auth=basic groups=monitor_group(ro),admin_group
void my_io_polling(void);
void main(){
    int rc;
    sspec_addrule("/index.html", "Pet",
        admin_group|monitor_group, 0, SERVER_HTTP,
        SERVER_AUTH_BASIC, NULL);
    sauth_setusermask(sauth_adduser("admin", "dog",
        SERVER_ANY), admin_group, NULL);
    sauth_setusermask(sauth_adduser("monitor", "cat",
        SERVER_ANY), monitor_group, NULL);
    rc = sspec_automount(SSPEC_MOUNT_ANY, NULL, NULL, NULL);
    if (rc)
        printf("Failed to initialize, rc=%d\n
            Proceeding anyway...\n", rc);
    sock_init();
    http_init();
    for (;;) {
        my_io_polling();
        http_handler();
    }
}

```

The first change is the addition of `FAT_USE_FORWARDSLASH` and `FAT_BLOCK`. These are needed by Zserver to work with the FAT filesystem. The definition of `USE_HTTP_UPLOAD` is needed for Zserver to use the file upload feature. Next, the libraries for the FAT (`fat.lib`) and for the serial flash driver (`sflash_fat.lib`) are brought in with `#use` statements.

The MIME type mapping for CGIs is added to the MIME table with `SSPEC_RESOURCE_CGI`. An empty string is the registered type for CGIs. This makes sense since CGIs are not displayed by the browser.

Next, we want to give the server access to the CGI function by creating an entry for it in the static resource table with `SSPEC_RESOURCE_CGI`. The first parameter is a string that must match the string used in the `FORM ACTION` tag in the HTML code. The second parameter identifies the CGI function that will be called when the form is submitted. `http_defaultCGI()` is a CGI that is provided with the HTTP server. It uploads files to a FAT filesystem, shows a status page to the browser after the upload and allows the user to click back to the server's home page. For a detailed description of the file upload feature, see Section 4.6.

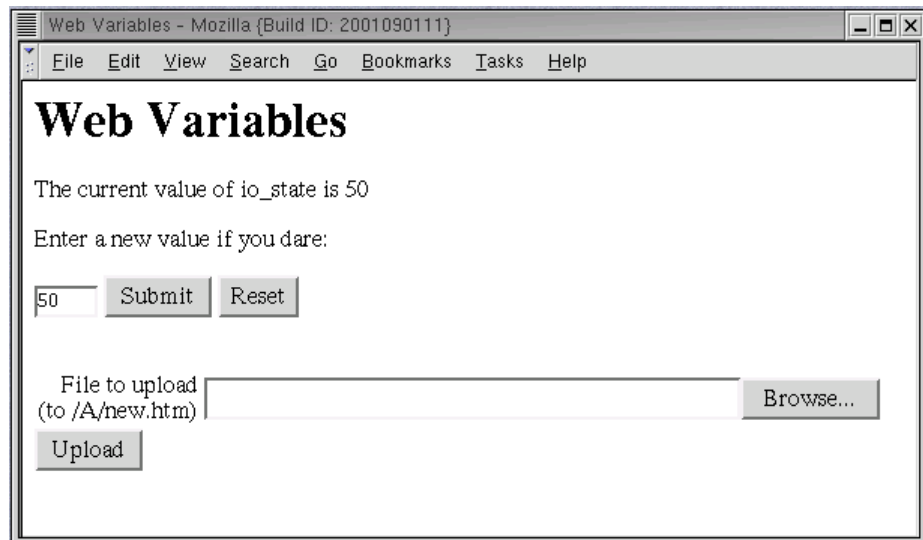
Finally, the FAT filesystem must be readied for use. The call to `sspec_automount()` takes care of everything, assuming that a FAT partition already exists on the serial flash. How to create the initial filesystem is discussed in the Dynamic C Module document titled, "Compact FAT Filesystem."

The application now supports uploading files to the FAT, but we have yet to give the user any way to actually do it. That involves changing the HTML page.

```
<HTML><HEAD><TITLE>Web Variables</TITLE></HEAD>
<BODY><H1>Web Variables</H1>
<P>The current value of io_state is
  <?z echo($io_state) ?>
</P>
<?z if (error($io_state)) { ?>
  <P>Sorry, you were not authorized to perform an update.</P>
<?z } ?>
<FORM ACTION="/index.html" METHOD="POST">
  <P>Enter a new value if you dare:</P>
  <INPUT TYPE="text" NAME="io_state" SIZE=5
    VALUE="<?z echo($io_state) ?>">
  <INPUT TYPE="submit" VALUE="Submit">
  <INPUT TYPE="reset" VALUE="Reset">
</FORM><BR>
<FORM ACTION="upload.cgi" METHOD="POST"
  enctype="multipart/form-data">
  <TABLE BORDER=0 CELLSPACING=2 CELLPADDING=1>
  <TR>
    <TD ALIGN=RIGHT>File to upload<BR>(to /A/new.htm)</TD>
    <TD><INPUT TYPE="FILE" NAME="/A/new.htm" SIZE=50></TD>
  </TR>
</TABLE>
  <INPUT TYPE="SUBMIT" VALUE="Upload">
</FORM>
</BODY></HTML>
```

The text in **boldface** is the description of a new form, which, when displayed by the browser, allows a file to be uploaded to a FAT filesystem.

The FORM tag includes the METHOD attribute, which is the same as that of the first form. The ACTION attribute has changed to specify the CGI function that was added to the server's static resource table; this is the



default CGI provided by the server. When the Upload button is clicked, `http_defaultCGI()` will be called by the server. A new attribute is included that specifies the MIME type used to submit the form to the server: `enctype="multipart/form-data"`. This is the MIME type required when the returned document includes files.

Note that the two forms are being submitted and processed separately. Could they be processed as one form? Yes, but from a modular design perspective, it makes sense to keep the form submissions separate when the purpose of each form is entirely separate.

You may have noticed that no security was added to protect the filesystem—anyone can upload a file that passed the initial user and password protection that limits access to the web page. This is probably not the ideal situation. Typically there needs to be some limit placed on who is able to write to the filesystem. When considering security, there are three possible things to protect:

- The web page that contains the form. Give read access only to those users who could conceivably upload the files specified therein.
- The CGI itself. Protect the same as the web page.
- The uploaded resource. You should set up a rule allowing write access only to the intended user(s).

When defining user IDs that can use the upload, don't forget to give those users overall write access using e.g.,

```
sauth_setwriteaccess(uid, SERVER_HTTP);
```

Another way to design this application is to have a separate HTML file that contains the form for the file upload; then instead of having the form for the file upload on the current HTML page, you put a link to the new page and then apply a permission to allow the new page to be displayed, such as:

```
sspec_addrule("/newpage.html", "Pet", admin_group,  
admin_group, SERVER_HTTP, SERVER_AUTH_BASIC, NULL);
```

That way the only people who see the Upload button are those authorized to use it. Design decisions such as these are guided by the needs of the application. The point here is that these design decisions are not limited by the underlying tools you are using to accomplish your goal.

2.8 Living Without RabbitWeb and FAT

Without the use of RabbitWeb we are back to SSI tags in the HTML page and writing a CGI to process them. With the new-style CGIs introduced in Dynamic C 8.50, this is easier than it used to be. If there is no serial flash, the FAT filesystem isn't available; but if there is a second flash or some battery-backed RAM, FS2 is. The following diagram shows the components that are used in this case. Note that even though both the second flash and the battery-backed RAM are high-lighted, an application can use either or both.

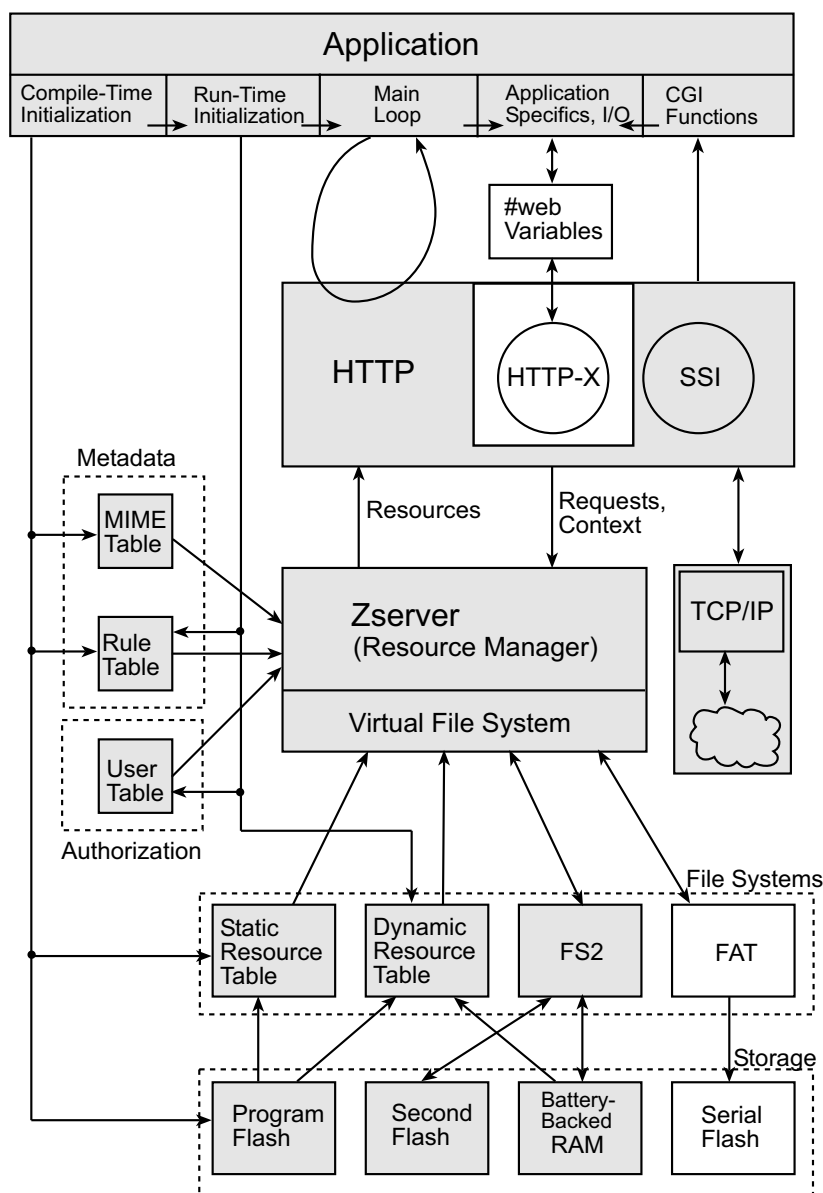


Figure 2.6. Components of a full-featured web-enabled application.

The sample program is now modified to use the FS2 filesystem. It still has the ability to upload files to the filesystem. As before, the changes are in **boldface**.

```
#define USE_HTTP_UPLOAD
#define TCPCONFIG 1
#define USE_HTTP_BASIC_AUTHENTICATION 1
#use "fs2.lib"

#define admin_group 0x0001
#define monitor_group 0x0002

#use "dcrtcp.lib"
#use "http.lib"

#ximport "my_app.shtml" my_app.shtml

SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".ssi", "text/html", shtml_handler),
    SSPEC_MIME(".cgi", "")
SSPEC_MIMETABLE_END

int io_state;

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_ROOTVAR("io_state", &io_state, INT16, "%d"),
    SSPEC_RESOURCE_XMEMFILE("/index.html", my_app.shtml),
    SSPEC_RESOURCE_CGI("upload.cgi", http_defaultCGI),
    SSPEC_RESOURCE_CGI("update.cgi", VarUpdateCGI)
SSPEC_RESOURCETABLE_END

void my_io_polling(void);

void main(){
    int rc;

    io_state = 42;

    sspec_addrule("/index.html", "Pet",
        admin_group|monitor_group, 0, SERVER_HTTP,
        SERVER_AUTH_BASIC, NULL);

    sauth_setusermask(sauth_adduser("admin", "dog",
        SERVER_ANY), admin_group, NULL);

    sauth_setusermask(sauth_adduser("monitor", "cat",
        SERVER_ANY), monitor_group, NULL);

    rc = sspec_automount(SSPEC_MOUNT_ANY, NULL, NULL, NULL);
    if (rc)
        printf("Failed to initialize, rc=%d\n
            Proceeding anyway...\n", rc);

    sock_init();
    http_init();

    for (;;) {
        my_io_polling();
        http_handler();
    }
}
```

The first change is the removal of the macros we added for FAT and also the removal of `#use` statements for the FAT library and the associated serial flash driver library. As with the sample in the last section, this code assumes that a valid filesystem partition exists on the target board; in this case, it's an FS2 partition. In the simplest case, which is one FS2 partition on the secondary flash, bringing in `fs2.lib` and then mounting the filesystem with a call to `spec_automount()` is all that is required. (For more information on FS2, refer to the *Dynamic C User's Manual*.)

The next change is the `#define` of the user groups. Each user group has to be explicitly given a value when RabbitWeb is not available to do it. Note that they are word values, each with a unique bit position set.

Next, the first entry in the MIME table was changed. Recall that the entry `"/` and requests without an extension are dealt with by the handler in the first entry of the MIME table. In this example, if a browser points to the Rabbit board's IP address, the page is processed by `shtml_handler()`, a handler that will understand the SSI tags that we are about to add to the HTML file. The `#ximport` statement did not, technically, need to change. The extension used for the file was changed from `.zhtml` to `.shtml`. These file extensions are only a convention. The important thing is that the HTML file is touched by the correct handler function. As a matter of fact, in this example, our HTML page is not recognized by the server as ending with either `.zhtml` or `.shtml`, but by `.html`. The name known to the server is determined by the name parameter of the file's resource table entry, `"/index.html`."

The next change is a new entry in the static resource table. This reflects the shift in how the variable `io_state` becomes known to the HTTP server. Previously, it was done using the `#web` statement of RabbitWeb.

A second new entry in the resource table is for a CGI function that will handle the processing when `io_state` is updated. When using RabbitWeb, this same form submission did not require a CGI. The enhanced HTTP server took care of all the details for us. Without RabbitWeb, we have to do the work ourselves. Fortunately, the new-style CGIs make this job easier. A detailed description of writing a new-style CGI is given in Section 4.6 "HTTP File Upload." As we saw in Section 2.7, there is a CGI in `http.lib` that processes file uploads to a filesystem. If you study and understand Section 4.6 and the code in `http_defaultCGI()`, you will be able to write a new-style CGI that will process the form that is submitted when `io_state` is changed.

Since we are not using RabbitWeb and have changed from using FAT to FS2, the HTML page must be changed. As before, all changes are in **boldface**.

```
<HTML><HEAD><TITLE>Web Variables</TITLE></HEAD>
<BODY><H1>Web Variables</H1>
<P>The current value of io_state is:
  <!--#echo var="io_state" -->
</P>
<FORM ACTION="update.cgi" METHOD="POST"
  enctype="multipart/form-data">
  <P>Enter a new value if you dare:</P>
  <INPUT TYPE="text" NAME="io_state" SIZE=5
    VALUE="<!--#echo var="io_state" -->">
  <INPUT TYPE="submit" VALUE="Submit">
  <INPUT TYPE="reset" VALUE="Reset">
</FORM><BR>
<FORM ACTION="upload.cgi" METHOD="POST"
  enctype="multipart/form-data">
  <TABLE BORDER=0 CELLSPACING=2 CELLPADDING=1>
  <TR>
    <TD ALIGN=RIGHT>File to upload<BR>(to /A/new.htm)</TD>
    <TD><INPUT TYPE="FILE" NAME="/fs2/ext1/new.htm"
      SIZE=50></TD>
  </TR>
</TABLE>
  <INPUT TYPE="SUBMIT" VALUE="Upload">
</FORM>
</BODY></HTML>
```

The first change is the substitution of the new server-parsed tags with SSI tags. The next change is the absence of any error checking. Without RabbitWeb, it is difficult to achieve this same functionality. The CGI responsible for the processing the variable update would need to do it. Which brings us to the next change in this HTML page, the need for a second CGI function.

The ACTION attribute in the FORM tag identifies the new CGI by name, `update.cgi`. The FORM tag also has a parameter for the encoding type. When no encoding type is specified, it defaults to URL-encoded. All new-style CGIs must set the encoding type in the FORM tag to “multipart/form-data” as shown above.

The other change on this page is the NAME attribute in the first INPUT tag of the second form. When uploading to an FS2 partition, the mount-point “/fs2” must be prepended to the filename. The /ext1 part is also prepended to the filename and refers to the second flash. The default CGI function can now store an uploaded file in a valid FS2 partition.

3. Server Utility Library

This chapter is intended to be a detailed description of the resource manager, `Zserver`, and how it interfaces to other libraries, such as servers (HTTP, FTP etc.) and filesystems (FS2, FAT). For an overview, please see Chapter 2. “Web-Enabling Your Application.”

The resource manager, `Zserver.lib`, contains the structures, functions, and constants to allow HTTP (Hypertext Transfer Protocol) and FTP (File Transfer Protocol) servers to share data and user authentication information while running concurrently.

In general, you do not need to know some of the details of `Zserver` described in this chapter if you are using the server libraries provided with Dynamic C. Such sections are marked as “advanced,” and you may skip them unless you are writing a server or filesystem. Some sections are marked “historical.” They are included to describe how previous versions of the library worked. These may be skipped for new code.

The basic facility provided by `Zserver` is the ability to translate resource names (URLs in the case of HTTP) into references to filesystem and memory objects. The term resource refers to the objects (files, functions and variables) that are manipulated by the `Zserver` library on behalf of the server. A *file resource* refers specifically to a resource of type file, as opposed to the actual file that is manipulated by an underlying filesystem (which may not be a resource as such).

Support for HTML forms is also included in `Zserver.lib`. Starting with Dynamic C 8.50, an enhanced HTTP server (RabbitWeb) is available that has an easy-to-use interface for form generation and no limitations on the form layout. See the document titled "RabbitWeb, To Web-Enable Embedded Applications" for more information on this enhanced HTTP server.

`Zserver` supports the concept of a virtual file system. This is modeled on the Unix directory structure.

3.1 Data Structures for `Zserver.lib`

There are several data structures in this library that servers with `Zserver` functionality must use, and may need to be manipulated or initialized by the application program:

- `MIMETypeMap`
- `ServerSpec`
- `ServerAuth`
- `ServerPermissions`
- `RuleEntry`

Use of the following structures is considered advanced:

- `ServerContext`
- `SSpecStat`
- `sspec_fatinfo`

The following structures are documented for historical reasons:

- FormVar
- SSpecFileHandle

3.1.1 MIMTypeMap Structure

This structure, organized into a table, associates a file extension with a MIME type (Multipurpose Internet Mail Extension) and a function that handles the MIME type. Users can override `HTTP_MAXNAME` (defaults to 20 characters) in their source file. If the function pointer given is `NULL`, then the default handler (which sends the content verbatim) is used.

```
typedef struct {
    char extension[10];
    char type[HTTP_MAXNAME];
    int (*fptr)(/* HttpState* */);
} MIMTypeMap;
```

For example, to create an HTTP server that can serve files with html or gif extensions, the following declaration is required in the application code:

```
SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME(".gif", "image/gif"),
SSPEC_MIMETABLE_END
```

Use of the above macros is the recommended method for maintaining forward compatibility. For more information, see Section 3.2.5.2 "Static MIME Type Table." All these macros are doing is generating the correct C syntax for a static constant initializer.

Note that servers that do not implement MIME, such as FTP, do not require a MIME table to be defined. Currently, this table is required only for HTTP.

3.1.2 ServerSpec Structure

This structure is used for both the static and dynamic resource tables. The only difference between these two tables is that one is a constant (initialized at compile-time) and the other is created at runtime in RAM, and thus modifiable.

Historical note: The `HttpSpec` data structure in `HTTP.lib` used prior to Dynamic C 8.50 is now synonymous with this structure, `ServerSpec`.

```
typedef struct {
    word type;
    char name[SSPEC_MAXNAME];
    long data;
    void *addr;
    word vartype;
    char *format;
    ServerPermissions perm;
#ifdef FORM_ERROR_BUF
    ...
#endif
} ServerSpec;
```

The structure fields are described below. The `#ifdef` expression adds some fields to the `ServerSpec` structure if the HTML form functionality provided by Zserver is included by the web server application. These fields are not described below. For more details, Section 4.5.4 "HTML Forms Using Zserver.lib."

Starting with Dynamic C 8.50, enhanced support is provided for HTML forms with the purchase of the Dynamic C RabbitWeb Module. This module provides an easy to develop web interface for your embedded device and allows for complete flexibility in form layout. See the document titled "RabbitWeb, To Web-Enable Embedded Applications" for more information on this enhanced HTTP server.

In older versions of Dynamic C, it was necessary to explicitly create the static resource table by doing something like this:

```
const HttpSpec http_flashspec[] = {
    ...
};
```

in your main application code (filling in the entries, of course). Starting with Dynamic C 8.50, there is now recommended syntax for creating these resources, using the `SSPEC_RESOURCE*` series of macros. This new method is recommended for maintaining future compatibility. For more information, see Section 3.2.5.3 "Static Resource Table."

3.1.2.1 ServerSpec Fields

The fields in each resource table (static or dynamic) are usually manipulated via Zserver functions, or by using the `SSPEC_RESOURCE*` macros. The field descriptions below are for reference only.

type	This field tells the server if the entry is a file, variable or function (<code>SSPEC_FILE</code> , <code>SSPEC_VARIABLE</code> , <code>SSPEC_FUNCTION</code> , etc.).
name	This field contains the resource name, as a null-terminated string.
data	Location of data (when <code>*FILE</code> is the type of data), or maximum number of variables in a form (when <code>SSPEC_FORM</code> is the type of data)
addr	Address of function or variable (when <code>SSPEC_FUNCTION</code> , <code>SSPEC_CGI</code> or <code>SSPEC_VARIABLE</code> is the type of data). Address of form struct for <code>SSPEC_FORM</code> .
vartype	Type of variable (when <code>SSPEC_VARIABLE</code> is the type of data), or length of data (when <code>*FILE</code> is the type of data and the length is needed e.g., a root file). For <code>SSPEC_HARDLINK</code> , contains the <code>sspec</code> index number of a <code>http_flashspec</code> or <code>server_spec</code> entry.
format	<code>printf()</code> format for a variable, or form title for a form, or base address for <code>SSPEC_ROOTFILE</code> . For <code>SSPEC_LINK</code> , points to a string containing the linked-to resource name.
perm	Permissions associated with this resource. If realm subfield is <code>NULL</code> , then the permissions table is consulted as for filesystem resources. Note: this field used to be <code>char*</code> for the realm string (only). Programs that used this feature need to be modified. This structure is detailed under <code>ServerPermissions</code> .

There are some other fields that are conditionally included if HTTP forms are in use. These are not generally relevant. See the library source for details.

3.1.3 ServerAuth Structure

This structure defines a global array that is a list of user name/password pairs.

```
ServerAuth server_auth[SAUTH_MAXUSERS];
```

Throughout this manual, this array is called the user table. The fields in the `ServerAuth` struct are manipulated using the `sauth_*()` functions. The description below is for reference only.

username	Name of user, or ""
password	Password, or ""
mask	Group mask
writeaccess	Which servers this user has write access to
servermask	Which servers this user is visible to
data	Arbitrary data (application-dependent)

3.1.4 ServerPermissions Structure

This data structure holds access permissions for a resource or a group of resources. An instance of `ServerPermissions` is contained in each `ServerSpec` structure, as well as within each rule table entry. The fields for the `ServerPermissions` struct are:

realm	Pointer to realm string of the resource. It is only used by HTTP servers, but can be used for other purposes.
readgroups	Read permission is granted if the current <code>ServerAuth.mask</code> value matches in at least one bit position.
writigroups	Write permissions is granted if the current <code>ServerAuth.mask</code> value matches in at least one bit position and <code>ServerAuth.writeaccess</code> is set.
servermask	A 16-bit mask with a bit set for each server that can access this resource. NB: for backwards compatibility, if this is set to zero then all servers are allowed.
method	Authentication method(s) allowed: combination of <code>SERVER_AUTH_*</code> bits. Note that <code>Zserver.lib</code> does not directly support anything other than basic authentication, that is <code>SERVER_AUTH_BASIC</code> ; however, the required information is stored here so that servers can access it as needed in a consistent manner.
mimetype	MIME type for this resource, or NULL. If NULL, the MIME type will be derived from the file name using the <code>MIMETYPEMap</code> table called <code>http_types</code> . If not found in that table, the first entry in that table will be used (for backward compatibility.)

Historical note: Prior to Dynamic C 8.50, `HttpRealm` was used in place of `ServerPermissions`. If you have used `HttpRealm` for password protection in existing

code and are upgrading to Dynamic C 8.50 or later, you must rewrite any code that used this old structure. For an example of the new way to password protect an entity, see the sample program `samples\tcpip\http\authentication.c`.

3.1.5 RuleEntry Structure

This structure associates a resource name prefix with a `ServerPermissions` structure. The rule table is an array of these structures.

prefix	Prefix of resource name(s) which are associated with this rule table entry. If there are multiple entries which match a resource name, then the rule with the longest matching prefix is used.
perm	<code>ServerPermissions</code> to use for this entry.

3.1.6 ServerContext Structure

Starting with Dynamic C 8.50, context information must be maintained by each server that wants Zserver functionality. Therefore, servers must provide a `ServerContext` struct when required. The fields of `ServerContext` are:

userid	This field identifies the current user.
server	This field identifies the server, for example, <code>SERVER_HTTP</code> . This is one of the few cases where only a single server bit should be set.
rootdir	This field is a pointer to the root directory. This is usually “/” if the whole namespace is to be accessible. Otherwise, it may be, for example, “/A” to restrict access to just the first DOS FAT partition. The first and last character must be “/”!
cwd[]	This field is an array containing the current working directory. This would normally contain the root directory as a prefix. The first and last character must be “/”!
dfltname	This field points to a file name to be used as a resource name suffix when the first parameter refers to a directory name.

The `ServerContext` structure helps support more powerful resource access control. It is needed by several of the new API functions that deal with resource retrieval and control, as well as functions that perform directory navigation.

There are two functions that return a `ServerContext` struct: `http_getcontext()` and `http_getContext()`. The latter is for use in CGI functions.

These functions can be used with other API functions that need the context structure. For example:

```
sspec_open("MyFile", http_getcontext(servno), O_READ, 0);
```

will open “MyFile” for reading for the server instance identified by `servno`.

3.1.7 SSpecStat Structure

This structure holds status information about a file resource. It is filled in by the function `sspec_stat()`.

The fields of `SSpecStat` are:

flags	A 16-bit mask that passes information about the file resource. The <code>flags</code> field can be any number of the following: <ul style="list-style-type: none">• <code>SSPEC_ATTR_MDTM</code> - have modification date/time• <code>SSPEC_ATTR_LENGTH</code> - have current length• <code>SSPEC_ATTR_WRITE</code> - file is writable• <code>SSPEC_ATTR_EXEC</code> - file is "executable"• <code>SSPEC_ATTR_HIDDEN</code> - "Hidden" attribute bit• <code>SSPEC_ATTR_SYSTEM</code> - "System" attribute bit• <code>SSPEC_ATTR_ARCHIVE</code> - "Archive" attribute bit• <code>SSPEC_ATTR_DIR</code> - directory name• <code>SSPEC_ATTR_COMPRESSED</code> - stored in compressed format• <code>SSPEC_ATTR_MAXLENGTH</code> - have maximum length• <code>SSPEC_ATTR_SEEKABLE</code> - resource is randomly accessible• <code>SSPEC_ATTR_EXTENSIBLE</code> - File may be expanded at end
mdtm	Modification date/time (<code>SEC_TIMER</code> format), this field is only valid if <code>SSPEC_ATTR_MDTM</code> is set.
length	The current file size; this field is only valid if <code>SSPEC_ATTR_LENGTH</code> is set.
maxlength	The maximum allowable file size; this field is only valid if <code>SSPEC_ATTR_MAXLENGTH</code> is set.
perm	Pointer to <code>ServerPermissions</code> struct. This structure is described above.

3.1.8 sspec_fatinfo Structure

This structure is only relevant if you are using the FAT filesystem. It allows the `sspec_automount()` function to return some FAT-related information to your application.

The fields in this structure are:

ctrl	Pointer to <code>dos_ctrl</code> (controller) structure.
drive	Pointer to <code>mbr_drive</code> structure.
part[4]	4 pointers to <code>fat_part</code> (partition) structures. Only the mounted partitions are returned.

Note that when used with `sspec_automount()`, some of the above fields may be set to non-NULL in order to indicate to `sspec_automount()` that the application has already initialized some or all of the FAT.

3.1.9 FormVar Structure

An array of `FormVar` structures represent the variables in an HTML form. The developer will declare an array of these structures, with the size needed to hold all variables for a particular form. The `FormVar` structure contains:

- A `server_spec` index that references the variable to be modified. This is the location of the form variable in the server spec list.
- An integrity-checking function pointer that ensures that the variables are set to valid values.
- High and low values (for numerical types).
- Length (for the string type, and for the maximum length of the string representations of values).
- A Pointer to an array of values (for when the value must be one of a specific, and probably short, list).

The developer can specify whether the variable is set through a text entry field or a pull-down menu, and if the variable should be considered read-only.

This `FormVar` array is placed in a `ServerSpec` structure using the function `sspec_addform()`. `ServerSpec` entries that represent variables will be added to the `FormVar` array using `sspec_addfv`. Properties for these `FormVar` entries (for example, the integrity-checking properties) can be set with various other functions. Hence, there is a level of indirection between the variables in the forms and the actual variables themselves. This allows the same variable to be included in multiple forms with different ranges for each form, and perhaps be read-only in one form and modifiable in another.

3.1.10 SSpecFileHandle Structure

This structure is used internally by Zserver, and is only of interest to developers of new filesystems which may be incorporated into Zserver.

3.2 Constants Used in Zserver.lib

The constants in this section are values assigned to the fields of the structures `ServerSpec` and `ServerAuth`. They are used in the functions described in Section 3.5, some as function parameters and some as return values.

3.2.1 ServerSpec Type Field

This field describes the resource in the server spec list. The possible values are:

- `SSPEC_XMEMFILE` - The data resides in xmem
- `SSPEC_ZMEMFILE` - The data resides in xmem and is compressed
- `SSPEC_ROOTFILE` - The data resides in root memory
- `SSPEC_FSFILE` - The data resides in an FS2 file.
- `SSPEC_FATFILE` - The data resides in a DOS FAT file.
- `SSPEC_FILE` - The data resides in a file - generic type returned by `sspec_gettype()`.
- `SSPEC_ROOTVAR` - The data is a variable in root memory (for HTTP)
- `SSPEC_XMEMVAR` - The data is a variable in xmem (for HTTP)

- `SSPEC_VARIABLE` The data is a variable (for HTTP) - generic type returned by `sspec_gettype()`.
- `SSPEC_FUNCTION` - The data is a function (for HTTP.)
- `SSPEC_FORM` - A set of modifiable variables.
- `SSPEC_CGI` - The data is a CGI function (for HTTP) - new style CGIs with better interface.
- `SSPEC_LINK` - Symbolic link ("alias") to another resource name.
- `SSPEC_HARDLINK` - Symbolic link ("alias") to another resource table entry.

3.2.2 ServerSpec Vartype Field

If the object is a variable, then this field will tell you what type of variable it is:

```
INT8, INT16, INT32, PTR16, FLOAT32
```

3.2.3 ServerPermissions Servermask Field

The type of server (HTTP and/or FTP) that has access to a particular resource is determined by the `servermask` field in the `ServerPermissions` structure.

- `SERVER_HTTP` - Web server
- `SERVER_FTP` - File transfer server
- `SERVER_SMTP` - Mail server
- `SERVER_HTTPS` - Secure web server
- `SERVER_SNMP` - SNMP agent
- `SERVER_USER` - Placeholder for first user-defined server
- `SERVER_USER2` - Placeholder for second user-defined server (etc.) - grow down.
- `SERVER_ANY` - Any server. May be passed in most cases when any server will do.

3.2.4 Configuration Macros

There are several configuration macros that may be set up by the application to control the memory usage and behavior of Zserver. These should be defined before `#use Zserver.lib`, unless otherwise noted.

HTTP_NO_FLASHSPEC

SSPEC_NO_STATIC

When defined, these macros saves space by not compiling in code that supports a static resource table. Presumably the application is using only the dynamic resource table, or filesystems are in use. Historical note: the name of `HTTP_NO_FLASHSPEC` implies HTTP, however it actually applies to Zserver as a whole, not any specific server. Dynamic C 8.50 introduces `SSPEC_NO_STATIC`, an alias for `HTTP_NO_FLASHSPEC`.

SAUTH_MAXNAME

Maximum length of the name and password strings in the `ServerAuth` structure. Default is 20. Strings must include a NULL character, so with its default value of 20, strings in this structure may be at most 19 characters long.

SAUTH_MAXUSERS

Define the maximum number of unique users. Defaults to 4. This determines the size of the userid table. Each table entry takes up $2 * SAUTH_MAXNAME + 8$ bytes of root storage.

SERVER_PASSWORD_ONLY

This is set to a bitmask of the server mask bits for each server that supports the concept of a password-only user, that is, no user name. Defaults to zero since currently no servers are implemented that use this facility.

SSPEC_DEFAULT_READGROUPS

SSPEC_DEFAULT_WRITEGROUPS

SSPEC_DEFAULT_SERVERMASK

SSPEC_DEFAULT_REALM

SSPEC_DEFAULT_METHOD

This group of macros establishes global default permissions for resources that do not have a rule associated. `SSPEC_DEFAULT_READGROUPS` is "0xFFFF" which means "all users." For writegroups, this is "0" meaning "no users." The servermask defaults to `SERVER_ANY` (all servers can access). realm defaults to "" that is, an empty string, or no realm. `SSPEC_DEFAULT_METHOD` defaults to no authentication method required.

SSPEC_MAX_FATDRIVES

Determine the maximum number of independent FAT filesystem "drives." Defaults to 1. Each drive takes 8 bytes of root storage (plus whatever is required by the filesystem itself). Each drive can have up to 4 partitions. This macro is only relevant if you use the FAT library.

SSPEC_MAXNAME

Define the maximum name length of each dynamic or static resource. Defaults to 20. You can minimize memory usage by choosing short names for all resources, and reducing the value of this macro.

SSPEC_MAXRULES

Define the maximum number of dynamically added "rules." Defaults to 10, but you can explicitly define it to zero if all the rule table entries are static (see `SSPEC_RULETABLE_*` macros). Each rule takes up 13 bytes of root storage, plus whatever storage is required for the realm and prefix strings (which must be null-terminated, and in static storage, since pointers to these are stored in the rule table).

SSPEC_MAXSPEC

Define to the number of dynamic (RAM) resource table entries to allocate for the global array, `server_spec`. Each entry takes `SSPEC_MAXNAME + 23` bytes of root memory (or `SSPEC_MAXNAME + 33` if `FORM_ERROR_BUF` is defined).

Defaults to 10 entries (approximately 530 bytes). Do not set higher than 511.

SSPEC_MAX_OPEN

Determine the maximum number of simultaneously open resources. Defaults to 4. Choose this number carefully, since each entry can take up a fairly large amount of root storage, depending on the mix of filesystems in use. Unless you are anticipating a very busy server, 4 should be enough.

If you increase the default value of `HTTP_MAXSERVERS` from 4, you may experience 404 or 503 messages. The solution is to increase `SSPEC_MAX_OPEN`. Ideally, this value should be `HTTP_MAXSERVERS + FTP_MAXSERVERS + any special use of zserver.lib` that you create.

SSPEC_XMEMVARLEN

Defines the size of the stack-allocated buffer used by `sspec_readvariable()` when reading a variable in `xmem`. It defaults to 20.

3.2.5 Macros for Control Data Initialization

As of Dynamic C 8.50, the following macros are available for building the static tables used by the servers.

3.2.5.1 Static Rule Table

Resource rules are used to associate access information with resource names. The following macros define and initialize a static rule table. If using a static rule table, the dynamically added entries will be searched before the static ones.

SSPEC_FLASHRULES

Define this if your application is using static rules. You must define this if you want to use the macro `SSPEC_RULETABLE_START`. If you define `SSPEC_FLASHRULES`, and you do not need dynamic rules, you can define the macro `SSPEC_MAXRULES` to zero to recover the root memory that would be wasted otherwise.

SSPEC_RULETABLE_START

SSPEC_RULE(prefix, realm, rg, wg, sm)

SSPEC_MM_RULE(prefix, realm, rg, wg, sm, method, mimetype)

SSPEC_RULETABLE_END

This sequence of macros is used to define static rules. See the documentation with the `sspec_addrule()` function for more information. You must define `SSPEC_FLASHRULES` to use these macros.

3.2.5.2 Static MIME Type Table

This table maps file extensions and MIME types. You only need such a table if using a server that requires MIME types. Currently, only the HTTP server needs this.

```
SSPEC_MIMETABLE_START
```

```
SSPEC_MIME(extension, type)
```

```
SSPEC_MIME_FUNC(extension, type, function)
```

```
SSPEC_MIMETABLE_END
```

This sequence sets up the MIME type mapping table. Currently only a static MIME table is supported. Though you cannot dynamically add new MIME types to this table, it is possible to allocate new `MIMETYPEMap` structures in RAM and assign them to specific resources using `sSpec_addrule()` or `sSpec_setpermissions()`.

Such entries will not be accessed using the default resource name extension method.

3.2.5.3 Static Resource Table

The static resource table associates the names of web server resources (files, functions, and variables) to references to memory objects.

HTTP_NO_FLASHSPEC

Define if there is to be NO static resource table, that is, all resources are in the dynamic (RAM) table or in the filesystem(s). If you define this, then there is no point in using the `SSPEC_RESOURCE_*` series of macros below.

```
SSPEC_RESOURCETABLE_START
SSPEC_RESOURCE_ROOTFILE(name, addr, len)
SSPEC_RESOURCE_XMEMFILE(name, addr)
SSPEC_RESOURCE_ZMEMFILE(name, addr)
SSPEC_RESOURCE_FSFILE(name, fnum)
SSPEC_RESOURCE_ROOTVAR(name, addr, type, format)
SSPEC_RESOURCE_XMEMVAR(name, addr, type, format)
SSPEC_RESOURCE_FUNCTION(name, addr)
SSPEC_RESOURCE_CGI(name, addr)
SSPEC_RESOURCE_P_ROOTFILE(name, addr, len, realm, rg, wg, sm, meth)
SSPEC_RESOURCE_P_XMEMFILE(name, addr, realm, rg, wg, sm, meth)
SSPEC_RESOURCE_P_ZMEMFILE(name, addr, realm, rg, wg, sm, meth)
SSPEC_RESOURCE_P_FSFILE(name, fnum, realm, rg, wg, sm, meth)
SSPEC_RESOURCE_P_ROOTVAR(name, addr, type, format, realm, rg, wg,
    sm, meth)
SSPEC_RESOURCE_P_XMEMVAR(name, addr, type, format, realm, rg, wg,
    sm, meth)
SSPEC_RESOURCE_P_FUNCTION(name, addr, realm, rg, wg, sm, meth)
SSPEC_RESOURCE_P_CGI(name, addr, realm, rg, wg, sm, meth)
SSPEC_RESOURCETABLE_END
```

These macros are used to initialize the static resource table. Prior to Dynamic C 8.50 this had to be done by explicitly using C language initialization of a table declared as:

```
const HttpSpec http_spec[]
```

These macros perform the same function. It is recommended to use them instead of static initializers in order to maintain forward compatibility.

The macros with `_P_` in the name are the same as the others, except that they explicitly allow all the server permissions information (except for the MIME type mapping) to be initialized. See `sspec_addrule()` for more information on the parameters.

The name parameter to all these macros is the resource name. This usually starts with a `"/` for files, but not for variables. The string length should be less than or equal to `SSPEC_MAXNAME`.

The other parameters depend on the resource type being created:

ROOTFILE: `addr` = root memory address of first byte of file, `len` = length of file (0..32767).

XMEMFILE: `addr` = longword (physical address) of the length word of the file. The length word (4 bytes) is followed by the first byte of data.

ZMEMFILE: as for XMEMFILE, except the file is compressed and imported using `#zimport` instead of `#ximport`.

FSFILE: `fnum` = FS2 file number of file (1..255)

ROOTVAR: `addr` = root memory address of data, `type` = type of data, as documented with `sspec_addvariable()`, `format` = `char * format`, as used by `printf()`. For example, “%d” for a decimal number.

XMEMVAR: as for ROOTVAR except the address is a longword physical address.

FUNCTION or CGI: `addr` = address of C function.

Note that a maximum of 511 static resource table entries can be defined.

3.3 File Compression Support

Dynamic C 8.50 introduces file compression support. The sample program `/samples/tcpip/http/zimport.c` demonstrates how to use this functionality. This sample is oriented towards the HTTP server; however, under the covers, HTTP is relying on Zserver to perform the compressed file handling.

In the sample program, notice that the statement `#use zimport.lib` comes before the statement `#use http.lib` in the code. This is required to have file compression support in Zserver and the web server. The next thing to notice is the use of the compiler directive `#zimport` instead of `#ximport`. `#zimport` performs a standard `#ximport`, but compresses the file by invoking a compression utility before emitting the file to the target.

When adding a compressed file to the static resource table, use the macro `SSPEC_RESOURCE_ZMEMFILE` instead of `SSPEC_RESOURCE_XMEMFILE`. When you add a compressed file to the dynamic resource table using the `sspec_addxmemfile()` function, it will be recognized as a compressed file automatically. `sspec_addxmemfile()` is thus used for both compressed and uncompressed imported files.

Each instance of a server will use a buffer for decompression—this is necessary since multiple server instances can be decompressing files at the same time. Make sure that the buffer macro `INPUT_COMPRESSION_BUFFERS` is at least as large as the number of servers which may need concurrently to decompress a compressed resource. The buffer macro describes the number of 4KB xmem RAM buffers used for decompression. This definition is used by the `zimport.lib` library.

For details on compression ratios, memory usage and performance, please see Technical Note 234, “File Compression.” For more information on using `#zimport` and the support libraries, please see the *Dynamic C User’s Manual* and the *Dynamic C Function Reference Manual*.

All of these documents are available on our website, at www.zworld.com.

3.4 HTML Forms

This facility is oriented towards the HTTP server, however it is Zserver that actually handles the form data (as a special resource type in the dynamic resource table only).

Defining `FORM_ERROR_BUF` is required to use the HTML form functionality in `ZSERVER.LIB`. The value assigned to this macro is the number of bytes to reserve in root memory for the buffer used for form processing. This buffer must be large enough to hold the name and value for each variable, plus four bytes for each variable.

An array of type `FormVar` must be declared to hold information about the form variables. Be sure to allocate enough entries in the array to hold all of the variables that will go in the form. If more forms are needed, then more of these arrays can be allocated. Please see Section 4.5.4 on page 176 for an example program.

Starting with Dynamic C 8.50, a more flexible way of supporting form generation is available with the Dynamic C RabbitWeb Module. For more information on this enhanced HTTP server, see the document "RabbitWeb, To Web-Enable Embedded Applications" available at:

www.zworld.com/products/dc/DC8/docs.shtml

3.5 API Functions

The resource manager API functions are described in this section. These functions give servers a consistent interface to files, variables and client information.

sauth_adduser

```
int sauth_adduser( char *username, char *password,  
                  word servermask );
```

DESCRIPTION

This function adds a user to the user table. It fills in the fields of the `ServerAuth` structure associated with this user. Three of the field are specified by the parameters passed into the function. Two other fields, one for the user group mask and the other for the write access mask, are given default values.

The default for the user group mask is the assigned index number (0 to `SAUTH_MAXNAME-1`) as a bit number; that is, $1 \ll \text{index}$. This effectively creates each user in a unique (single) group. Since this does not offer any real control over the assigned group mask, it is recommended to use `sauth_setusermask()` after this to assign the correct access masks.

The default for the write access mask is the user has no write access to any server. To assign this permission, call the function `sauth_setwriteaccess()` with the user table index returned by `sauth_adduser()`.

PARAMETERS

username	Name of the user, a character string up to <code>SAUTH_MAXNAME</code> characters.
password	Password for the user, another character string up to <code>SAUTH_MAXNAME</code> characters.
servermask	Bitmask representing valid servers (e.g., <code>SERVER_HTTP</code> , <code>SERVER_FTP</code>).

RETURN VALUE

-1: Failure.
≥0: Success; index into user table (id passed to `sauth_getusername()`).

LIBRARY

ZSERVER.LIB

SEE ALSO

`sauth_authenticate`, `sauth_getwriteaccess`, `sauth_setusermask`,
`sauth_setwriteaccess`, `sauth_removeuser`

sauth_authenticate

```
int sauth_authenticate( char *username, char *password,
                        word server );
```

DESCRIPTION

Authenticate user and return the user index representing the authenticated user, that is, the user table index. This performs only a plaintext comparison of the userid and password. Servers probably will have their own, more sophisticated, checks.

If `username` is `NULL`, or empty string, then password-only matching is attempted for servers who allow this type of authentication (as defined by the `SERVER_PASSWORD_ONLY` macro).

PARAMETERS

username	Name of user.
password	Password for the user.
server	The server for which this function is authenticating (e.g., <code>SERVER_HTTP</code> , <code>SERVER_FTP</code>).

RETURN VALUE

-1: Failure or user not authorized.
≥0: Success, array index of the `ServerAuth` structure for authenticated user.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sauth_adduser`

sauth_getpassword

```
sauth_getpassword( int userid );
```

DESCRIPTION

Get the password for a user.

PARAMETER

userid user index

RETURN VALUE

!=NULL: password string
NULL: Failure

LIBRARY

ZSERVER.LIB

SEE ALSO

sauth_setpassword

sauth_getserver

```
int sauth_getserver( int sauth );
```

DESCRIPTION

Returns whether or not a user is visible to particular server(s).

PARAMETER

sauth user index

RETURN VALUE

0: This user is visible to all servers
>0: Visible to select servers. One bit is set for each server that knows about this user.
-1: Failure; for example, sauth is an invalid index into the user table.

SEE ALSO

sauth_setserver

sauth_getuserid

```
int sauth_getuserid( char *username, word server );
```

DESCRIPTION

Gets the user index for a user.

PARAMETERS

username	User's name. If this name is not found, then the list is re-scanned looking for an entry with an empty user name ("") and a password that matches <code>username</code> . The second pass is only done for servers that allow password-only matching. Such servers must be specified by defining a symbol <code>SERVER_PASSWORD_ONLY</code> to be a bitmask of such servers.
server	Server(s) for which we are looking up. Use <code>SERVER_ANY</code> if not concerned with the server mask.

RETURN VALUE

≥0: Success, index of user in the user table.
-1: Failure.

LIBRARY

ZSERVER.LIB

sauth_getusermask

```
int sauth_getusermask(int userid, word * groupbits,  
    void ** authdata);
```

DESCRIPTION

Get the group access bit(s) and/or authorization data for a given user ID.

PARAMETERS

userid	User index
groupbits	Pointer to bitmask that will be set to group(s) of which this user is a member. If NULL, this information is not retrieved.
authdata	Pointer to <code>void*</code> that is set to arbitrary server data. If NULL, this information is not retrieved.

RETURN VALUE

0: OK
-1: Failed: `userid` not valid.

sauth_getusername

```
char *sauth_getusername( int userid );
```

DESCRIPTION

Returns the name of the user, a character string from the `ServerAuth` structure associated with `userid`.

PARAMETERS

userid The user's id, that is, the index into the user table.

RETURN VALUE

NULL: Failure.

!NULL: Success, pointer to the user's name string.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_getusername`

`sauth_getwriteaccess`

```
int sauth_getwriteaccess( int sauth );
```

DESCRIPTION

Checks whether or not a user has write access to any server's resources. This is an “in principle” test. Each resource is individually protected from write access: this is not checked. In other words, this function may return TRUE even when none of the resources are writable to this user.

PARAMETERS

`sauth` Index into the user table.

RETURN VALUE

0: User does not have write access.
1: User has write access.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sauth_setwriteaccess`

sauth_removeuser

```
int sauth_removeuser( int userid );
```

DESCRIPTION

Remove the given user from the user list. **IMPORTANT:** Any associations of the given user with web pages should be changed. Otherwise, no one will have access to the unchanged web pages. Authentication can be turned off for a page with `sspec_setrealm(sspec, "")`.

PARAMETERS

userid	Index in user table.
---------------	----------------------

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sauth_adduser

sauth_setpassword

```
int sauth_setpassword( int userid, char *password );
```

DESCRIPTION

Sets the password for a user.

PARAMETERS

userid	Index of user in user table.
password	User's new password.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sauth_getpassword

`sauth_setserver`

```
int sauth_setserver(int sauth, int server);
```

DESCRIPTION

Sets whether a user is visible to the specified server(s).

PARAMETERS

<code>sauth</code>	User index
<code>server</code>	Server bitmask, with bit set to 1 to make this user “known” to the server. If this parameter is zero, then the user is visible to ALL servers, however it is recommended to pass the value <code>SERVER_ANY</code> in this case.

RETURN VALUE

0: Success
-1: Failure

SEE ALSO

`sauth_getserver`

sauth_setusermask

```
int sauth_setusermask(int userid, word usermask, void *  
    authdata);
```

DESCRIPTION

Set the group access bit(s) and authorization data for a given user ID.

PARAMETERS

userid	User index
usermask	Bitmask of group(s) of which this user is a member. This should be non-zero, otherwise the user will not have access to any resources.
authdata	Arbitrary data that can be used by specific servers.

RETURN VALUE

0: OK
-1: Failed: **userid** not valid.

sauth_setwriteaccess

```
int sauth_setwriteaccess( int sauth, int writeaccess );
```

DESCRIPTION

Set whether or not a user has write access with the specified server(s).

PARAMETERS

sauth	Index of the user in the user table.
writeaccess	Server bitmask, with bit set to 1 for write access, 0 for no write access. This is a bitwise OR of the server macros, <code>SERVER_HTTP</code> , etc., that you want the user to have write access to.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sauth_getwriteaccess`

sspec_access

```
int sspec_access( char * name, ServerContext * context );
```

DESCRIPTION

Test access to a given resource by a specified user. The `userid` is set in `context->userid`, or -1 for testing access by the server in general.

NOTE: `sspec_checkpermissions()` performs a similar function, except on a resource handle rather than a resource name.

PARAMETERS

name	Resource name, as a null-terminated string. This name is assumed to be relative to <code>context->cwd</code> if it does not begin with a “/” character. Otherwise, the name is assumed to be relative to <code>context->rootdir</code> .
context	Additional context information. The <code>ServerContext</code> structure is set up by the caller. See <code>sspec_open()</code> for documentation on this structure. For this function, <code>context->userid</code> should be set to the current user whose access is being tested, or may be set to -1 to test access by the server in general.

RETURN VALUE

≥0: Success. The return value is a bitmask of the following values:

- `O_READ` - user+server has read access
- `O_WRITE` - user+server has write access
- 0 (zero) - no access

The following return values are negatives of the values defined in `errno.lib`.

- `-ENOENT` - The resource was not found.
- `-EINVAL` - The resource name was malformed (e.g., too long), or `context` was `NULL`, or the resource was not a file type.

SEE ALSO

`sspec_read`, `sspec_write`, `sspec_seek`, `sspec_tell`, `sspec_close`,
`sspec_checkpermissions`

sspec_addCGI

```
int sspec_addCGI( char* name, void (*fptr)(), word servermask );
```

DESCRIPTION

Add a CGI function to the RAM resource list. This function is currently only useful for the HTTP server, in which case the function is registered as a CGI processor. Make sure that `SSPEC_MAXSPEC` is large enough to hold this new entry.

PARAMETERS

name	URL name of the new function, for example, <code>myCGI.cgi</code>
fptr	Pointer to the function. The prototype for this function is: <pre>int (*fptr)(HttpState * state);</pre> There is a specific documented interface that must be used when specifying this type of CGI handler function. See the manual for details.
servermask	Bitmask representing valid servers (currently only useful with <code>SERVER_HTTP</code>)

RETURN VALUE

≥0: Successfully added spec index
-1: Failed to add function.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_addfsfile`, `sspec_addfunction`, `sspec_addrootfile`,
`sspec_addvariable`, `sspec_addxmemvar`, `sspec_addxmemfile`
`sspec_aliasspec`, `sspec_addform`

sspec_addform

```
int sspec_addform( char *name, FormVar *form, int formsize,
                  word servermask );
```

DESCRIPTION

Adds a form (set of modifiable variables) to the TCP/IP servers' object list. Make sure that SSPEC_MAXSPEC is large enough to hold this new entry. This function is currently only useful for the HTTP server.

PARAMETERS

name	Name of the new form.
form	Pointer to the form array. This is a user-defined array to hold information about form variables.
formsize	Size of the form array
servermask	Bitmask representing valid servers (currently only useful with SERVER_HTTP)

RETURN VALUE

≥0: Success; location of form in server spec list.
-1: Failed to add form.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_addfsfile, sspec_addfunction, sspec_addrootfile,
sspec_addvariable, sspec_addxmemvar, sspec_addxmemfile
sspec_aliasspec, sspec_addfv

sspec_addfsfile

```
int sspec_addfsfile( char *name, byte filenum,  
                    word servermask );
```

DESCRIPTION

Adds a file, located in the FS2 filesystem, to the RAM resource list. Make sure that SSPEC_MAXSPEC is large enough to hold this new entry. This function associates a name with the file.

This creates an alias entry for /fs2/file<n>.

Note that all FS2 files are automatically accessible. There is no need to call this function unless it is desired to assign a name to an FS2 file other than the default, which is file1, file2 etc.

For more information regarding the FS2 filesystem, please see the *Dynamic C User's Manual*.

PARAMETERS

name	Name of the new file.
filenum	Number of the file in the file system (1-255) . This is the number passed in as the second parameter to <code>fcreate()</code> or the return value from <code>fcreate_unused()</code> .
servermask	Bitmask representing servers for which this entry will be valid (e.g., <code>SERVER_HTTP</code> , <code>SERVER_FTP</code>).

RETURN VALUE

-1: Failure.
≥0: Success; location of file in TCP/IP servers' object list.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_addrootfile`, `sspec_addfunction`, `sspec_addvariable`,
`sspec_addxmemfile`, `sspec_addform`, `sspec_aliasspec`

sspec_addfunction

```
int sspec_addfunction( char *name, void (*fptr)(),
    word servermask );
```

DESCRIPTION

Adds a function to the RAM resource list. Make sure that `SSPEC_MAXSPEC` is large enough to hold this new entry. This function is currently only useful for HTTP servers.

NOTE: If using HTTP upload facility and/or the new CGI interface, use `sspec_addCGI()` instead.

PARAMETERS

name	Name of the function.
(*fptr)()	Pointer to the function.
servermask	Bitmask representing servers for which this function will be valid (currently only useful with <code>SERVER_HTTP</code>).

RETURN VALUE

-1: Failure.
≥0: Success, location of the function in the TCP/IP servers' object list.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_addform`, `sspec_addfsfile`, `sspec_addrootfile`,
`sspec_addvariable`, `sspec_addxmemvar`, `sspec_addxmemfile`,
`sspec_aliasspec`

`s_spec_addfv`

```
int s_spec_addfv( int form, int var );
```

DESCRIPTION

Adds a variable to a form.

PARAMETERS

form	spec index of the form (previously returned by <code>s_spec_addform()</code>).
var	spec index of the variable to add (which must have been previously created using <code>s_spec_addvariable()</code>)

RETURN VALUE

-1: Failure.
≥0: Success; next available index into the FormVar array.

LIBRARY

ZSERVER.LIB

SEE ALSO

`s_spec_addform`

sspec_addrootfile

```
int sspec_addrootfile( char *name, char *fileloc, int len, word
    servermask );
```

DESCRIPTION

Adds a file that is located in root memory to the dynamic resource table. Make sure that SSPEC_MAXSPEC is large enough to hold this new entry.

PARAMETERS

name	Name of the new file. This must be unique, but this function does not check. The name should not conflict with the virtual filesystem hierarchy. That is, it should not start with /fs2/, /A/, /B/ etc.
fileloc	Pointer to the beginning of the file.
len	Length of the file in bytes.
servermask	Bitmask representing servers for which this entry will be valid (e.g., SERVER_HTTP, SERVER_FTP).

RETURN VALUE

-1: Failure.
≥0: Success; file index into the resource list.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_addfsfile, sspec_addxmemfile, sspec_addxmemvar,
sspec_addvariable, sspec_addfunction, sspec_addform,
sspec_aliasspec, sspec_resizerootfile

sspec_addrule

```
int sspec_addrule(char * pfx, char * realm, word readgroups,
                  word writegroups, word servermask, word method,
                  MIMETYPEMap * mimetype);
```

DESCRIPTION

Add a rule to the dynamic resource rule table. Resource rules are used to associate access information with resource names matching the specified prefix string. The most specific, that is, the longest, matching string is used.

Normally, the rule table is consulted only for resource names that belong in a file system (FS2 or FAT). You can also cause the rule table to be consulted for flash- or RAM-table entries if you leave the `realm` field as `NULL` in the entry. If the `realm` field is not `NULL`, then the rule table is not consulted for that entry. If the `realm` field was `NULL`, and there was no applicable entry in the rule table, then the resource table permissions are used (with `NULL` realm).

Do not attempt to use a very large number of rule table entries, since the table must be searched exhaustively for each initial resource access. There should be no need for a large number of entries provided that the resource name hierarchy is organized in a reasonably efficient manner. For example, keep the resources for a particular user or realm in one directory, and just add an entry for that directory instead of an entry for each resource. This works because the full path name is always used for matching, and the directory will always be a prefix string of the files that reside in that directory.

As an alternative to this function, you can statically initialize a rule table using the following macros:

```
#define SSPEC_FLASHRULES           // Required.
#use "zserver.lib"                 // this lib

SSPEC_RULETABLE_START
SSPEC_RULE("prefix", realm, rg, wg, sm)
SSPEC_RULE("prefix", realm, rg, wg, sm)
SSPEC_MM_RULE("prefix", realm, rg, wg, sm, meth, mime)
SSPEC_MM_RULE("prefix", realm, rg, wg, sm, meth, mime)
...
SSPEC_RULETABLE_END
```

The `SSPEC_MM_RULE` macro parameters are basically the same parameters as would be passed to this function. These macros define and initialize a constant rule table named `f_rule_table`. `SSPEC_RULE` just omits the (rarely used) `method` and `mimetype` fields.

When using a static rule table, the dynamically added entries are searched before the static ones.

PARAMETER

prefix	Prefix of resource name. This must include the initial “/” character, since all matching is done using absolute path names. If this prefix string exactly matches an existing entry in the table, that entry is replaced. Otherwise, a new entry is created (if possible). This string is not copied, only the pointer is stored. Thus, <code>prefix</code> must point to static storage, that is, a string constant or a global variable. Initial characters other than “/” are reserved for future use.
realm	If not NULL, is an arbitrary null-terminated string that may be used by the server. For HTTP, this is used as the “realm” of the resource. This string is not copied, only the pointer is stored. Thus, the parameter must point to static storage.
readgroups	A word with a bit set for each group that can access this resource for reading. A maximum of 16 different user groups can exist.
writegroups	A word with a bit set for each group that can access this resource for writing. The user must also be given write permission to resources in the <code>userid</code> table entry for the appropriate server(s).
servermask	The server(s) that are allowed to access this resource. Servers have predefined bits. This parameter should be a combination of <ul style="list-style-type: none">• <code>SERVER_HTTP</code>: web server• <code>SERVER_FTP</code>: file transfer protocol server• <code>SERVER_SMTP</code>: email• <code>SERVER_HTTPS</code>: secure web server• <code>SERVER_SNMP</code>: SNMP agent• <code>SERVER_USER</code>: user-defined server• <code>SERVER_ANY</code>: for all servers.
method	Allowable authentication method(s) to be used when accessing this resource. If zero, then the resource has no particular authentication method requirements. This is a bitwise combination of: <ul style="list-style-type: none">• <code>SERVER_AUTH_BASIC</code>: plaintext <code>userid/password</code>• <code>SERVER_AUTH_DIGEST</code>: challenge-response protocol• <code>SERVER_AUTH_PK</code>: public key (such as SSL/TLS)
mimetype	An appropriate MIME type to use. If NULL, then the default table (called <code>http_types</code>) will be consulted.

RETURN VALUE

≥0: OK

-1: Error. For example, out of space in rule table; increase `SSPEC_MAXRULES`.

SEE ALSO

`sspec_remove_rule`, `sspec_getMIMETYPE`

sspec_adduser

```
int sspec_adduser(int sspec, int userid);
```

DESCRIPTION

Add to the read permission mask for the given resource. The groups that `userid` is a member of are ORed into the existing permission mask for the resource. The write permissions are not modified.

NOTE: This is not used to create new userids. For that, see `sauth_adduser()`.

Adds a user to the list of users that have access to the given spec entry. Up to `SSPEC_USERSPERRESOURCE` users can be added. Any more than that will result in this function returning -1.

This function is deprecated as of Dynamic C 8.50. Use the more general `sspec_setpermissions()` function instead.

PARAMETERS

sspec	Spec index.
userid	User index.

RETURN VALUE

≥0: Success, index of `userid` added for given spec entry.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_setuser`, `sspec_getusername`, `sspec_getuserid`,
`sspec_removeuser`, `sspec_setpermissions`

sspec_addvariable

```
int sspec_addvariable( char *name, void *variable, word type,
    char *format, word servermask );
```

DESCRIPTION

Adds a variable to the dynamic resource table (aka, the RAM resource list). Make sure that `SSPEC_MAXSPEC` is large enough to hold this new spec entry. This function is currently only useful for the HTTP server.

PARAMETERS

name	Name of the new variable. This must be unique, but this function does not check. The name should not conflict with the virtual file-system hierarchy. That is, it should not start with <code>/fs2/</code> , <code>/A/</code> , <code>/B/</code> etc. Variables appear in a directory listing of the root directory <code>/</code> however, they cannot be opened using <code>sspec_open()</code> .
variable	Address of actual variable.
type	Variable type, one of: <ul style="list-style-type: none">• INT8 - single character• INT16 - 2-byte integer• PTR16 - string in root memory• INT32 - 4-byte (long) integer• FLOAT32 - floating point variable
format	Output format of the variable as a <code>printf()</code> conversion specifier, e.g., <code>“%d.”</code>
servermask	Bitmask representing servers for which this function will be valid (currently only useful with <code>SERVER_HTTP</code>).

RETURN VALUE

-1: Failure.
≥0: Success, the index of the variable in the resource list.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_addfsfile`, `sspec_addrootfile`, `sspec_addxmemfile`,
`sspec_addxmemvar`, `sspec_addfunction` `sspec_addform`,
`sspec_aliasspec`

sspec_addxmemfile

```
int sspec_addxmemfile( char *name, long fileloc,  
word servermask );
```

DESCRIPTION

Adds a file, located in extended memory, to the RAM resource list. Make sure that `SSPEC_MAXSPEC` is large enough to hold this new entry.

PARAMETERS

name	Name of the new file. This must be unique, but this function does not check. The name should not conflict with the virtual filesystem hierarchy. That is, it should not start with <code>/fs2/</code> , <code>/A/</code> , <code>/B/</code> etc.
fileloc	Location of the beginning of the file. The first 4 bytes of the file must represent the length of the file (<code>#ximport</code> does this automatically).
servermask	Bitmask representing servers for which this entry will be valid (e.g., <code>SERVER_HTTP</code> , <code>SERVER_FTP</code>).

RETURN VALUE

-1: Failure.
≥0: Success, the location of the file in the dynamic resource list.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_addfsfile`, `sspec_addrootfile`, `sspec_addvariable`,
`sspec_addxmemvar`, `sspec_addfunction`, `sspec_addform`,
`sspec_aliasspec`

sspec_addxmemvar

```
int sspec_addxmemvar( char *name, long variable, word type,
    char *format, word servermask );
```

DESCRIPTION

Add a variable located in extended memory to the RAM resource list. Make sure that SSPEC_MAXSPEC is large enough to hold this new entry. Currently, this function is useful only for the HTTP server.

PARAMETERS

name	Name of the new variable. This must be unique, but this function does not check. The name should not conflict with the virtual file-system hierarchy. That is, it should not start with /fs2/, /A/, /B/ etc. Variables appear in directory listing of the root directory "/" however, they cannot be opened using <code>sspec_open()</code> .
variable	Address of the variable in extended memory.
type	Variable type, one of: <ul style="list-style-type: none">• INT8 - single character• INT16 - 2-byte integer• PTR16 - string in root memory• INT32 - 4-byte (long) integer• FLOAT32 - floating point variable
format	Output format of the variable as a <code>printf()</code> conversion specifier e.g., "%d."
servermask	Bitmask representing valid servers (currently only useful with <code>SERVER_HTTP</code>).

RETURN VALUE

-1: Failure.
≥0: Success, the index of the variable in the resource list.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_addfsfile`, `sspec_addrootfile`, `sspec_addvariable`,
`sspec_addfunction`, `sspec_addform`, `sspec_addxmemfile`,
`sspec_aliasspec`

`sspec_aliasspec`

```
int sspec_aliasspec( int sspec, char *name );
```

DESCRIPTION

Creates an alias to an existing `ServerSpec` structure. Make sure that `SSPEC_MAXSPEC` is large enough to hold this new entry.

This is NOT a deep copy. That is, any file, variable, or form that the alias (the new spec entry) references will be the same copy of the file, variable, or form that already exists in the old spec entry. This should be called only when the original entry has been completely set up.

NOTE: do not attempt to alias a `sspec` handle that was returned by `sspec_open()`, because the handle may be dynamically allocated. In such a case, the alias will not work once the original handle is closed. You can test whether such a "virtual" handle has been returned using the macro `SSPEC_IS_VIRT(sspec)`.

PARAMETERS

<code>sspec</code>	sspec index that this function will alias.
<code>name</code>	Alias name.

RETURN VALUE

-1: Failure.
≥0: Success; return location of alias, i.e., new index.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_addform`, `sspec_addfsfile`, `sspec_addfunction`,
`sspec_addrootfile`, `sspec_addvariable`, `sspec_addxmemfile`

sspec_automount

```
int sspec_automount( word which, void ** fatstuff,  
    void ** fs2stuff, void ** reserved );
```

DESCRIPTION

This function automatically initializes and *mounts* the specified filesystem(s) for use by Zserver. Mounting a filesystem creates an entry point to that filesystem for the server.

You must #use the appropriate filesystem library (for example, FS2.LIB) otherwise the filesystem will not be mountable.

If using the FAT library, you must include one or more “driver libraries” (such as `sflash_fat.lib`) before #use `fat.lib`. Only the default device from the first driver library will be initialized and used by this routine. If you need to use any other devices, you will need to initialize them individually and call the `sspec_fatregister()` function. `SSPEC_MAX_FATDRIVES` will also need to be increased from its default value of one.

For the FAT library, this routine calls `fat_init()` and mounts the first available FAT partition on that drive (if any). If the first available partition is the first partition on the drive, then it will be mounted at mount point “/A”. If it is the second partition, it will be mounted at “/B” etc. Up to four partitions are scanned. If none are found (or none are FAT12 or FAT16 partitions) then an error is returned.

For FS2, all logical extents will be initialized via the `fs_init()` function.

PARAMETERS

which The filesystem(s) to mount. This is a bitwise OR of the following constants:

- `SSPEC_MOUNT_FS` - FS or FS2 flash filesystem
- `SSPEC_MOUNT_FAT` - FAT filesystem (1st drive).

You can also pass `SSPEC_MOUNT_ANY` to mount all known filesystems.

fatstuff Either NULL (no info return) or must point to a struct of type `sspec_fatinfo`. This structure (defined in `zserver.lib`) consists of the following fields:

```
typedef struct {
    dos_ctrl * ctrl;
    mbr_drive * drive;
    fat_part * part[4];
} sspec_fatinfo;
```

When calling this function, you should NULL out all these pointers using `memset(..., 0, ...)`. You can then optionally set some of the pointers to valid non-NULL values in order to override the defaults supplied by this function. If you set the `ctrl` pointer, then it is assumed by this function that you have already called the controller initialization function. If the pointer is NULL on entry, then this function will call the default controller initialization via the `DOS_CONTROLLER_INIT` macro.

On return, pointers that were NULL on entry may be changed to point to valid default information. In particular, the `ctrl` and `drive` fields will point to defaults. One (and only one) of the NULL `part` pointers may be set to a default partition structure if a default partition could be located on the drive.

If `fat.lib` is not included, the above structure is still defined, but contains 6 `void` pointers. This is just to avoid compilation problems, since no information will be used or returned.

fs2stuff This parameter is currently reserved for returning FS2 information. For now, pass as NULL.

reserved Reserved for other filesystems. For now, pass as NULL.

RETURN VALUE

0: OK

Otherwise, if a filesystem fails to mount, the return code is the bitwise OR of the `SSPEC_MOUNT_*` constants of those filesystem(s) that failed to initialize.

SEE ALSO

`sspec_fatregister`, `sspec_fatregistered`

sspec_cd

```
int sspec_cd(char * path, ServerContext * context, int check);
```

DESCRIPTION

Change the current working directory in the `ServerContext` structure. This function may be used by servers that support the concept of a current directory, such as FTP (but not HTTP). Standard Unix-like path names are used, including support for “.” and “..” directory components.

The resulting directory name is not allowed to be closer to the root directory than `context->rootdir`. If there is any specification error, then the current directory is not changed. The resulting absolute directory name cannot be longer than `SSPEC_MAXNAME`, including a leading and trailing “/” character.

PARAMETERS

path	New directory path string, as a null-terminated string. If this starts with “/” it is merely appended to the <code>context->rootdir</code> string. Otherwise, it is appended to the current directory (in <code>context->cwd</code>). Directory components are separated by “/” characters. A trailing slash is optional. A directory component “.” means “no change to this level,” and a component of “..” means “up one level” (towards the root).
context	Server context structure. Two fields in this are of interest: <code>rootdir</code> points to a virtual root directory for this server. For example, if the FTP server is only allowed to access files under the <code>/A/</code> directory, then <code>rootdir</code> points to a string “/A/”. If the user entered a directory name of “/ftpfiles/” the full directory would be “/A/ftpfiles/” The other field that is updated by this function, is <code>cwd</code> . This is an array of characters of length <code>SSPEC_MAXNAME</code> . It contains the absolute path of the current directory, with leading and trailing slash, including the <code>rootdir</code> part (if any).
check	If TRUE, check the resulting directory name to see if it exists. Otherwise, no check is made.

RETURN VALUE

0: OK.

Any other negative values indicate an error:

- E2BIG: Resulting directory name too long
- EACCES: Attempt to change above root directory
- ENOENT: 3rd parameter was TRUE, and the directory did not exist.

SEE ALSO

`sspec_pwd`

sspec_checkaccess

```
int sspec_checkaccess( int sspec, int userid );
```

DESCRIPTION

This function checks whether or not the specified user has permission to access the specified resource in the resource table. Only read access is checked.

This function is deprecated as of Dynamic C 8.50. Use the function `sspec_checkpermissions()` instead.

PARAMETERS

sspec	spec index
userid	user index

RETURN VALUE

0: User does not have access.
1: User has access.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_needsauthentication`, `sspec_checkpermissions`

`s`**spec**`_`**checkpermissions**

```
int sspec_checkpermissions(int sspec, ServerContext * context);
```

DESCRIPTION

Returns the access permissions for the given server and user, for the given resource.

`sspec_access()` performs the same function, except that a resource name can be given (rather than an open resource handle).

PARAMETERS

sspec spec index

context Server context. The relevant fields are:

`context->server`: the current server (`SERVER_HTTP` etc.)

`context->userid`: current user, or `-1` for testing the server in general.

When testing the server in general, both `O_READ` and `O_WRITE` will be returned.

RETURN VALUE

≥ 0 : Bitwise combination of:

- `O_READ`: resource is readable
- `O_WRITE`: resource has write permission. This does NOT necessarily mean that the resource can actually be written, only that the permission bits allow it.

< 0 : Error. For example, no permissions can be located or the `sspec` handle is invalid.

SEE ALSO

`sspec_setpermissions`, `sspec_getpermissions`, `sspec_access`

sspec_close

```
int sspec_close( int sspec );
```

DESCRIPTION

Close a file resource. This function must be called by servers when they have completed operations on the file, otherwise there will be a resource leak and future open calls will fail.

PARAMETER

sspec	Open file handle. This must be a handle that was returned by <code>sspec_open()</code> .
--------------	--

RETURN VALUE

≥0: Success.

The following return values are negatives of the values defined in `errno.lib`.

- -EBADF: The specified handle was not open or invalid.
- Other negative values indicate an error in closing the file resource.

SEE ALSO:

`sspec_read`, `sspec_write`, `sspec_seek`, `sspec_tell`, `sspec_open`

`sspec_delete`

```
int sspec_delete(char * name, ServerContext * context);
```

DESCRIPTION

Delete a resource by name. See `sspec_open()` for a detailed description of parameters.

PARAMETERS

<code>name</code>	Name of resource.
<code>context</code>	Current server context.

RETURN VALUE

0: OK.

The following return value is a negative of the values defined in `errno.lib`. Any other negative values indicate an error.

- `-ENOENT`: The specified resource did not exist.

SEE ALSO

`sspec_mkdir`, `sspec_rmdir`, `sspec_open`

sspec_dirlist

```
int sspec_dirlist( int item, char *line, int linelen,  
                  ServerContext * context, word options );
```

DESCRIPTION

Return formatted directory listing line. To use this function, call it with `item = 0` the first time, then keep calling it with `item = <previous return value>` until it returns negative. This allows you to iterate through all entries in a directory.

The `ServerContext` structure contains the current user ID, server, and the name of the directory to list.

Note: For a given directory, you should call this function with `item = 0`, followed by more calls until it returns -1. If you want to terminate the directory listing without iterating through every entry, pass the `SSPEC_LIST_END` option flag (see below). This allows Zserver to release any temporary resources acquired for the purpose of iterating through the directory. This is especially important for FAT filesystem listings. After this function returns negative, you must start the next directory listing from the top, that is, `item = 0`.

If you do not complete the listing, then your application may not be able to perform further listings owing to internal resource leakage. This is similar to the need to close file resources that are opened. See the second example below.

Pass the same `ServerContext` structure for the entire directory list sequence, since Zserver keeps track of state information in this structure.

EXAMPLE

To iterate through all resources under `"/A/`":

```
ServerContext ctx;  
int item;  
char buf[80];  
word opts;  
word n;  
  
ctx.rootdir = "/";  
ctx.server = SERVER_FTP;  
ctx.userid = sauth_getuserid("foo", SERVER_FTP);  
sspec_cd("/A", &ctx);  
for (item = 0; item >= 0; ) {  
    item = sspec_dirlist(item, buf, sizeof(buf), &ctx,  
                        SSPEC_LIST_LONG);  
    if (item >= 0)  
        printf(buf);  
} // finished now, can re-use ctx.
```

To iterate through the first 5 resources only:

```
opts = SSPEC_LIST_LONG;
for (item = 0, n = 0; item >= 0; ++n) {
    if (n >= 4)
        opts |= SSPEC_LIST_END;
    item = sspec_dirlist(item, buf, sizeof(buf), &ctx, opts);
    if (item >= 0)
        printf(buf);
}
```

PARAMETERS

item	Directory entry to list. If zero, this always returns the first entry in the directory. Thereafter, pass the return value from the previous call to this function to get the next item(s). NOTE: the return value does not necessarily count up 1, 2, 3 etc. Apart from 0, the only values you should pass in this parameter are previous return values, otherwise the results will be undefined.
line	Points to buffer that is filled with resulting string. The string will be terminated with <code>\r\n</code> (CRLF) then a NULL.
linelen	Length of the above buffer. If it is not long enough, then the line will be truncated (however it will still have the terminating CRLF + null). The minimum reasonable value is about 15 for format 0, and 80 for format 1.
context	Server context. This structure will have the following fields initialized: userid : current user who is doing the listing, or -1 if no specific user. server : mask bit of the server who is performing the listing. cwd [] : set to the directory to list. The <code>sspec_cd()</code> function can be used to set this field correctly. This struct must be the same instance for all calls in a single directory listing sequence.

options

Listing options. This is a bit field that should have a combination of the following flags:

- `SSPEC_LIST_LONG`: Long format listing (else just names)
- `SSPEC_LIST_END`: Close the current directory listing.

For the long format, the template is:

```
<permissions> 1 <user> <group> <length> <date> <name>
```

Where

- `permissions` is a string of 10 characters in 3 sets of 3, plus one. Each set of 3 indicates read, write or execute permissions for the user, group, and “world” respectively. The 1st char is “d” if the entry is a directory, or “-” otherwise. Since Zserver does not really support file owners or groups, or execute permissions, the 3 sets will be either “rw-” or “r--” or sometimes “-w-”. The user bits are set according to the current user's access. The group bits are set if any other user has access, and the “world” bits are set if any other server has access.
- “1” is a constant for Unix compatibility.
- `user` is the username who “owns” the file resource. Since Zserver does not have the concept of resource ownership, this is set to the user ID of the `context->userid` field. If `context->userid` is -1, this is set to `anon`.
- `group` is the resource “group name.” Zserver does not support this Unix concept either, so this field is set to the realm of the file resource (if it has one) otherwise it is set to `anon`.
- `length` is set to the current length of the file resource, or 0 if not known.
- `date` is set to the modification date of the file resource in `Mon dd yyyy` format.
- `name` is the name of the file resource in this directory.

Example:

```
dr--r--r-- 1 foo admin 0 Jan 1 1980 ftpfiles
-rw-rw-rw- 1 foo admin 1250 Mar 6 2003 index.htm
```

RETURN VALUE

-EEOF: there were no (more) entries in this directory.

Any other negative value: parameter or I/O error.

Otherwise (non-negative): the return value should be passed back to this function as the `item` parameter value, to retrieve the next entry.

SEE ALSO

`ssec_cd`

sspec_fatregister

```
int sspec_fatregister( int partno, fat_part * pt );
```

DESCRIPTION

This function must be used to register all FAT partitions that will be accessible to `Zserver.lib`. Partitions are numbered consecutively from 0, and they correspond to mount points `/A`, `/B`, `/C` etc.

It is assumed that by the time this function is called the required drives and partitions have been mounted. For example, call `fat_EnumDrive()` followed by as many `fat_MountPartition()` calls as required. The `fat_part` pointer returned by `fat_MountPartition()` should be passed to this function. Up to `SSPEC_MAX_PARTITIONS` can be registered. This number can be changed indirectly by defining `SSPEC_MAX_FATDRIVES` before `#use zserver.lib`. This defaults to one drive, and the number of partitions is set to 4 times this number (hence the default allows up to four partitions).

NOTE: It is NOT necessary to call this function if you called `sspec_automount(SSPEC_MOUNT_FAT, . . .)` since that function does all the necessary initializations for a single “drive.”

PARAMETERS

partno	Partition number to register. This starts at 0, corresponding to the “/A” mount point; 1 for “/B” etc.
pt	Pointer to <code>fat_part</code> data structure returned by <code>fat_MountPartition</code> etc. To unregister a partition, pass NULL for this parameter. Note: attempted access to an unregistered partition generally results in an error code of <code>-ENXIO</code> .

RETURN VALUE

≥ 0 : Success.
`-ENXIO`: `partno` outside the allowable range of `0 .. SSPEC_MAX_PARTITIONS-1`.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`fat_EnumDrive`, `fat_EnumPartition`, `fat_MountPartition`,
`sspec_automount`, `sspec_fatregistered`

sspec_fatregistered

```
fat_part * sspec_fatregistered( int partno );
```

DESCRIPTION

Test whether a FAT partition has been registered with Zserver.

PARAMETER

partno	Partition number to test. This starts at 0, corresponding to the “/A” mount point; 1 for “/B”etc.
---------------	---

RETURN VALUE

NULL: Not registered.

Otherwise: Registered, and this is the `fat_part` pointer.

SEE ALSO

`fat_EnumDrive`, `fat_EnumPartition`, `fat_MountPartition`,
`sspec_automount`, `sspec_fatregister`

`sspec_findfv`

```
int sspec_findfv( int form, char *varname );
```

DESCRIPTION

Finds the index of a form variable in a given form.

PARAMETERS

form	spec index of the form in which to search.
varname	Name of the variable to find.

RETURN VALUE

-1: Failure.

≥0: Success; the index of the form variable in the array of type `FormVar`.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_addfv`

sspec_findname

```
int sspec_findname( char *name, word server );
```

DESCRIPTION

Find the spec entry with a name field that matches the given name and is allowed with the specified server(s). Note that a leading slash in the given name and/or in the resource name is ignored for backwards compatibility.

PARAMETERS

name	Name to search for in the resource list.
server	The server making the request (e.g., SERVER_HTTP).

RETURN VALUE

-1: Failure.
≥0: Success, spec index. The special value SSPEC_VIRTUAL is returned if the name refers to part of the virtual filesystem hierarchy. In this case, the server mask is not consulted. SSPEC_VIRTUAL is *not* a valid handle for other functions.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_findnextfile

sspec_findfsname

```
int sspec_findfsname( byte filename, word server );
```

DESCRIPTION

Find the server spec entry for `filename`. The entry must be of type `SSPEC_FSFILE` and be allowed with the specified server.

PARAMETERS

filename	File to search for. This value is the number passed in as the 2nd parm to <code>fcreate()</code> or the return value from <code>fcreate_unused()</code> .
server	The server making the request (e.g., <code>SERVER_HTTP</code>).

RETURN VALUE

-1: Failure.
≥0: Success, index into resource list.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_findname`

sspec_findnextfile

```
int sspec_findnextfile( int start, word servermask );
```

DESCRIPTION

Find the first spec file entry at or following the `start` spec that is accessible by the given server. When the end of the RAM entries is reached, the flash entries are searched. Virtual filesystem entries are not considered. Only entries for which `sspec_gettype()` would return `SSPEC_FILE` are considered.

If you are using this function to iterate through the available resources, then the caller is responsible for incrementing the starting point. To do this, you can call the function `sspec_nexthandle()` which will return the next valid handle after the given one (or -1 if no more handles).

PARAMETERS

start	The array index at which to begin the search. -1 starts searching the RAM entries.
servermask	The server making the request (e.g., <code>SERVER_HTTP</code>).

RETURN VALUE

-1: Failure.
≥0: Success, index of requested `ServerSpec` structure.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_findname`, `sspec_gettype`

sspec_getfileloc

```
long sspec_getfileloc( int sspec );
```

DESCRIPTION

Gets the location in memory or in the file system of a file represented by a `ServerSpec` structure. The location of the file is returned as a `long`, even if the file location should be represented by a `char*` (for a root file) or a `FileNum` (for the file-system). The return value should be cast to the appropriate type by the user.

`sspec_getfiletype()` can be used to find the file type.

PARAMETERS

sspec spec index of the file in the resource list

RETURN VALUE

≥0: Success, location of the file.

-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_getfiletype`, `sspec_getlength`

sspec_getfiletype

```
word sspec_getfiletype( int sspec );
```

DESCRIPTION

Get the type of a file represented by the given spec index.

PARAMETERS

sspec spec index of the file in the resource list, that is, the index into the array of ServerSpec structures.

RETURN VALUE

SSPEC_ROOTFILE: root memory data
SSPEC_XMEMFILE: xmem data
SSPEC_ZMEMFILE: compressed xmem data
SSPEC_FSFILE: FS2 file
SSPEC_ERROR: failure - not a file, or invalid handle

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getfileloc, sspec_gettype

sspec_getformtitle

```
char *sspec_getformtitle( int form );
```

DESCRIPTION

Gets the title for an automatically generated form.

PARAMETERS

form server_spec index of the form.

RETURN VALUE

NULL: Failure.
!NULL: Success, title string.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setformtitle

sspec_getfunction

```
void *sspec_getfunction( int sspec );
```

DESCRIPTION

Returns a pointer to the function represented by the sspec index. The entry must have been created as a SSPEC_FUNCTION or as a SSPEC_CGI.

PARAMETERS

sspec spec index

RETURN VALUE

NULL: Failure.

!NULL: Success, pointer to requested function.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_addfunction

`sspec_getfvdesc`

```
char *sspec_getfvdesc( int form, int var );
```

DESCRIPTION

Gets the description of a variable that is displayed in the HTML form table.

PARAMETERS

form	spec index of the form.
var	Index (into the <code>FormVar</code> array) of the variable.

RETURN VALUE

NULL: Failure.
!NULL: Success, description string.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_setfvdesc`

sspec_getfventrytype

```
int sspec_getfventrytype( int form, int var );
```

DESCRIPTION

Gets the type of form entry element that should be used for the given variable.

PARAMETERS

form	spec index of the form.
var	Index (into the FormVar array) of the variable.

RETURN VALUE

-1: Failure;

Type of form entry element on success:

HTML_FORM_TEXT is a text box.

HTML_FORM_PULLDOWN is a pull-down menu.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setfventrytype

sspec_getfvlen

```
int sspec_getfvlen( int form, int var );
```

DESCRIPTION

Gets the length of a form variable (the maximum length of the string representation of the variable).

PARAMETERS

form	spec index of the form.
var	Index (into the <code>FormVar</code> array) of the variable.

RETURN VALUE

-1: Failure.
≥0: Success, length of the variable.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setfvlen

sspec_getfvname

```
char *sspec_getfvname( int form, int var );
```

DESCRIPTION

Gets the name of a variable that is displayed in the HTML form table.

PARAMETERS

form	spec index of the form.
var	Index into the array of FormVar structures of the variable.

RETURN VALUE

NULL: Failure.
!NULL, name of the form variable.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setfvname

`s_spec_getfvnum`

```
int s_spec_getfvnum( int form );
```

DESCRIPTION

Gets the number of variables in a form.

PARAMETERS

form spec index of the form.

RETURN VALUE

- 1: Failure.
≥ 0: Success, number of form variables.

LIBRARY

ZSERVER.LIB

`s`spec_getfvopt

```
char *s
```

spec_getfvopt(int form, int var, int option);

DESCRIPTION

Gets the numbered option (starting from 0) of the form variable. This function is only valid if the form variable has the option list set.

PARAMETERS

form	spec index of the form.
var	Index into the array of FormVar structures of the variable.
option	Index of the form variable option.

RETURN VALUE

NULL: Failure.
!NULL: Success, form variable option.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setfvoptlist, sspec_getfvoptlistlen

sspec_getfvoptlistlen

```
int sspec_getfvoptlistlen( int form, int var );
```

DESCRIPTION

Gets the length of the options list of the form variable. This function is only valid if the form variable has the option list set.

PARAMETERS

form	spec index of the form.
var	Index (into the <code>FormVar</code> array) of the variable.

RETURN VALUE

-1: Failure.
>0: Success, length of the options list.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getfvopt, sspec_setfvoptlist

sspec_getfvreadonly

```
int sspec_getfvreadonly( int form, int var );
```

DESCRIPTION

Checks if a form variable is read-only.

PARAMETERS

form	spec index of the form.
var	Index (into the FormVar array) of the variable.

RETURN VALUE

0: Not read-only.
1: Read-only.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setfvreadonly

sspec_getfvspec

```
int sspec_getfvspec( int form, int var );
```

DESCRIPTION

Gets the `server_spec` index of a variable in a form.

PARAMETERS

form	server_spec index of the form.
var	Index into the array of <code>FormVar</code> structures of the variable.

RETURN VALUE

-1: Failure.
≥0: Success, index of the form variable in the resource list.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_addfv`

`sspec_getlength`

```
long sspec_getlength( int sspec );
```

DESCRIPTION

Gets the length of the file associated with the specified `ServerSpec` structure. Get the length of the file specified by the `sspec` index. Note that compressed files (`#zimport`) return -1 because their expanded length is not known until they are processed.

PARAMETERS

`sspec` spec index of file in resource list

RETURN VALUE

-1: Failure (compressed file, or other type whose effective length is not known).
≥0: Success, length of the file in bytes.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_readfile`, `sspec_getfileloc`

sspec_getMIMEtype

```
MIMETypeMap * sspec_getMIMEtype(char* name,  
    ServerContext * context);
```

DESCRIPTION

Return the MIME type information for a specified resource name, in the given server context.

Note that the available MIME types are set up by defining a global variable (or constant) table using the definition (for example),

```
const MIMETypeMap http_types [] =  
{  
    { ".html", "text/html", NULL },  
    { ".gif", "image/gif", NULL }  
};
```

The name `http_types` is required for backward compatibility even though servers other than HTTP can make use of MIME types.

When searching for the appropriate type, the rule table is consulted first. Only if this results in a NULL MIME type pointer is the `http_types` table consulted.

See `sspec_open()` for a detailed description of the parameters.

PARAMETER

name	Name of the resource.
context	Current server context.

RETURN VALUE

Pointer to the appropriate table entry. `MIMETypeMap` is defined as:

```
typedef struct {  
    char extension[10];           // File extension or suffix.  
    char type[SSPEC_MAXNAME];    // MIME type e.g., "text/html"  
    int (*fptr)();              // Server-specific processing, e.g., SSI.  
} MIMETypeMap;
```

A valid pointer is always returned. If the appropriate table entry cannot be located by the resource's extension (or using a rule (see `sspec_addrule`)) then the first table entry is returned.

SEE ALSO

`sspec_addrule`

sspec_getname

```
char *sspec_getname( int sspec );
```

DESCRIPTION

Returns the name of the spec entry represented by the sspec index

This only works for RAM and flash table entries.

PARAMETERS

sspec spec index

RETURN VALUE

NULL: Failure.

!NULL: Success, pointer to name string.

LIBRARY

ZSERVER.LIB

sspec_getpermissions

```
int sspec_getpermissions(int sspec, char ** realm,
    word * readgroups, word * writegroups, word * servermask,
    word * method, MIMETypeMap ** mimetype);
```

DESCRIPTION

Get the permission (access control) attributes of a resource.

Except for `sspec`, all parameters are pointers to variables that will be set to the appropriate return value. If the parameter is `NULL`, then that information is not retrieved.

NOTE: The data at `**realm` and `**mimetype` should not be altered by the caller. The data is read-only.

PARAMETERS

<code>sspec</code>	spec index
<code>realm</code>	Pointer to pointer to realm string
<code>readgroups</code>	Pointer to mask of user groups who have read access
<code>writegroups</code>	Pointer to mask of user groups who have write access
<code>servermask</code>	Pointer to servers allowed to access this resource.
<code>method</code>	Pointer to required authentication method.
<code>mimetype</code>	Pointer to pointer to MIME table entry.

RETURN VALUE

0: Success.
<0: Failure. For example, an invalid `sspec` handle

SEE ALSO

`sspec_setpermissions`, `sspec_checkpermissions`, `sspec_access`

sspec_getpreformfunction

```
void *sspec_getpreformfunction( int form );
```

DESCRIPTION

Gets the user function that will be called just before HTML form generation. This function is useful mainly for custom form generation functions.

PARAMETERS

form spec index of the form

RETURN VALUE

NULL: No user function.
!NULL: Pointer to user function.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setpreformfunction, sspec_setformfunction

s_spec_getrealm

```
char *s_spec_getrealm( int s_spec );
```

DESCRIPTION

Returns the realm of the spec entry represented by `s_spec`.

PARAMETERS

s_spec spec index

RETURN VALUE

NULL: Failure.
!NULL: Success, pointer to the realm string.

LIBRARY

ZSERVER.LIB

SEE ALSO

`s_spec_setrealm`

sspec_getservermask

```
int sspec_getservermask(int sspec, word *servermask);
```

DESCRIPTION

Gets the server mask for the given spec entry. This is the bitmask passed in when the entry is created with the `sspec_add*()` functions.

This function only works for RAM and flash table entries.

PARAMETERS

sspec	spec index of the variable
servermask	Address in which the servermask will be returned

RETURN VALUE

0: Success
-1: Failure

sspec_gettype

```
word sspec_gettype( int sspec );
```

DESCRIPTION

Returns the type (SSPEC_FILE, SSPEC_VARIABLE, etc.) of the spec entry represented by `sspec`. This is a generic type, in that, SSPEC_FILE is returned for any type (SSPEC_ROOTFILE, SSPEC_FSFILE etc.) that has file properties and SSPEC_VARIABLE is returned for SSPEC_ROOTVAR or SSPEC_XMEMVAR. Other types are returned without translation.

PARAMETERS

sspec spec index

RETURN VALUE

SSPEC_ERROR: Failure.
!SSPEC_ERROR: Success, type as described above.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getfiletype, sspec_getvartype

`sspec_getuserid`

```
int sspec_getuserid( int sspec, int index );
```

DESCRIPTION

Returns a userid for the given `sspec` resource. Since a resource can have multiple users associated with it, `index` indicates which userid should be returned. Note that `index` should follow the relation $0 \leq \text{index} < \text{SSPEC_USERSPERRESOURCE}$.

If there is no userid for a given `index`, -1 will be returned. If -1 is returned for an `index`, then -1 will also be returned for all higher indices.

This function may be used to iterate through all users that have read access to a particular resource.

This only works for RAM and flash table entries.

Starting with Dynamic C 8.50, access control is done by user groups rather than individual users; therefore, `sspec_getuserid()` may not work as expected.

PARAMETERS

<code>sspec</code>	spec index
<code>index</code>	index of userid for this <code>sspec</code> resource to return: 0, 1, 2 ...

RETURN VALUE

- 1: Error, or no such userid.
≥ 0: Success, userid is returned.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_getusername`, `sauth_getusername`

sspec_getusername

```
char *sspec_getusername( int sspec );
```

DESCRIPTION

Gets the username field of the first user in the user table that has read access to the resource indexed by `sspec`. If multiple users are associated with this resource, the first user's username will be returned. See `sspec_getuserid()` to get all userids for a resource, and `sauth_getusername()` to convert the userids to usernames.

Starting with Dynamic C 8.50, access control is done by groups rather than individual users, therefore, `sspec_getusername()` may not work as expected.

This only works for RAM and flash table entries.

PARAMETERS

sspec	spec index
--------------	------------

RETURN VALUE

NULL: Failure, or no user has read access to this resource.
!=NULL: Success, pointer to username.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sauth_adduser`, `sspec_setuser`, `sauth_getuserid`,
`sauth_getusername`

sspec_getvaraddr

```
void *sspec_getvaraddr( int sspec );
```

DESCRIPTION

Returns a pointer to the requested variable in the resource list.

PARAMETERS

sspec spec index

RETURN VALUE

NULL: Failure.

!NULL: Success, pointer to variable.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_readvariable

sspec_getvarkind

```
word sspec_getvarkind( int sspec );
```

DESCRIPTION

Returns the kind of variable represented by `sspec`.

PARAMETERS

sspec spec index

RETURN VALUE

0: Failure.

On success, returns one of:

- INT8 - single character
- INT16 - 2-byte integer
- PTR16 - string in root memory
- INT32 - 4-byte (long) integer
- FLOAT32 - floating point variable

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_getvaraddr`, `sspec_getvartype`, `sspec_gettype`

sspec_getvartype

```
word sspec_getvartype( int sspec );
```

DESCRIPTION

Gets the type of variable represented by the spec index.

PARAMETERS

sspec spec index.

RETURN VALUE

SSPEC_ERROR: Failure.

SSPEC_ROOTVAR or SSPEC_XMEMVAR: Success.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getvaraddr, sspec_getvarkind, sspec_gettype

sspec_getxvaraddr

```
long sspec_getxvaraddr( int sspec );
```

DESCRIPTION

Returns a pointer to the variable in xmem represented by the sspec index.

PARAMETER

sspec spec index

RETURN VALUE

≥ 0: Variable pointer.

-1: Failure.

SEE ALSO

sspec_readvariable

s_{spec}_mkdir

```
int sspec_mkdir( char * name, ServerContext * context );
```

DESCRIPTION

Create a named directory in the FAT filesystem.

PARAMETERS

name	Name of new directory.
context	Current server context.

RETURN VALUE

0: OK.
- EPERM: Not a filesystem that supports creation of new directories.
- EACCES: Not authorized
Any other negative values indicate an error.

SEE ALSO

s_{spec}_delete, s_{spec}_rmdir, s_{spec}_open

`sspec_needsauthentication`

```
int sspec_needsauthentication( int sspec );
```

DESCRIPTION

Checks if the item represented by the spec entry needs authentication for access. This is defined by having a non-NULL “realm” string for the resource.

This function is deprecated starting with Dynamic C 8.50 in favor of `sspec_checkpermissions()`. It is retained for cases where the permissions structure for a resource contains an authentication method of `SERVER_AUTH_DEFAULT`.

PARAMETERS

sspec spec index

RETURN VALUE

0: Does NOT need authentication.
1: Does need authentication.
-1: Failure, no permissions struct assigned or invalid sspec handle.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_getrealm`, `sspec_checkpermissions`

sspec_open

```
int sspec_open(char * name, ServerContext * context, word mode);
```

DESCRIPTION

Open a file resource by name. The name may refer to a flash- or RAM-spec entry, or may be the name of a file in a filesystem.

The resource namespace is specified as a directory hierarchy, similar to a Unix-like filesystem. The root directory, “/”, is the base for all named resources.

If `fs2.lib` is included, then files stored in the FS2 filesystem are accessible under a mount point called “/fs2.” FS2 files do not have native names. Instead, each file is numbered from 1 to 255. Zserver assigns names to FS2 files by appending the file number (in decimal) to the string “file.” For example, FS2 file number 99 has a complete resource name of “/fs2/file99.”

If `fat.lib` is included, then all DOS FAT files are mounted under a drive letter. The first partition of the first DOS FAT filesystem is called “/A” and the second partition (if any) is called “/B” etc. For example, if the FAT filesystem has a file called “/system/admin.htm” then the complete resource name will be “/A/system/admin.htm”.

NOTE: Forward slashes are required. Do not use backslashes as is customary with DOS filesystems.

If the resource name does not begin with “/fs2” or “/A” etc., then the resource is located in the static resource table (“flashspec” that is, the `http_flashspec` global table) or in the dynamic (RAM) table.

To access the file resource, the return value from this function must be passed to other functions, such as `sspec_read()`. A few functions do not work with resources opened with this function. These cases are documented with the function.

NOTE: When the application has finished accessing the resource, it must be closed using `sspec_close()`. This must be done because there is a limited amount of storage for maintaining the necessary file handles.

PARAMETERS

name	Resource name, as a NULL terminated string. This name is assumed to be relative to <code>context->cwd</code> if it does not begin with a “/” character. Otherwise, the name is assumed to be relative to <code>context->rootdir</code> . Note that the name string can contain “.” and “..” directory components. These will be interpreted as “same directory” and “one level up” as is customary. If “..” components are included, the resulting name cannot be above or outside the root directory specified in <code>context->rootdir</code> .
-------------	---

context

Additional context information. The `ServerContext` structure is set up by the caller. It has the following fields:

```
typedef struct {
    int userid;           // User ID of the current user, or
                        // -1 if not applicable.
    word server;         // Server id (e.g. SERVER_HTTP)
    char * rootdir;     // Root directory. Usually "/"
                        // if the whole namespace is to
                        // be accessible. Otherwise, may
                        // be e.g., "/A" to restrict access to
                        // just first DOS FAT partition.
                        // First and last char must be "/".
    char cwd[];         // Current working directory.
                        // Normally includes rootdir as
                        // a prefix. First and last char
                        // must be "/".
    char * dfltname;    // A file name to be used as a
                        // resource name suffix in the case
                        // that the first parameter refers
                        // to a directory name.
} ServerContext;
```

mode

Resource opening mode. Bitwise OR of the following macros:

- `O_READ`: open for reading
- `O_WRITE`: open for writing (implies reading as well)
- `O_CREAT`: with `O_WRITE`, if file does not exist then create it with zero length and allocation.
- `O_TRUNC`: with `O_WRITE`, if file already exists, truncate it to zero length.
- `O_APPEND`: with `O_WRITE`, if file already exists, position at end of file so as to append new data. You can later seek to the existing portion of the file.

RETURN VALUE

≥ 0 : Success. The returned value should be passed to other functions that require a general handle, such as `sspec_read()`, `sspec_seek()`, `sspec_write()`, `sspec_tell()`, and `sspec_close()`.

The following return values are negatives of the values defined in `errno.lib`.

- `-ENOENT`: The resource was not found when it was expected to exist.
- `-EACCES`: The `context->userid` field was not -1, and the specified user is not allowed to access the resource using the specified mode.
- `-EINVAL`: The resource name was malformed (e.g., too long), or `context` was `NULL`, or the resource was not a file type, or `O_CREAT`, `O_TRUNC` or `O_APPEND` were specified without `O_WRITE`.
- `-ENOMEM`: Insufficient storage for handle or buffers. Increase definition of `SSPEC_MAX_OPEN`.
- `-EPERM`: Operation not permitted, for example., opening an `xmem` file for writing.

SEE ALSO

`sspec_read`, `sspec_write`, `sspec_seek`, `sspec_tell`, `sspec_close`

sspec_pwd

```
char * sspec_pwd( ServerContext * context, char * buf );
```

DESCRIPTION

Print the current working directory in the `ServerContext` structure to the specified buffer. The `context->cwd` field contains the CWD. This function removes the root directory component (`context->rootdir`) and copies the result. This makes `rootdir` invisible to the end user.

The leading slash is included, but the trailing slash is omitted from the result (unless the result is just “/”).

For example, if

```
context->rootdir points to “/A/” and  
context->cwd[] contains “/A/ftpfiles/”  
“/ftpfiles” will be the result returned in buf.
```

PARAMETERS

context	Server context structure. Two fields in this are of interest: <code>rootdir</code> points to a virtual root directory for this server, and <code>cwd</code> is a character array containing the CWD.
buf	Points to buffer that is filled with resulting string. This buffer is assumed to be dimensioned at least <code>SSPEC_MAXNAME</code> chars long, and it will be null terminated on return.

RETURN VALUE

The `buf` parameter is returned.

SEE ALSO

`sspec_cd`

sspec_read

```
int sspec_read( int sspec, char * buf, int len );
```

DESCRIPTION

Read the next byte(s) from the given file resource.

PARAMETERS

sspec	Open file handle. This must be a handle that was returned by <code>sspec_open()</code> .
buf	Buffer into which data is copied.
len	Length of the above buffer. If <code>len</code> is zero, then the return value will be the minimum number of characters that could be read at the current position, which is usually at least 1 except at EOF (0). Thus, this function can be used to test for end-of-file (EOF), that is, if <pre>(sspec_read(sspec, NULL, 0) == 0)</pre> is TRUE, then EOF has been reached in the file identified by <code>sspec</code> .

RETURN VALUE

0: No data is currently available. If the `len` parameter was zero, then a return value of zero definitely means end-of-file has been reached. If `len > 0`, there may be data available in the future, e.g., because the underlying filesystem is socket-based and this host has read all available data, but the socket is still open to receive more data.

1 . . `len`: the specified number of characters has been copied to the supplied buffer, and the current file position has been advanced by that many bytes. Possibly less than `len` bytes may be read, in which case the server should test for EOF.

>`len`: no data was copied, because the underlying filesystem is unable to return a partial record and maintain its current position. The return value is the minimum sized buffer that should be passed on the next call. Note: this sort of return is not currently implemented by any of the file systems, however servers should be coded to handle this case for future anticipated systems which have record-level access rather than byte-level.

The following return values are negatives of the values defined in `errno.lib`.

- -EINVA: `len` parameter was < 0.
- -EBADF: The specified handle was not open or invalid.
- Any other negative values indicate an error.

SEE ALSO

`sspec_close`, `sspec_write`, `sspec_seek`, `sspec_tell`, `sspec_open`

sspec_readfile

```
int sspec_readfile( int sspec, char *buffer, long offset,
                    int len );
```

DESCRIPTION

Read a file (represented by the `sspec` index) into `buffer`, starting at `offset`, and only copying `len` bytes. For `xmem` files, this function automatically skips the first 4 bytes. Hence, an offset of 0 marks the beginning of the file contents, not the file length.

This function is intended for file types that do not require explicit open or close calls, that is, root or `xmem` files. It can also be called for FS2 files, but this is not recommended since each call requires the file to be opened, seeked, read then closed. Instead, use `sspec_open()`, `sspec_read()` and `sspec_close()` calls which are the most efficient.

`sspec_readfile()` has the advantage of being “stateless,” but the price to pay is great loss of efficiency (especially when sequential access is all that is required.)

This function will NOT work for compressed `xmem` files or DOS FAT files.

PARAMETERS

sspec	spec index
buffer	The buffer to put the file contents into.
offset	The offset from the start of the file, in bytes, at which copying should begin.
len	The number of bytes to copy.

RETURN VALUE

- 1: Failure.
- ≥0: Success, number of bytes copied.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_getlength`, `sspec_getfileloc`

sspec_readvariable

```
int sspec_readvariable( int sspec, char *buffer );
```

DESCRIPTION

Formats the variable associated with the specified `ServerSpec` structure, and puts a NULL-terminated string representation of it in `buffer`. The macro `SSPEC_XMEMVARLEN` (default is 20) defines the size of the stack-allocated buffer when reading a variable in `xmem`.

PARAMETERS

sspec	spec index
buffer	The buffer in which to put the variable.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_getvaraddr`

sspec_remove

```
int sspec_remove( int sspec );
```

DESCRIPTION

Removes a spec entry (by marking it unused). In the case of files, note that this function does not actually remove the file, only the reference to the file in the spec structure.

This only works for RAM table entries.

PARAMETERS

sspec spec index

RETURN VALUE

- 0: Success.
- 1: Failure (i.e., the index is already unused).

LIBRARY

ZSERVER.LIB

`sspec_remove_rule`

```
int sspec_remove_rule( char * pfx );
```

DESCRIPTION

Remove a rule from the dynamic resource rule table.

PARAMETER

pfx Prefix of resource name. This must be an exact match to one of the rules previously added using `sspec_addrule()`.

RETURN VALUE

≥0: OK
-1: Error. For example, the rule was not found, or maybe the rule was in the flash table (`f_rule_table`).

SEE ALSO

`sspec_addrule`

sspec_removeuser

```
int sspec_removeuser( int sspec, int userid );
```

DESCRIPTION

Removes the user group(s) that `userid` belongs to from the read and write access masks for the specified resource. This will deny access to other users who have the same group(s) as the current user.

This function is deprecated as of Dynamic C 8.50. Use the more general `sspec_setpermissions()` function instead.

PARAMETERS

sspec	spec index
userid	user index

RETURN VALUE

0: Success, user was removed.
-1: Failure, no such `userid` found.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_setuser`, `sspec_adduser`, `sspec_getusername`,
`sspec_getuserid`, `sspec_setpermissions`

sspec_resizeroofile

```
int sspec_resizeroofile( int spec_index, int new_size );
```

DESCRIPTION

Change the byte size of a SSPEC item stored in root memory. Item must be a ROOTFILE, thus the item must have been created with `sspec_addrootfile()`.

PARAMETERS

spec_index spec index of the item
new_size New size to assign to item.

RETURN VALUE

≥0: Successfully adjust size of item.
-1: Failed to adjust size.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_addrootfile`

sspec_restore

```
int sspec_restore( void );
```

DESCRIPTION

Restores the TCP/IP servers' object list and the TCP/IP users list (and some user-specified data if set up with `sspec_setsavedata()`) from the file system. This does not restore the actual files and variables, but only the structures that reference them. If the files are stored in flash, then the references will still be valid. Files in volatile RAM and variables must be rebuilt through other means.

RETURN VALUE

0: Successfully restored the `server_spec` and `server_auth` tables.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_save`, `sspec_setsavedata`

sspec_rmdir

```
int sspec_rmdir( char * name, ServerContext * context );
```

DESCRIPTION

Delete a named directory in the FAT filesystem.

PARAMETERS

name	Name of directory to delete.
context	Current server context.

RETURN VALUE

0: OK.
- EPERM: Not a filesystem that supports deletion of directories.
- EACCES: Not authorized
Any other negative values indicate an error.

SEE ALSO

sspec_delete, sspec_mkdir, sspec_open

sspec_save

```
int sspec_save( void );
```

DESCRIPTION

Saves the servers' object list and server authorization list (along with some user-specified data if set up with `sspec_setsavedata()`) to the file system. This does not save the actual files and variables, but only the structures that reference them. If the files are stored in flash, then the references will still be valid. Files in volatile RAM and variables must be rebuilt through other means.

RETURN VALUE

0: Successfully save the `server_spec` and `server_auth` tables.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_restore`, `sspec_setsavedata`

sspec_seek

```
int sspec_seek(int sspec, long offset, int whence);
```

DESCRIPTION

Seek to specified offset in the file resource. The next `sspec_read()` or `sspec_write()` call will start at this position.

Note that offsets that are not in the file are *clamped* to the start or end of the file as appropriate.

Clamp is terminology meaning that a value past the end is set to the end, or a value before the beginning is set to the beginning. For example, if a file is actually 10 bytes, then seek to position 20 is actually a seek to position 10. Likewise, seek to -20 is set to position 0.

PARAMETERS

sspec	Open file handle. This must be a handle that was returned by <code>sspec_open()</code> .
offset	Byte offset.
whence	Reference point for seek. One of the following constants: <ul style="list-style-type: none">• <code>SEEK_SET</code>: start of file, offset should be non-negative.• <code>SEEK_CUR</code>: current position in file, offset may be negative, zero, or positive.• <code>SEEK_END</code>: end of file, offset should be non-positive to stay within the file.

RETURN VALUE

0: OK.

The following return values are negatives of the values defined in `errno.lib`.

- `-EINVAL`: `whence` parameter was invalid.
- `-EBADF`: The specified handle was not open or invalid.
- `-EPERM`: Operation not permitted on this file resource. This is usually because the resource is not seekable (such as a compressed file).
- Any other negative values indicate an error.

SEE ALSO:

`sspec_close`, `sspec_write`, `sspec_read`, `sspec_tell`, `sspec_open`

`s`spec_setformepilog

```
int sspec_setformepilog( int form, int function );
```

DESCRIPTION

Sets the user-specified function that will be called when the form has been successfully submitted. This function can, for example, execute a `cgi_redirect` to redirect to a specific page. It should accept `HttpState *state` as an argument, return 0 when it is not finished, and 1 when it is finished (i.e., behave like a normal CGI function).

PARAMETERS

form	spec index of the form
function	spec index of the function to call when the specified form has been successfully submitted. This is the return value of the function <code>sspec_addfunction()</code> .

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_setformprolog`

`s_spec_setformfunction`

```
int s_spec_setformfunction( int form, void (*fptr)() );
```

DESCRIPTION

Sets the function that will generate the form.

PARAMETERS

form	spec index of the form.
fptr	Form generation function (NULL for the default function).

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

`sspec_setformprolog`

```
int sspec_setformprolog( int form, int function );
```

DESCRIPTION

Allows a user-specified function to be called just before form variables are updated. This is useful for implementing locking on the form variables (which can then be unlocked in the epilog function), so that other code will not update the variables during form processing. The user-specified function should accept `HttpState *state` as an argument, return 0 when it is not finished, and 1 when it is finished (i.e., behave like a normal CGI function).

PARAMETERS

form	spec index of the form
function	spec index of the function. This is the return value of <code>sspec_addfunction()</code> .

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_setformepilog`

sspec_setformtitle

```
int sspec_setformtitle( int form, char *title );
```

DESCRIPTION

Sets the title for an automatically generated form.

PARAMETERS

form	spec index of the form.
title	Pointer to the title of the HTML page.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getformtitle

sspec_setfvcheck

```
int sspec_setfvcheck( int form, int var, int (*varcheck)() );
```

DESCRIPTION

Sets a function that can be used to check the integrity of a variable. The function should return 0 if there is no error, or !0 if there is an error.

PARAMETERS

form	spec index of the form.
var	Index (into the <code>FormVar</code> array) of the variable.
varcheck	Pointer to integrity-checking function.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setfvfloatrange, sspec_setfvoptlist, sspec_setfvrange

`sspec_setfvdesc`

```
int sspec_setfvdesc( int form, int var, char *desc );
```

DESCRIPTION

Sets the description of a variable that is displayed in the HTML form table.

PARAMETERS

form	server_spec index of the form.
var	Index (into the FormVar array) of the variable.
desc	Description of the variable. This text will display on the HTML page.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getfvdesc

sspec_setfventrytype

```
int sspec_setfventrytype( int form, int var, int entrytype );
```

DESCRIPTION

Sets the type of form entry element that should be used for the given variable.

PARAMETERS

form	spec index of the form.
var	Index (into the FormVar array) of the variable.
entrytype	HTML_FORM_TEXT for a text box, HTML_FORM_PULLDOWN for a pull-down menu. The default is HTML_FORM_TEXT.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getfventrytype

sspec_setfvfloatrange

```
int sspec_setfvfloatrange( int form, int var, float low,
    float high );
```

DESCRIPTION

Sets the range of a float variable.

PARAMETERS

form	spec index of the form.
var	Index (into the <code>FormVar</code> array) of the variable.
low	Minimum value of the variable.
high	Maximum value of the variable.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_setfvrange`, `sspec_setfvoptlist`

`sspec_setfvlen`

```
int sspec_setfvlen( int form, int var, int len );
```

DESCRIPTION

Sets the length of a form variable (the maximum length of the string representation of the variable). Note that for string variables, `len` should *not* include the NULL terminator.

PARAMETERS

<code>form</code>	spec index of the form.
<code>var</code>	Index (into the <code>FormVar</code> array) of the variable.
<code>len</code>	Length of the variable.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_getfvlen`

`sspec_setfvname`

```
int sspec_setfvname( int form, int var, char *name );
```

DESCRIPTION

Sets the name of a variable that is displayed in the HTML form.

PARAMETERS

form	spec index of the form
var	Index (into the <code>FormVar</code> array) of the variable.
name	Display name of the variable.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_getfvname`

sspec_setfvoptlist

```
int sspec_setfvoptlist( int form, int var, char *list[],
    int listlen );
```

DESCRIPTION

Sets an enumerated list of possible values for a string variable.

PARAMETERS

form	spec index of the form.
var	Index (into the <code>FormVar</code> array) of the variable.
list []	Array of string values that the variable can assume.
listlen	Length of the array.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_getfvopt`, `sspec_getfvoptlistlen`, `sspec_setfvfloatrange`,
`sspec_setfvrange`

sspec_setfvrange

```
int sspec_setfvrange( int form, int var, long low, long high );
```

DESCRIPTION

Sets the range of an integer variable.

PARAMETERS

form	spec index of the form.
var	Index (into the FormVar array) of the variable.
low	Minimum value of the variable.
high	Maximum value of the variable.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setfvfloatrange, sspec_setfvoptlist

`sspec_setfvreadonly`

```
int sspec_setfvreadonly( int form, int var, int readonly );
```

DESCRIPTION

Sets the form variable to be read-only.

PARAMETERS

form	spec index of the form.
var	Index (into the <code>FormVar</code> array) of the variable.
readonly	0 for read/write (this is the default); 1 for read-only.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

`ZSERVER.LIB`

SEE ALSO

`sspec_getfvreadonly`

sspec_setpermissions

```
int sspec_setpermissions(int sspec, char * realm,
    word readgroups, word writegroups, word servermask,
    word method, MIMETypeMap * mimetype);
```

DESCRIPTION

Set the permission (access control) attributes of a resource.

This only works for RAM table entries. For entries in a filesystem, use `sspec_addrule()`.

PARAMETERS

sspec	spec index
realm	Realm string, or NULL
readgroups	Mask of user groups who have read access
writegroups	Mask of user groups who have write access
servermask	Servers that can access this resource (or <code>SERVER_ANY</code> for all servers).
method	Required authentication method (0, <code>SERVER_AUTH_BASIC</code> etc.)
mimetype	MIME table entry, or NULL.

RETURN VALUE

0: Success.
<0: Failure. For example, not a RAM spec handle.

SEE ALSO

`sspec_checkpermissions`, `sspec_getpermissions`, `sspec_access`

`sspec_setpreformfunction`

```
int sspec_setpreformfunction( int form, void (*fptr)() );
```

DESCRIPTION

Sets a user function that will be called just before form generation. The user function is not called when the form is being generated because of errors in the form input. The user function must have the following prototype:

```
void userfunction(int form);
```

The function may not use the `form` parameter, but it is useful if the same user function is used for multiple forms.

PARAMETERS

<code>form</code>	spec index of the form.
<code>fptr</code>	Pointer to user function to be called just before form generation

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_getpreformfunction`

sspec_setrealm

```
int sspec_setrealm( int sspec, char *realm );
```

DESCRIPTION

Sets the realm field of a `ServerSpec` structure for HTTP authentication purposes. Setting this field enables authentication for the given spec entry . Authentication can be turned off again by passing "" as the `realm` parameter to this function.

Note: `realm` must NOT point to an auto variable, since only the pointer is stored. The string is NOT copied.

PARAMETERS

sspec	spec index - this must refer to the RAM resource table
realm	Name of the realm.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_getrealm`

sspec_setsavedata

```
int sspec_setsavedata( char *data, unsigned long len,  
    void *fptr );
```

DESCRIPTION

Sets user-supplied data that will be saved in addition to the spec and user authentication tables when `sspec_save()` is called.

PARAMETERS

data	Pointer to location of user-supplied data.
len	Length of the user-supplied data in bytes.
fptr	Pointer to a function that will be called when the user-supplied data has been restored.

RETURN VALUE

0: Successfully set up the user-supplied data.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sspec_save`, `sspec_restore`

sspec_setuser

```
int sspec_setuser( int sspec, int userid );
```

DESCRIPTION

Set the read permission mask of a spec entry (usually a file). The permissions for this resource are set to readable only by the group(s) which this user is a member of. Write access is set to "none."

This function is deprecated in Dynamic C 8.50. Use `sspec_setpermissions()` instead.

PARAMETERS

sspec	spec index - this must refer to a RAM resource
userid	user index

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

`sauth_adduser`, `sspec_getusername`, `sspec_setpermissions`

sspec_stat

```
int sspec_stat(char * name, ServerContext * context,  
              SSpecStat * s);
```

DESCRIPTION

Get information about a resource by name. The name may refer to a flash- or ram-spec entry, or may be the name of a file in a filesystem. See `sspec_open()` for a more detailed description of the name and context parameters.

PARAMETERS

name	Resource name, as a null-terminated string. This name is assumed to be relative to <code>context->cwd</code> if it does not begin with a “/” character. Otherwise, the name is assumed to be relative to <code>context->rootdir</code> .
context	Additional context information. The <code>ServerContext</code> structure is set up by the caller.
s	Returned data. This is a pointer to the following structure, which will be filled in on return.

```
typedef struct  
{  
    word flags;           // See below.  
    long mdtm;           // Date/time (SEC_TIMER format)  
    long length;        // Current file size  
    long maxlength;     // Max allowable file size  
    ServerPermissions *perm; // See below.  
} SSpecStat;
```

The `flags` field can be one of the following:

- `SSPEC_ATTR_MDTM` - Modification date/time
- `SSPEC_ATTR_LENGTH` - Current length
- `SSPEC_ATTR_WRITE` - File is writable
- `SSPEC_ATTR_EXEC` - File is executable
- `SSPEC_ATTR_HIDDEN` - "Hidden" attribute bit
- `SSPEC_ATTR_SYSTEM` - "System" attribute bit
- `SSPEC_ATTR_ARCHIVE` - "Archive" attribute bit
- `SSPEC_ATTR_DIR` - This is directory name
- `SSPEC_ATTR_COMPRESSED` - Compressed format
- `SPEC_ATTR_MAXLENGTH` - Have maximum length
- `SSPEC_ATTR_SEEKABLE` - Randomly accessible
- `SSPEC_ATTR_EXTENSIBLE` - File may be expanded at end

The `ServerPermissions` structure is defined as follows:

```
typedef struct {
    word readgroups;
    word writegroups;
    word servermask;
    char * realm;
    char method;
} ServerPermissions;
```

Read (or write) permission is granted for `readgroups` (or `writegroups`) if current `ServerAuth.mask` (i.e., user id entry group mask) matches in at least one bit position.

Bit is set in `servermask` field for each server that can access the resource.

Realm string of the resource (only used by HTTP server, but can be used for other purposes).

Authentication method(s) allowed: combination of `SERVER_AUTH_*` bits.

RETURN VALUE

≥ 0 : Success.

The following return values are negatives of the values defined in `errno.lib`.

- `-ENOENT`: The resource was not found.
- `-EINVAL`: The resource name was malformed (for example, too long), or context was `NULL`, or the resource was not a file type.

SEE ALSO

`sspec_open`, `sspec_delete`, `sspec_close`

`sspec_tell`

```
long sspec_tell( int sspec );
```

DESCRIPTION

Return the current read/write offset in the file resource. This will be a non-negative value unless there was an error.

PARAMETER

sspec Open file handle. This must be a handle that was returned by `sspec_open()`.

RETURN VALUE:

≥0: Offset in the file resource.

The following return value is a negative of the value defined in `errno.lib`. Any other negative values indicate an error.

- `-EBADF`: The specified handle was not open or invalid.

SEE ALSO:

`sspec_close`, `sspec_write`, `sspec_read`, `sspec_tell`, `sspec_open`

sspec_write

```
int sspec_write( int sspec, char * buf, int len );
```

DESCRIPTION

Write byte(s) to the given file resource. The data is written to the current position, then the current position is advanced by the number of bytes written.

PARAMETERS

sspec	Open file handle. This must be a handle that was returned by <code>sspec_open()</code> .
buf	Buffer from which data is copied.
len	Length of the above buffer.

RETURN VALUE

0: No data was written because `len` was zero or because a local buffer is full (e.g., when writing to an underlying filesystem that streams data to a peer).

1 . . len: The specified number of characters has been copied from the supplied buffer, and the current file position has been advanced by that many bytes. Possibly less than `len` bytes may be written, in which case the server should attempt to write the remaining data later.

The following return values are negatives of the values defined in `errno.lib`.

- -EINVAL: `len` parameter was < 0.
- -EBADF: The specified handle was not open or invalid.
- -ENOSPC: There is insufficient space in the underlying filesystem, or the file cannot be extended.
- -EPERM: The file resource does not support writing (e.g. `xmem` files, or a read-only filesystem).

Any other negative values indicate an error.

SEE ALSO

`sspec_close`, `sspec_read`, `sspec_seek`, `sspec_tell`, `sspec_open`

4. HTTP Server

This chapter is intended to be a detailed description of the HTTP server, and how it interfaces to other libraries, such as Zserver and TCP/IP. For an overview of how these libraries interface with one another and with your application, please see Chapter 2. “Web-Enabling Your Application.”

An HTTP (Hypertext Transfer Protocol) server makes HTML (Hypertext Markup Language) pages and other resources available to clients (that is, web browsers). HTTP is implemented by `HTTP.LIB`, thus you need to write `#use "http.lib"` near the top of your program. HTTP depends on the Dynamic C networking suite, which is included in your program by writing `#use "dcrtcp.lib"`.

Setting up the network subsystem is a necessary pre-requisite for use of HTTP. This is described in volume 1 of the manual. However, it can be quite simple for test applications and samples to initialize the network subsystem. In the file `tcp_config.lib` are predefined configurations that may be accessed by a `#define` of the macro `TCPCONFIG`. For instructions on how to set up different configurations, please see volume 1 of the manual or look in the file `\LIB\TCPIP\TCP_CONFIG.LIB`.

HTTP makes use of the Zserver library to manage resources and access control. The previous chapter discusses Zserver. When reading this chapter, it will help if you are familiar with Zserver, its interfaces and capabilities.

Much of this chapter contains material that could be considered advanced usage. There is also some material of a historical nature, with relevant sections marked as such.

4.1 HTTP Server Data Structures

The single data structure in `HTTP.LIB` of interest to developers of CGI functions is discussed in this section.

4.1.1 `HttpState`

Use of the `HttpState` structure is necessary for CGI functions (whether or not they were written prior to Dynamic C 8.50). Some of the fields are off-limits to developers. The field that are available for use are described in the next section.

Historical note: prior to Dynamic C 8.50, it was sometimes necessary for CGI functions to access directly the fields of this structure. New programs should not directly access the fields, since it reduces the chance of upward compatibility. There is a new suite of macros (see `http_getAction()` and related macros) that should be used instead. Where applicable, the equivalent macro is documented with the field. Some fields do not have an equivalent macro (such as the cookie field); for now, use read-only access to such fields.

A pointer to `HttpState` is the first (and only) parameter to all CGI functions. Most of the time, this pointer should be passed on to other HTTP library functions.

Note that the `HttpState` structure is only valid within a CGI function that has been called from the HTTP server. Outside of this (for example, in your `main()` function) none of the fields are guaranteed to be meaningful or consistent.

4.1.1.1 `HttpState` Fields

The fields discussed here are available for developers to use in their CGI functions.

s This is the socket associated with the given HTTP server. A developer can use this in a CGI function to output dynamic data (although there are better, safer ways of doing this: see the section on "Writing a CGI Function"). Any of the TCP functions can be used; however, you should not use any functions that may wait for long periods, or may change the state or mode of the socket (since the HTTP server depends on it being a normal ASCII mode TCP socket).

It is recommended that you use the `http_getSocket()` macro instead of directly accessing this field.

substate
subsubstate

Intended for holding the current state of a state machine for a CGI function. That is, if a CGI function relinquishes control back to the HTTP server, then the values in these variables will be preserved for the next `http_handler()` call, in which the CGI function will be called again. These variables are initialized to 0 before the CGI function is called for the first time. Hence, the first state of a state machine using `substate` should be 0.

It is recommended that you use the macros `http_getState()` and `http_setState()` to manipulate the `substate` field instead of directly accessing it. `subsubstate` is not accessible via these macros, but there are better alternatives.

timeout	This value can be used by the CGI function to implement an internal time-out.
main_timeout	This value holds the timeout that is used by the web server. The web server checks against this timeout on every call of <code>http_handler()</code> . When the web server changes states, it resets <code>main_timeout</code> . When it has stayed in one state for too long, it cancels the current processing for the server and goes back to the initial state. Hence, a CGI function may want to reset this timeout if it needs more processing time (but care should be taken to make sure that the server is not locked up forever). This can be achieved like this: <pre>state->main_timeout=set_timeout(HTTP_TIMEOUT);</pre> <p><code>HTTP_TIMEOUT</code> is the number of seconds until the web server will time out. It is 16 seconds by default.</p>
buffer[]	A buffer that the developer can use to put data to be transmitted over the socket. It is of size <code>HTTP_MAXBUFFER</code> (defaults to 256 bytes). Note: It is not recommended to directly access “buffer” or “p” (below). Use the new-style CGI functions and the <code>http_write()</code> , <code>http_getData()</code> and <code>http_getDataLength()</code> functions instead. These create a much easier-to-use and safer method of reading/writing data to the client.
p	Pointer into the buffer given above. See above note.
method	This should be treated as read-only. It holds the method by which the web request was submitted. The value is either <code>HTTP_METHOD_GET</code> or <code>HTTP_METHOD_POST</code> , for the GET and POST request methods, respectively. Use <code>http_getHTTPMethod()</code> for new code.
url[]	This should be treated as read-only. It holds the URL by which the current web request was submitted. . If there is GET-style form information, then that information will follow the first NULL byte in the url array. The form information will itself be NULL-terminated. If the information in the url array is truncated to <code>HTTP_MAXURL</code> bytes, the truncated information is also NULL-terminated. Use <code>http_getURL()</code> for new code.
version	This should be treated as read-only. This holds the version of the HTTP request that was made. It can be <code>HTTP_VER_09</code> , <code>HTTP_VER_10</code> , or <code>HTTP_VER_11</code> for 0.9, 1.0, or 1.1 requests, respectively. Use <code>http_getHTTPVersion()</code> for new code.

content_type[]	This should be treated as read-only. This buffer holds the value from the Content-Type header sent by the client. Use <code>http_getContentType()</code> for new code.
content_length	This should be treated as read-only. This variable holds the length of the content sent by the client. It matches the value of the Content-Length header sent by the client. Use <code>http_getContentLength()</code> for new code.
has_form	This should be treated as read-only. If the value is 1 there is a GET style form, after the <code>\0</code> byte in <code>url[]</code> .
abort_notify	Set to !0 in user-defined <code>formprolog()</code> function to indicate that the <code>formepilog()</code> function needs to be called on an abort condition. If the epilog function is reached normally, this field must be set to zero. This prevents the <code>formepilog</code> function from being called one more time on a connection abort.
cancel	This should be treated as read-only. It is intended for when the user-defined functions, which may be called before and after an HTML form is submitted, are used for locking resources. If the <code>formprolog</code> function was called and then the connection is aborted before the <code>formepilog</code> function can be called, <code>cancel</code> is set to 1 and the <code>formepilog</code> function is called exactly once. If the epilog function was already called but returned zero (not finished yet), then it is called again if the connection is aborted, except if <code>cgi_redirectto()</code> has been called from the epilog function. In that case the epilog function is not called after an abort.
username[]	Read-only buffer has username of the user making the request, if authentication took place. Note: New code should use the <code>http_getContext()</code> macro, then use the results to look up the user details using the <code>sauth_*</code> functions. See the documentation for the ServerContext Structure in the previous chapter.
password[]	Read-only buffer has password of the user making the request, if authentication took place. See the above note.
cookie[]	Read-only buffer contains the value of the cookie "DCRABBIT" (see <code>http_setcookie()</code> for more information).

headerlen	These variables can be used together to cause the web server to flush data from the <code>buffer []</code> array in the <code>HttpState</code> structure. <code>headerlen</code> should be set to the amount of data in <code>buffer []</code> , and <code>headeroff</code> should be set to 0 (to indicate the offset into the array). The next time the CGI function is called the data in <code>buffer []</code> will be flushed to the socket.
headeroff	
	For new code, consider writing a new-style CGI function, which obviates the need to manipulate these fields.
cond []	Support for conditional SSI (error feedback etc.). New code should use the macros <code>http_getCond()</code> and <code>http_setCond()</code> .
userdata []	This field is included if <code>HTTP_USERDATA_SIZE</code> is defined. It is an optional user data area. The area is cleared to zero when the structure is initialized, otherwise it is not touched. Its size must be greater than zero. New code should use the <code>http_getUserData()</code> macro to obtain a pointer to user-defined storage in this structure.

4.2 Configuration Macros

The following macros are specified in `HTTP.LIB`. Unless otherwise noted, you can override the default values by defining the macro (same name, different value) before you `#use "http.lib"`.

HTTP_HOMEDIR

Specify the “home directory” for the server. This is the root directory to which all URLs are appended. The default is “/”, which means that all resources are accessible. If this is set to, say, “/htdocs”, then an incoming URL of “foo/bar.html” gets turned into “/htdocs/foo/bar.html”. You can use this to restrict the HTTP server’s access to all but a specific “branch” of resources.

Note: the string value for this macro must start and end with a “/” character.

HTTP_DFLTFILE

Specify the default file name to append to the URL if the URL refers to a directory. This is only applicable if the URL is “/”, or is in a filesystem (not the static or dynamic resource tables). The default setting is “index.html”. The value must *not* start or end with a “/” character.

HTTP_SOCK_BUF_SIZE

This macro is not defined by default. If you define it, then it specifies the amount of extended memory to allocate (`xalloc()`) for each HTTP server instance. If you do not define it, then socket buffers are allocated from the usual pool. See `tcp_extopen()` for more details.

HTTP_DIGEST_NONCE_TIMEOUT

This macro is used when `USE_HTTP_DIGEST_AUTHENTICATION` is set to one. Nonces that are generated by the server are valid for this many seconds (900 by default). If set to 0, nonces are good forever. Setting this to a smaller value can possibly result in higher security, although internal use of the nonce-count facility offsets this. Setting it to a larger value reduces the negotiation between the browser and the server, since when a nonce times out, the browser must be told that it is using a stale nonce value and provided with a new one. Since Mozilla and Netscape ignore the stale parameter, the user must reenter the username and password when a nonce times out. Internet Explorer and Opera respect the stale parameter, so they automatically try the username and password with the new nonce without asking the user.

HTTP_MAXBUFFER

This is the size of the buffer accessible through the `HttpSpec` structure. It defaults to 256 bytes. The size of this buffer affects the speed of the HTTP server; the larger the buffer (up to a point), the faster the server will run. The buffer size is also important for use in CGI functions because it is a work space the programmer can use.

`HTTP_MAXBUFFER` must be at least 180 bytes for CGI functionality.

HTTP_MAX_COND

Support for conditional SSI (error feedback etc.). It defaults to 4. This is the maximum number of state variables that may be accessed using the `http_getCond()` or `http_setCond()` macros.

HTTP_MAX_NONCES

This macro is used when `USE_HTTP_DIGEST_AUTHENTICATION` is set to one. Defined to 5 by default, it specifies the number of nonces the HTTP server will allow as valid at any one time. This value should be somewhat larger than the maximum number of clients expected to be accessing the server simultaneously. Otherwise performance could suffer as clients are forced to retry authorization in order to acquire a fresh nonce.

HTTP_MAXSERVERS

This is the maximum number of HTTP servers listening on port 80. The default is 2. You may increase this value to the maximum number of independent entities on your page. For example, for a Web page with four pictures, two of which are the same, set `HTTP_MAXSERVERS` to 4: one for the page, one for the duplicate images, and one for each of the other two images. By default, each server takes 2500 bytes of RAM. This RAM usage can be changed by the macro `SOCK_BUF_SIZE` (or `tcp_MaxBufSize` which is deprecated as of Dynamic C ver. 6.57). Another option is to use the `tcp_reserveport()` function and a smaller number of sockets.

HTTP_MAXURL

This macro defines the maximum incoming URL. This could be important if someone is allowing GET requests with a large number of parameters.

HTTP_PORT

This macro allows the user to override the default port of 80.

HTTP_IFACE

This macro allows the user to override the default listening network interface. The default is `IF_ANY`, meaning that the HTTP server(s) will listen for incoming network connections on all interfaces which are up. You can restrict the HTTP servers to a single interface by overriding this macro to the specific interface number (for example, `IF_ETH0`).

HTTP_TIMEOUT

Defines the number of seconds of no activity that can elapse before the HTTP server closes a connection. The default is 16 seconds.

HTTP_USERDATA_SIZE

This macro causes "char userdata[]" to be added to the `HttpState` structure. Define your structure before the statement `#use HTTP.LIB`.

```
struct UserStateData {char name[50]; int floor; int model;};
#define HTTP_USERDATA_SIZE (sizeof(struct UserStateData))
#use http.lib
```

In your own CGI function code, access it using:

```
mystate = (struct UserStateData *)http_getUserData(state);
```

USE_HTTP_DIGEST_AUTHENTICATION

Set to 1 to enable digest authentication, 0 to disable digest authentication. Set to 0 by default.

USE_HTTP_BASIC_AUTHENTICATION

Set to 1 to enable basic authentication, 0 to disable basic authentication. Set to 1 by default.

4.2.1 Sending Customized HTTP Headers to the Client

The callback macro, `HTTP_CUSTOM_HEADERS`, will be called whenever HTTP headers are being sent. It must be defined as a function with the following prototype:

```
void my_headers(HttpState *state, char *buffer, int bytes);
```

state Pointer to the state structure for the calling web server.

buffer The buffer in which the header(s) can be written.

bytes The number of bytes available in the buffer.

Typically, the macro would be defined by the user before the `#use "http.lib"` statement, like in the following:

```
#define HTTP_CUSTOM_HEADERS(state, buffer, bytes) \
    my_headers(state, buffer, bytes)
```

Then, for the above to work, `my_headers()` must be defined by the user, like so:

```
void my_headers(HttpState *state, char *buffer, int bytes)
{
    strcpy(buffer, "Fake-Header: Hello Z-World!\r\n");
    printf("bytes: %d\n", bytes);
}
```

In the real world, the user may need to check the number of bytes available to be sure they don't overwrite the buffer. The buffer must end with `"\r\n"` and be NULL-terminated.

4.2.2 Saving Custom Headers from the Client

Customers may want to save some specific headers that a web client sends to the server as part of a request. One possibility for this is to check the browser version of the client and display a different page depending on that value. This is mostly useful for CGI functions.

The user can create a structure like the following to indicate to the web server that it should save the specified tags:

```
const HttpHeaders http_headers[] = {
    "Host",
    "Content-Length",
    "User-Agent",
    END_HTTP_HEADERS
};
```

`END_HTTP_HEADERS` is simply a macro (NULL) that indicates the end of the structure. These headers will be saved in an internal buffer of a user-specified size:

```
#define HTTP_CUSTOM_HEADERS_SIZE 1024
```

By default, `HTTP_CUSTOM_HEADERS_SIZE` is undefined, which disables the custom header functionality (since, in most cases, it will not need to be used). This buffer will be located in `xmem`, and there will be one per HTTP server. A define will also be provided to limit the maximum size of a single header (to keep one very long header from monopolizing all of the buffer space):

```
#define HTTP_CUSTOM_HEADER_MAX_SIZE 128
```

By default, this is undefined and there is no limit.

The user will also need functions that look up the data:

```
int http_getheader(HttpState *state, char *header,
                  char *dest, int destlen);

int http_xgetheader(HttpState *state, char *header,
                   long *destptr);
```

The first function requires the user to provide a root buffer to place the header. The `HttpState` state structure must be passed so that the server knows which set of headers to access. The header parameter is, of course, the name of the header the user wants to retrieve. `dest` is a pointer to the destination buffer. `destlen` is the length of the destination buffer (provided by the user). The function returns -1 on error, and the number of bytes in the header on success.

The second function, `http_xgetheader()`, simply returns a long pointer into the internal header buffer for the given header. It returns -1 on error, and the number of bytes in the header on success.

Note that some headers are saved by the HTTP server by default into the HTTP state structure, such as “Content-Length.” We will also begin saving the “Host” header, which is useful in performing CGI redirection. Hence, we can change the semantics of the `cgi_redirectto()` function:

```
int cgi_redirectto(HttpState *state, char *url);
```

such that the `url` parameter no longer needs to be an absolute URL.

4.3 Authentication Methods

HTTP/1.0 Basic Authentication is used by default. This scheme is not a secure method of user authentication across an insecure network (e.g., the Internet). HTTP/1.0 does not, however, prevent additional authentication schemes and encryption mechanisms from being employed to increase security.

Starting with Dynamic C version 8.01, HTTP Digest Authentication as specified in RFC 2617 is supported. Instead of sending the password in cleartext as is done using Basic Authentication, MD5 is used to perform a cryptographic hash. To use HTTP Digest Authentication, define `USE_HTTP_DIGEST_AUTHENTICATION` as 1. When this `USE_*` macro is defined, the macros `HTTP_MAX_NONCES` and `HTTP_DIGEST_NONCE_TIMEOUT` are available; they affect negotiation time between server and client. For more details see Section 4.2 “Configuration Macros.”

In either case (basic and/or digest), you will need to add the appropriate rules and/or permissions to the appropriate tables. See the previous chapter for details on protecting resources. The HTTP server applies the strongest applicable authentication mechanism depending on the information it retrieves from the resource manager. Typically, in addition to defining user IDs and groups, you also need to associate an authentication mechanism with the resource using e.g. the `SSPEC_MM_RULE` macro, or the `sspec_setpermissions()` function.

Starting with Dynamic C 8.50, Secure Socket Layer (SSL) as specified in RFC 2818, is supported. It is also known by its newer official name, TLS (Transport Layer Security). To use SSL, you must create a secure HTTP server, known as an HTTPS server. To do this you must define some macros and import the SSL certificate.

```
#define USE_HTTP_SSL
#define HTTP_SSL_SOCKETS 1

#ximport "cert\mycert.dcc" SSL_CERTIFICATE
```

For complete documentation on the Dynamic C implementation of SSL, see the Dynamic C Module document, “Secure Sockets Layer (SSL) Protocol.” Another good source of information are the sample programs that demonstrate using SSL. They are located in the `/Samples/tcpip/ssl` folder that will be created when the Dynamic C Module for SSL is installed.

4.4 Setting the Time Zone

The HTTP specification requires the server to indicate its current clock time in the response to any request. The HTTP implementation performs this function by consulting the `rtc_timezone()` library function (in `RTCLOCK.LIB`). The server uses the returned time zone to adjust the local real-time clock (RTC) value so that it is always returned to the client in UTC (Co-ordinated Universal Time).

There are several macros which you can set to define

- `TIMEZONE`: The local timezone offset from UTC.
- `RTC_IS_UTC`: Whether the RTC is already running on UTC.

The local timezone offset may be defined using the `TIMEZONE` macro, or it may be obtained automatically from a DHCP server if you are using DHCP to configure the network interface. Failing that, it defaults to zero.

If the RTC is already set to UTC (not local time), then you must define the macro `RTC_IS_UTC`, in which case the local timezone offset will be ignored.

For many reasons, including the fact that daylight savings transitions are more manageable, it is better to set the RTC to UTC, however some users prefer the clock to run in local time.

See the documentation for `rtc_timezone()` for more details. To do this, use the function lookup feature in Dynamic C or refer to the *Dynamic C Function Reference Manual*.

4.5 Sample Programs

Sample programs demonstrating HTTP are in the `Samples\Tcpip\Http` directory. There is a configuration block at the beginning of each sample program. The macros in this block need to be changed to reflect your network settings.

Starting with Dynamic C 7.30, setting up the network addresses is both more complex and more simple. The complexity lies in the added support for multiple interfaces. Luckily for us, the simplicity is in the interface to this more intricate implementation. In the file `tcp_config.lib` are predefined configurations that may be accessed by a `#define` of the macro `TCPCONFIG`. For instructions on how to set the configuration, please see volume 1 of the manual or `LIB\TCPIP\TCP_CONFIG.LIB`.

4.5.1 Serving Static Web Pages

The sample program, `Static.c`, initializes `HTTP.LIB` and then sets up a basic static web page. It is assumed you are on the same subnet as the controller. The code for `Static.c` is explained in the following pages.

From Dynamic C, compile and run the program. You will see the LNK light on the board come on after a couple of seconds. Point your internet browser at the controller (e.g., `http://10.10.6.100/`). The ACT light will flash a couple of times and your browser will display the page.

Program Name: `Static.c`

```
#define TCPCONFIG 1
#define TIMEZONE -8
#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"
#ximport "samples/tcpip/http/pages/static.html" index_html
#ximport "samples/tcpip/http/pages/rabbit1.gif" rabbit1_gif
SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME(".gif", "image/gif")
SSPEC_MIMETABLE_END
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_XMEMFILE("/rabbit1.gif", rabbit1_gif)
SSPEC_RESOURCETABLE_END
main()
{
    sock_init();          // Initializes the TCP/IP stack
    http_init();         // Initializes the web server

    tcp_reserveport(80);
    while (1) {
        http_handler();
    }
}
```

The program serves the `static.html` file and the `rabbit1.gif` file to any user contacting the controller. If you want to change the file that is served by the controller, find and modify this line in `Static.c`:

```
#ximport "samples/tcpip/http/pages/static.html" index_html
```

Replace `static.html` with the name of the file you want the controller to serve.

4.5.1.1 Adding Files to Display

Adding additional files to the controller to serve as web pages is slightly more complicated. First, add an `#ximport` line with the filename as the first parameter, and a symbol that references it in Dynamic C as the second parameter.

```
#ximport "samples/tcpip/http/pages/static.html" index_html
#ximport "samples/tcpip/http/pages/newfile.html" newfile_html
```

Next, find these lines in `Static.c`:

```
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_XMEMFILE("/rabbit1.gif", rabbit1_gif)
SSPEC_RESOURCETABLE_END
```

Insert the name of your new file, preceded by “/”, into this structure, using the same format as the other lines.

```
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_XMEMFILE("/newfile.html", newfile_html),
    SSPEC_RESOURCE_XMEMFILE("/rabbit1.gif", rabbit1_gif)
SSPEC_RESOURCETABLE_END
```

Compile and run the program. Open up your browser to the new page (for example, “`http://10.10.6.100/newfile.html`”), and your new page will be displayed by the browser.

4.5.1.2 Adding Files with Different Extensions

If you are adding a file with an extension that is not `html` or `gif`, you need to use the appropriate macros to make an entry in the `MIMETypeMap` structure for the new extension. The first field is the extension and the second field describes the MIME type for that extension. You can find a list of MIME types at:

`ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types`

In the `media-types` document located there, the text in the type column would precede the “/”, and the subtype column would directly follow. Find the type subtype entry that matches your extension and add it to the `http_types` table.

```

SSPEC_MIMETABLE_START
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME(".pdf", "application/pdf"), //added this one
    SSPEC_MIME(".gif", "image/gif")
SSPEC_MIMETABLE_END

```

4.5.1.3 Handling of Files With No Extension

The entry “/” and files without an extension are dealt with by the handler specified in the first entry in the MIME table. If you use the `SSPEC_MIME` macro, the default handler is used. It passes the information verbatim. You can also use the macro `SSPEC_MIME_FUNC` to specify a non-default text processor; this is necessary for SSI and RabbitWeb scripts (described later).

4.5.2 Dynamic Web Pages Without HTML Forms

Serving a dynamic web page without the use of HTML forms is done by sample program `ssi.c`. This program displays four “lights” and four buttons to toggle them. Users can browse to the device and change the status of the lights.

The sample code follows, but it has been edited for brevity. Open `ssi.c` in Dynamic C to see the fully-commented source.

```

#define TCPCONFIG 1
#define HTTP_MAXSERVERS 1
#define MAX_TCP_SOCKET_BUFFERS 1
#define REDIRECTHOST _PRIMARY_STATIC_IP
#define REDIRECTTO "http://" REDIRECTHOST "/index.shtml"
#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"

#ximport "samples/tcpip/http/pages/ssi.shtml" index_html
#ximport "samples/tcpip/http/pages/rabbit1.gif" rabbit1_gif
#ximport "samples/tcpip/http/pages/ledon.gif" ledon_gif
#ximport "samples/tcpip/http/pages/ledoff.gif" ledoff_gif
#ximport "samples/tcpip/http/pages/button.gif" button_gif
#ximport "samples/tcpip/http/pages/showsrc.shtml" showsrc_shtml
#ximport "samples/tcpip/http/ssi.c" ssi_c

SSPEC_MIMETABLE_START
    SSPEC_MIME_FUNC(".shtml", "text/html", shtml_handler),
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME(".gif", "image/gif"),
    SSPEC_MIME(".cgi", "")
SSPEC_MIMETABLE_END

```

```

char led1[15];
char led2[15];
char led3[15];
char led4[15];

int led1toggle(HttpState* state){
    if (strcmp(led1,"ledon.gif")==0)
        strcpy(led1,"ledoff.gif");
    else
        strcpy(led1,"ledon.gif");
    cgi_redirectto(state,REDIRECTTO);
    return 0;
}

int led2toggle(HttpState* state){
    // Entirely analogous to led1toggle
}

int led3toggle(HttpState* state){
    // Entirely analogous to led1toggle
}

int led4toggle(HttpState* state){
    // Entirely analogous to led1toggle
}

SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/", index_html),
    SSPEC_RESOURCE_XMEMFILE("/index.shtml", index_html),
    SSPEC_RESOURCE_XMEMFILE("/showsrc.shtml", showsrc_shtml),
    SSPEC_RESOURCE_XMEMFILE("/rabbit1.gif", rabbit1_gif),
    SSPEC_RESOURCE_XMEMFILE("/ledon.gif", ledon_gif),
    SSPEC_RESOURCE_XMEMFILE("/ledoff.gif", ledoff_gif),
    SSPEC_RESOURCE_XMEMFILE("/button.gif", button_gif),
    SSPEC_RESOURCE_XMEMFILE("/ssi.c", ssi_c),
    SSPEC_RESOURCE_ROOTVAR("led1", led1, PTR16, "%s"),
    SSPEC_RESOURCE_ROOTVAR("led2", led2, PTR16, "%s"),
    SSPEC_RESOURCE_ROOTVAR("led3", led3, PTR16, "%s"),
    SSPEC_RESOURCE_ROOTVAR("led4", led4, PTR16, "%s"),
    SSPEC_RESOURCE_FUNCTION("/led1tog.cgi", led1toggle),
    SSPEC_RESOURCE_FUNCTION("/led2tog.cgi", led2toggle),
    SSPEC_RESOURCE_FUNCTION("/led3tog.cgi", led3toggle),
    SSPEC_RESOURCE_FUNCTION("/led4tog.cgi", led4toggle)
SSPEC_RESOURCETABLE_END

```

```
void main() {
    strcpy(led1, "ledon.gif");
    strcpy(led2, "ledon.gif");
    strcpy(led3, "ledoff.gif");
    strcpy(led4, "ledon.gif");

    sock_init();
    http_init();

    tcp_reserveport(80);
    while (1) http_handler();
}
```

When you compile and run `ssi.c`, you see the LNK light on the board come on. Point your browser at the controller (e.g., `http://10.10.6.100/`). The ACT light will flash a couple of times and your browser will display the page.

This program displays pictures of LEDs. Their state is toggled by pressing the image of a button. This program uses Server Side Includes (SSI) and the old style of CGI (`SSPEC_RESOURCE_FUNCTION`). Use of SSI is explained in greater detail below.

4.5.2.1 SSI Feature

SSI commands are an extension of the HTML comment command (`<!--This is a comment -->`). They allow dynamic changes to HTML files and are resolved at the server side, so the client never sees them. HTML files that need to be parsed because they contain SSI commands, are conventionally recognized by the HTTP server by the resource name extension `.shtml`.¹

The supported SSI commands are:

- `#echo var`
- `#exec cmd`
- `#include file`

They are used by inserting the command into an HTML file:

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

The server replaces the command, `#include file`, with the contents of `anyfile`.

`#exec cmd` executes a command i.e. an old-style CGI and replaces the SSI command with the output.

Dynamically Changing the Display of a Variable on a Web Page

The `Ssi.shtml` file, located in `Samples\Tcpip\Http\Pages`, gives an example of dynamically changing a variable on a web page using `#echo var`.

```
<img SRC="<!--#echo var="led1" -->">
```

In an `shtml` file, the `<!--#echo var="led1" -->` is replaced by the value of the variable `led1` from the static resource table.

```
SSPEC_RESOURCECETABLE_START
...
SSPEC_RESOURCE_ROOTVAR("led1", led1, PTR16, "%s"),
SSPEC_RESOURCE_ROOTVAR("led2", led2, PTR16, "%s"),
SSPEC_RESOURCE_ROOTVAR("led3", led3, PTR16, "%s"),
SSPEC_RESOURCE_ROOTVAR("led4", led4, PTR16, "%s"),
...
SSPEC_RESOURCECETABLE_END
```

`shtml_handler` (which is the built-in script processor for SSI) looks up `led1` and replaces it with the text output from:

```
printf("%s", (char*)led1);
```

The `led1` variable is either `ledon.gif` or `ledoff.gif`. When the browser loads the page, it replaces

```
<img SRC="<!--#echo var="led1"-->">
```

with

```
<img SRC="ledon.gif">
```

1. This is just a convention. If you add a `MIMETypeMap` entry `SSPEC_MIME_FUNC("shtml", "text/html", shtml_handler)` then you are following this convention.

or

```
<img SRC="ledoff.gif">
```

This causes the browser to load the appropriate image file.

SSI string variables are only appropriate for relatively short strings. (In the above example, the SSI string variables are “ledon.gif” and “ledoff.gif.”) The size that can be output is limited to the size `HTTP_MAXBUFFER`. If you need larger strings, you should either increase `HTTP_MAXBUFFER` (which will use more root RAM) or switch to using a CGI function.

4.5.2.2 CGI Feature

`Ssi.c` also demonstrates the Common Gateway Interface. CGI is a standard for interfacing external applications with HTTP servers. Each time a client requests a URL corresponding to a CGI program, the server will execute the CGI program in real-time.

For increased flexibility, a CGI function is responsible for outputting its own HTTP headers. Information about HTTP headers can be found at:

<http://deesse.univ-lemans.fr:8003/Connected/RFC/1945/>

and many other web sites and books. In the `Ssi.shtml` file, this line creates the clickable button viewable from the browser.

```
<TD> <A HREF="/led1tog.cgi"> <img SRC="button.gif"> </A> </TD>
```

When the user clicks on the button, the browser will request the `/led1tog.cgi` entity. This causes the HTTP server to examine the contents of the `http_flashspec` structure looking for `/led1tog.cgi`. It finds it and notices that `led1toggle()` needs to be called.

The `led1toggle` function changes the value of the `led1` variable, then redirects the browser back to the original page. When the original page is reloaded by the browser, the LED image will have changed states to reflect the user’s action.

This sample demonstrates the so-called “old-style” CGI. New-style CGIs are easier to write (especially when they are doing something non-trivial). They are described in Section 4.6 “HTTP File Upload.”

Connection Abort Condition

There are two fields in the `HttpState` structure that allow a CGI function to appropriately respond to a connection abort condition. The user may set the field `abort_notify` to a non-zero value in a CGI function to request that the CGI function be called one more time with the `cancel` field set to one if a connection abort occurs.

4.5.3 Web Pages With HTML Forms

With a web browser, HTML forms enable users to input values. With a CGI program, those values can be sent back to the server and processed. The FORM and INPUT tags are used to create forms in HTML.

The FORM tag specifies which elements constitute a single form and what CGI program to call when the form is submitted. The FORM tag has an option called ACTION. This option defines what CGI program is called when the form is submitted (when the “Submit” button is pressed). The FORM tag also has an option called METHOD that defines the method used to return the form information to the web server. In Section 4.5.3.1, “Sample HTML Page,” on page 171, the POST method is used, which will be described later. All of the HTML between the <FORM> and </FORM> tags define what is contained within a form.

Starting with Dynamic C 8.50, you can also use the enctype option inside the FORM tag. This specifies a return encoding type for the form’s information. If you did not specify this option, then you can use old-style CGIs (as described in this section). If you specify enctype="multipart/form-data" then you should specify a new-style CGI instead. See Section 4.6 describing the HTTP upload feature for more details on writing a new-style CGI.

The INPUT tag defines a specific form element, the individual input fields in a form. For example, a text box in which the user may type in a value, or a pull-down menu from which the user may choose an item. The TYPE parameter defines what type of input field is being used. In the following example, in the first two cases, it is the text input field, which is a single-line text entry box. The NAME parameter defines what the name of that particular input variable is, so that when the information is returned to the server, then the server can associate it with a particular variable. The VALUE parameter defines the current value of the parameter. The SIZE parameter defines how long the text entry box is (in characters).

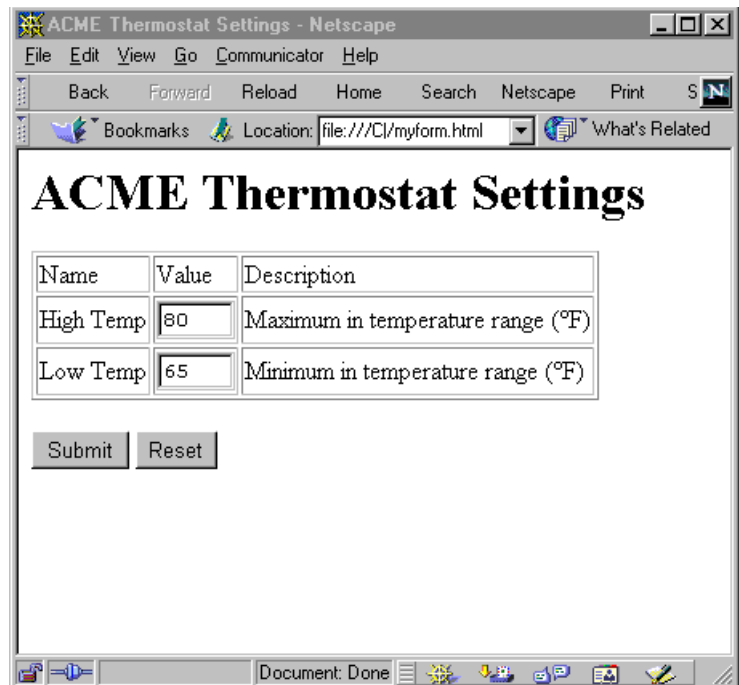
At the end of the HTML page in our example, the Submit and Reset buttons are defined with the INPUT tag. These use the special types “submit” and “reset,” since these buttons have special purposes. When the submit button is pressed, the form is submitted by calling the CGI program “myform.”

4.5.3.1 Sample HTML Page

An HTML page that includes a form may look like the following:

```
<HTML><HEAD><TITLE>ACME Thermostat Settings</TITLE></HEAD>
<BODY>
<H1>ACME Thermostat Settings</H1>
<FORM ACTION="myform.html" METHOD="POST">
  <TABLE BORDER>
    <TR>
      <TD>Name</TD> <TD>Value</TD> <TD>Description</TD></TR>
    <TR>
      <TD>High Temp</TD>
      <TD><INPUT TYPE="text" NAME="temphi" VALUE="80"
        SIZE="5">
      </TD>
      <TD>Maximum in temperature range (&deg;F)</TD></TR>
    <TR>
      <TD>Low Temp</TD>
      <TD><INPUT TYPE="text" NAME="templo" VALUE="65"
        SIZE="5">
      </TD>
      <TD>Minimum in temperature range (&deg;F)</TD></TR>
  </TABLE>
  <P>
    <INPUT TYPE="submit" VALUE="Submit">
    <INPUT TYPE="reset" Value="Reset">
  </P>
</FORM></BODY>
</HTML>
```

The form might display as follows:



When the form is displayed by a browser, the user can change values in the form. But how does this changed data get back to the HTTP server? By using the HTTP POST command. When the user presses the “Submit” button, the browser connects to the HTTP server and makes the following request:

```
POST myform HTTP/1.0
.
. (some header information)
.
Content-Length: 19
```

where “myform” is the CGI program that was specified in the ACTION attribute of the FORM tag and POST is the METHOD attribute of the FORM tag. “Content-Length” defines how many bytes of information are being sent to the server (not including the request line and the headers).

Then, the browser sends a blank line followed by the form information in the following manner:

```
temphi=80&templo=65
```

That is, it sends back name and value pairs, separated by the ‘&’ character. (There can be some further encoding done here to represent special characters, but we will ignore that in this explanation.) The server must read in the information, decode it, parse it, and then handle it in some fashion. It will examine the new values, and assign them to the appropriate C variables if they are valid.

4.5.3.2 POST-Style Form Submission

If an HTML file specifies a POST-style form submission (that is, METHOD="POST"), the form will still be waiting on the socket when the old-style CGI handler is called. Therefore, it is the job of the CGI handler to read this data off the socket and parse it in a meaningful way. The sample files `Post.c` and `Post2.c` in the `\Samples\Tcpip\Http` folder show how to do this.

The HTTP POST command can put any kind of data onto the network. There are many encoding schemes currently used, but we will only look at URL-encoded data in this document. Other encoding schemes can be handled in a similar manner.

4.5.3.3 URL-Encoded Data

URL-encoded data is of the form "name1=value1&name2=value2," and is similar to the CGI form submission type passed in normal URLs. This has to be parsed to name=value pairs. The rest of this section details an extensible way to do this.

This initializes two possible HTML form entries to be received, and a place to store the results.

```
#define MAX_FORMSIZE 64
typedef struct {
    char *name;
    char value[MAX_FORMSIZE];
} FORMType;
FORMType FORMSpec[2];
void init_forms(void) {
    FORMSpec[0].name = "user_name";
    FORMSpec[1].name = "user_email";
}
```

Reading & Storing URL-encoded Data

`parse_post()` is called from the CGI function (`submit()`) to read URL-encoded data off the network. It calls `http_scanpost()` to store the data in `FORMSpec[]`. These code snippets are from `Samples\tcpip\http\post.c`.

```
int parse_post(HttpState *state) {
    auto int retval;
    auto int i;
    retval = sock_ared(&state->s, state->p, \
        (state->content_length < HTTP_MAXBUFFER-1)?\
        (int)state->content_length:HTTP_MAXBUFFER-1);
    if (retval < 0)
        return 1;
    state->subsubstate += retval;
    if (state->subsubstate >= state->content_length) {
        state->buffer[(int)state->content_length] = '\0';
        for(i=0; i<(sizeof(FORMSpec)/sizeof(FORMType)); i++) {
            http_scanpost(FORMSpec[i].name, state->buffer, \
                FORMSpec[i].value, MAX_FORMSIZE);
        }
        return 1;
    }
    return 0;
}
```

4.5.3.4 Sample of a CGI Handler

This next function is the CGI handler that calls `parse_post()`. It is a state machine-based handler that generates the page. It calls `parse_post()` and references the structure that is now filled with the parsed data we wanted.

This function is from `Samples\tcpip\http\post.c`.

```
int submit(HttpState *state) {
    auto int i;
    if(state->length) { // buffer to write out
        if(state->offset < state->length) {
            state->offset += sock_fastwrite(&state->s, state->buffer
                + (int)state->offset, (int)state->length -
                (int)state->offset);
        } else {
            state->offset = 0;
            state->length = 0;
        }
    } else {
        switch(state->substate) {
            case 0:
                strcpy(state->buffer, "HTTP/1.0 200 OK\r\n\r\n");
                state->length = strlen(state->buffer);
                state->offset = 0;
                state->substate++;
                break;
            case 1:
                strcpy(state->buffer, "<html><head><title>Results</title>
                    </head><body>\r\n");
                state->length = strlen(state->buffer);
                state->substate++;
                break;
            case 2: // initialize the FORMSpec data
                FORMSpec[0].value[0] = '\0';
                FORMSpec[1].value[0] = '\0';
                state->p = state->buffer;
                state->substate++;
                break;
        }
    }
}
```

```

case 3:                                     // parse the POST information
    if(parse_post(state)) {
        sprintf(state->buffer, "<p>Username:
            %s<p>\r\n<p>Email: %s<p>\r\n", FORMSpec[0].value,
            FORMSpec[1].value);
        state->length = strlen(state->buffer);
        state->substate++;
    }
    break;

case 4:
    strcpy(state->buffer, "<p>Go <a
        href=\"/\>home</a></body> </html>\r\n");
    state->length = strlen(state->buffer);
    state->substate++;
    break;
default:
    state->substate = 0;
    return 1;
}
}
return 0;
}

```

4.5.4 HTML Forms Using Zserver.lib

In this section, we will step through a sample program, `Samples\tcpip\http\form1.c`, that uses HTML forms. Through this step-by-step explanation, the method of using the functions in `zserver.lib` will become clear. (As of Dynamic C 8.50, you have the option of using the RabbitWeb server, with its easier-to-use interface and completely flexible ZHTML page layout capabilities. For more information, see the document titled "RabbitWeb, To Web-Enable Embedded Applications" available on our [website](#).)

Defining `FORM_ERROR_BUF` is required in order to use the HTML form functionality in `Zserver.lib`. The value represents the number of bytes that will be reserved in root memory for the buffer that will be used for form processing. This buffer must be large enough to hold the name and value for each variable, plus four bytes for each variable. Since we are building a small form, 256 bytes is sufficient.

```
#define FORM_ERROR_BUF 256
```

Since we will not be using the static resource table, we can define the following macro, to remove some code for handling this table from `Zserver`.

```
#define HTTP_NO_FLASHSPEC
```

These lines are part of the standard TCP/IP and MIME table configuration.

```
#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"
    SSPEC_MIMETABLE_START
        SSPEC_MIME(".html", "text/html")
    SSPEC_MIMETABLE_END
```

These are the declarations of the variables that will be included in the form.

```
int temphi;
int tempnow;
int templo;
float humidity;
char fail[21];
```

```
void main(void)
{
```

An array of type `FormVar` must be declared to hold information about the form variables. Be sure to allocate enough entries in the array to hold all of the variables that will go in the form. If more forms are needed, then more of these arrays can be allocated.

```
FormVar myform[5];
```

These variables will hold the indices in the TCP/IP servers' object list for the form and the form variables.

```
int var;
int form;
```

This array holds the possible values for the fail variable. The fail variable will be used to make a pulldown menu in the HTML form.

```
const char *const fail_options[] = {
    "Email",
    "Page",
    "Email and page",
    "Nothing"
};
```

These lines initialize the form variables.

```
temphi = 80;
tempnow = 72;
templo = 65;
humidity = 0.3;
strcpy(fail, "Page");
```

The next line adds a form to the dynamic resource table. The first parameter gives the name of the form. When a browser requests the page "myform.html" the HTML form is generated and presented to the browser. The second parameter gives the developer-declared array in which form information will be saved. The third parameter gives the number of entries in the `myform` array (this number should match the one given in the `myform` declaration above). The fourth parameter indicates that this form should only be accessible to the HTTP server, and not the FTP server. `SERVER_HTTP` should always be given for HTML forms. The return value is the index of the newly created form in the dynamic resource table.

```
form = sspec_addform("myform.html", myform, 5, SERVER_HTTP);
```

This line sets the title of the form. The first parameter is the form index (the return value of `sspec_addform()`), and the second parameter is the form title. This title will be displayed as the title of the HTML page and as a large heading in the HTML page.

```
sspec_setformtitle(form, "ACME Thermostat Settings");
```

The following line adds a variable to the resource table. It must be added to this table before being added to the form. The first parameter is the name to be given to the variable, the second is the address of the variable, the third is the type of variable (this can be `INT8`, `INT16`, `INT32`, `FLOAT32`, or `PTR16`), the fourth is a printf-style format specifier that indicates how the variable should be printed, and the fifth is the server for which this variable is accessible. The return value is the handle of the variable in the resource table.

```
var = sspec_addvariable("temphi", &temphi, INT16, "%d",  
SERVER_HTTP);
```

The following line adds a variable to a form. The first parameter is the index of the form to add the variable to (the return value of `sspec_addform()`), and the second parameter is the index of the variable (the return value of `sspec_addvariable()`). The return value is the index of the variable within the developer-declared `FormVar` array, `myform`.

```
var = sspec_addfv(form, var);
```

This function sets the name of a form variable that will be displayed in the first column of the form table. If this name is not set, it defaults to the name for the variable in the resource table ("temphi", in this case). The first parameter is the form in which the variable is located, the second parameter is the variable index within the form, and the third parameter is the name for the form variable.

```
sspec_setfvname(form, var, "High Temp");
```

This function sets the description of the form variable, which is displayed in the third column of the form table.

```
sspec_setfvdesc(form, var, "Maximum in temperature range  
(60 - 90 &deg;F)");
```

This function sets the length of the string representation of the form variable. In this case, the text box for the form variable in the HTML form will be 5 characters long. If the user enters a value longer than 5 characters, the extra characters will be ignored.

```
sspec_setfvlen(form, var, 5);
```

This function sets the range of values for the given form variable. The variable must be within the range of 60 to 90, inclusive, or an error will be generated when the form is submitted.


```
sspec_setfvrange(form, var, 60, 90);
```

This concludes setting up the first variable. The next five lines set up the second variable, which represents the current temperature.

```
var = sspec_addvariable("tempnow", &tempnow, INT16, "%d",  
    SERVER_HTTP);  
var = sspec_addfv(form, var);  
sspec_setfvname(form, var, "Current Temp");  
sspec_setfvdesc(form, var, "Current temperature in &deg;F");  
sspec_setfvlen(form, var, 5);
```

Since the value of the second variable should not be modifiable via the HTML form (by default variables are modifiable,) the following line is necessary and makes the given form variable read-only when the third parameter is 1. The variable will be displayed in the form table, but can not be modified within the form.

```
sspec_setfvreadonly(form, var, 1);
```

These lines set up the low temperature variable. It is set up in much the same way as the high temperature variable.

```
var = sspec_addvariable("templo", &templo, INT16, "%d",  
    SERVER_HTTP);  
var = sspec_addfv(form, var);  
sspec_setfvname(form, var, "Low Temp");  
sspec_setfvdesc(form, var, "Minimum in temperature range  
    (50 - 80 &deg;F)");  
sspec_setfvlen(form, var, 5);  
sspec_setfvrange(form, var, 50, 80);
```

This code begins setting up the string variable that specifies what to do in case of air conditioning failure. Note that the variable is of type PTR16, and that the address of the variable is not given to `sspec_addvariable()`, since the variable `fail` already represents an address.

```
var = sspec_addvariable("failure", fail, PTR16, "%s",  
    SERVER_HTTP);  
var = sspec_addfv(form, var);  
sspec_setfvname(form, var, "Failure Action");  
sspec_setfvdesc(form, var,  
    "Action to take in case of air-conditioning failure");  
sspec_setfvlen(form, var, 20);
```

This line associates an option list with a form variable. The third parameter gives the developer-defined option array, and the fourth parameter gives the length of the array. The form variable can now only take on values listed in the option list.

```
sspec_setfvoptlist(form, var, fail_options, 4);
```

This function sets the type of form element that is used to represent the variable. The default is HTML_FORM_TEXT, which is a standard text entry box. This line sets the type to HTML_FORM_PULLDOWN, which is a pull-down menu.

```
sspec_setfventrytype(form, var, HTML_FORM_PULLDOWN);
```

Finally, this code sets up the last variable. Note that it is a float, so FLOAT32 is given in the sspec_addvariable() call. The last function call is sspec_setfvfloatrange() instead of sspec_setfvrange(), since this is a floating point variable.

```
var = sspec_addvariable("humidity", &humidity, FLOAT32,
    "%.2f", SERVER_HTTP);
var = sspec_addfv(form, var);
sspec_setfvname(form, var, "Humidity");
sspec_setfvdesc(form, var, "Target humidity (between 0.0 and 1.0)");
sspec_setfvlen(form, var, 8);
sspec_setfvfloatrange(form, var, 0.0, 1.0);
```

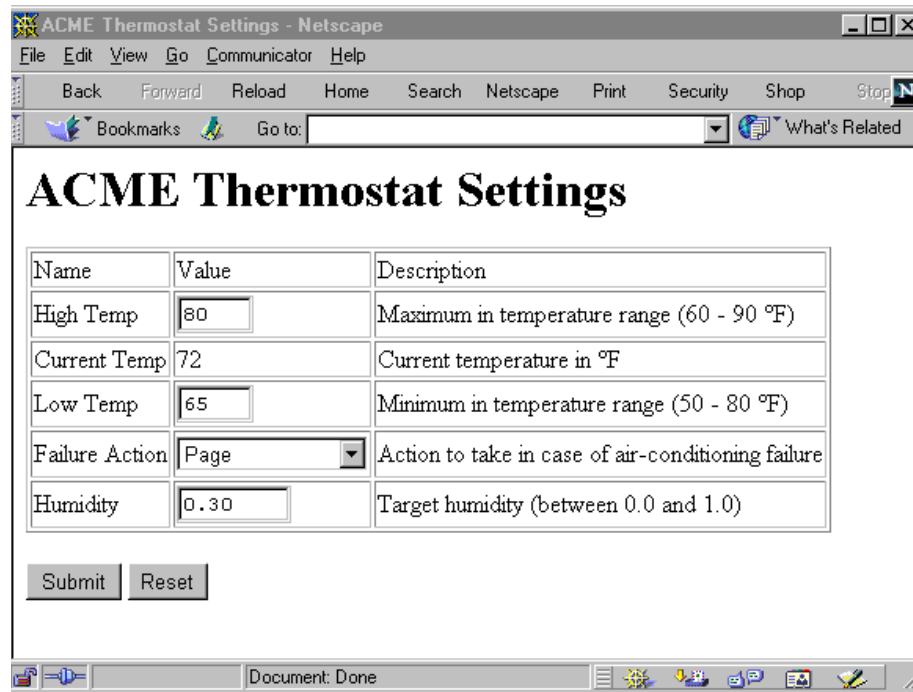
These calls create aliases in the dynamic resource table for the HTML form. That is, the same form can now be generated by requesting "index.html" or "/". Note that sspec_aliasspec() should be called after the form has already been set up. The aliasing is done by creating a new entry in the resource table and copying the original entry into the new entry. Note that aliasing can also be done for files and other types of server objects.

```
sspec_aliasspec(form, "index.html");
sspec_aliasspec(form, "/");
```

These lines complete the sample program. They initialize the TCP/IP stack and web server, and run the web server.

```
sock_init();
http_init();
while (1) {
    http_handler();
}
}
```

This is the form that is generated:



The screenshot shows a Netscape browser window titled "ACME Thermostat Settings - Netscape". The browser's menu bar includes "File", "Edit", "View", "Go", "Communicator", and "Help". The toolbar contains "Back", "Forward", "Reload", "Home", "Search", "Netscape", "Print", "Security", "Shop", and "Stop". The address bar shows "Go to:" followed by a dropdown menu and a "What's Related" link. The main content area displays the title "ACME Thermostat Settings" in a large, bold, serif font. Below the title is a table with three columns: "Name", "Value", and "Description". The table contains six rows of settings. At the bottom of the form are two buttons: "Submit" and "Reset". The browser's status bar at the bottom shows "Document: Done" and several icons.

Name	Value	Description
High Temp	<input type="text" value="80"/>	Maximum in temperature range (60 - 90 °F)
Current Temp	72	Current temperature in °F
Low Temp	<input type="text" value="65"/>	Minimum in temperature range (50 - 80 °F)
Failure Action	<input type="text" value="Page"/> ▾	Action to take in case of air-conditioning failure
Humidity	<input type="text" value="0.30"/>	Target humidity (between 0.0 and 1.0)

4.6 HTTP File Upload

This section describes the HTTP file upload feature available starting with Dynamic C 8.50. The enhanced CGI capabilities of this version of Dynamic C allow files of unlimited size to be uploaded using a web interface. It has always been possible to upload files using FTP; however, it is usually more convenient to use a browser-based upload.

4.6.1 What is a CGI Function and Why is It Useful?

The HTTP library provided with Dynamic C allows the association of C functions with web page URLs. When the user, via their web browser, retrieves a specified resource, the C function may be called from the HTTP server. Such a function is called a Common Gateway Interface (CGI) function, and it is responsible for generating a response to the user's request.

The advantage of using a CGI is that it can generate web page content on-the-fly, and cause the browser to display or do anything that it is capable of. In addition, the CGI is able to read data that was sent by the browser.

Previous to this release of Dynamic C, the CGI was limited to handling relatively small amounts of data sent from the browser. This is satisfactory for processing simple forms, but does not allow large data sets to be uploaded. This release of Dynamic C supports upload of one or more files from the browser. The files can be of unlimited size. In conjunction with the latest Zserver (resource manager) enhancements available in Dynamic C 8.50, the uploaded files may be stored in the FS2 or FAT file systems, or even processed dynamically.

The new CGI file upload facility enables a range of convenient firmware features. Possibilities include:

- Remote firmware updates.
- Web page content updates (i.e. "publishing").
- Executable (interpreter) scripts.
- Remote hardware updates (if using an FPGA or other configurable logic device).
- Firmware configuration.

NOTE: Throughout this document the FAT file system is the destination for the uploaded file. The FAT uses onboard serial flash and is available as a module which may be purchased separately. See www.zworld.com.

4.6.2 How Do I Use the New CGI Facility?

There are a number of steps, some of which will be familiar to users of CGIs in previous releases. They are listed here and described in more detail in the following pages. The steps, if coding from scratch, are:

1. #use "dcrtcp.lib", and specify network configuration options.
2. #use <filesystem(s) of choice>, and specify the file system configuration.
3. #define USE_HTTP_UPLOAD
4. #use "http.lib"
5. Create an initial web page with a form asking for the file(s) to be uploaded. The main requirement is that you specify `enctype="multipart/form-data"` inside the `<FORM>` tag(s).
6. Write a CGI function (if not using the default one provided).
7. Create an initial resource table containing at least an entry for each of the above two resources (the web page and the CGI).
8. Create a list of content type mappings, i.e., the MIME table.
9. Create rules which limit the upload facility to select user groups.
10. Create a set of user IDs
11. In the main program, call `http_handler()` in a loop.

Step 1: Specify Network Configuration

To make use of HTTP upload, you need to perform the usual inclusion and configuration of the networking library, `dcrtcp.lib`. At its simplest, it is two lines of code at the top of your main program:

```
#define TCPCONFIG 1
#use "dcrtcp.lib"
```

This specifies that the default TCP (networking) configuration is to be used. If you want to change the default networking configuration, first read the comments at the top of `tcp_config.lib`.

HTTP upload usually requires at least two additional libraries to be included: a file system library, and `http.lib` itself. A file system is required, otherwise the uploaded file has nowhere to go (although you can write a CGI which processes the file as it is uploaded, in which case you do not need to store it permanently, and thus you do not need to include a file system; the following discussion assumes that you are using a file system).

Steps 2, 3 and 4: Specify File system and Web Server

You need to include the file system library (or libraries) *before* including `http.lib`. This is because the HTTP library needs to know about the filesystem(s) it is going to support. In addition, you need to tell the HTTP library to use the upload facility. For example, if you want to use the FAT file system, then you would write the following:

```
#define TCPCONFIG 1
#use "dcrtcp.lib"
#use "fat.lib"           // Step 2: the filesystem
#define USE_HTTP_UPLOAD // Step 3: enable upload feature
#use "http.lib"         // Step 4: HTTP server code
```

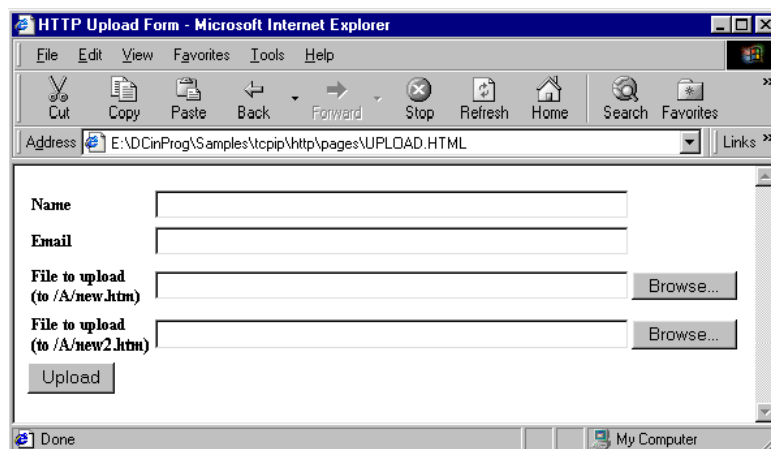
The order of the above statements is important. A possible exception is that the order of `dcrtcp.lib` and `fat.lib` may be interchanged, since these libraries are independent. However, it is recommended you use the given ordering since future releases of the FAT may be able to use networking services.

Step 5: Create a Web Page

When using HTTP upload, there needs to be a way to prompt the user (web browser) to enter a file name to upload. This is done by using an HTML form. The form specifies input fields that may be filled out by the user, and one or more “submit” buttons that the user presses to start the upload process.

If you have an existing web-based application to which you want to add a file upload facility, you probably already have a web page with a form on it; in this case, you can add an extra input field to an existing form on that page, or create a new form on the same page. You may already have a CGI function that processes the results of the form submission. This will need to be rewritten to process data that is not URL encoded.

If you are creating a new application, you need to construct an initial page to contain the necessary form elements. As a starting point you can use the sample page in `samples\tcpip\http\pages\upload.html`. Click on `upload.html` and the browser will display something like this:



The construction of this page is outlined below, but it has been simplified and reformatted slightly. A blow-by-blow description of each line is added in *italics*.

```
<html>
```

This introduces the page as an HTML document.

```
<head><title>HTTP Upload Form</title></head>
```

This (“HTTP Upload Form”) gets displayed at the top of the browser window. You can change this to whatever is appropriate for describing the overall purpose of this page.

```
<body>
```

Introduce the main content of this page.

```
<FORM ACTION="upload.cgi" METHOD="POST" enctype="multipart/form-data">
```

Start a form definition. The parameters are
action="upload.cgi": this refers to the CGI function that will process the results of the form submission. This is a URL name, which is mapped to a C function on the server.
method=post: this is required, since a post-type request must be sent to the server.
enctype="multipart/form-data": this is also required, and is the part that is different from the old style of processing. The old style did not specify an encoding type, thus the default of “URL encoded” was used.

```
<TABLE BORDER=0 CELLSPACING=2 CELLPADDING=1>
```

For neatness of screen layout, we put everything in an HTML table. The following <TR>...</TR> sections delimit each row of the table, and the data for each cell is delimited by <TD>...</TD>.

```
<TR>
```

```
  <TD WIDTH=130 ALIGN=RIGHT><B>Name</B></TD>
```

```
  <TD WIDTH=500><INPUT TYPE="TEXT" NAME="user_name" SIZE=50></TD>
```

This is the first input field. It is not a file to upload, but it is information that the server may nevertheless be interested in. This shows that not every form field needs to be a file to upload. The order is important. Browsers will send back the form fields in the same order that they are defined in the HTML, however it is probably best not to rely on this if you can help it.

```
</TR>
```

```
<TR>
```

```
  <TD ALIGN=RIGHT><B>File to upload<BR>(to /A/new.htm)</B></TD>
```

```
  <TD><INPUT TYPE="FILE" NAME="/A/new.htm" SIZE=50></TD>
```

This is the file-to-upload input field. The browser displays this as a text input field, with an additional “browse” button so that the user can easily navigate his local filesystem to find the appropriate file. The critical distinction is that it contains a type=file parameter (as opposed to, for example, type=text in the previous field). The name="/A/new.htm" parameter specifies the name of the input field, not the name of the file on the user’s system! As it happens, this looks like a file name, and indeed the server may use it as the name of a local file, but this is a convention only. The size=50 parameter specifies the number of characters that the browser will display for file name selection.

```
<TR>
```

```
</TABLE>
```

```
<INPUT TYPE="SUBMIT" VALUE="Upload">
```

It is necessary to supply a type=submit form element. The user presses this button to start to post (upload) process. Note that this is another input field, however if you leave out the name= parameter (as in this example) then the browser will not send the value of this button back with the form submission. If there is only one submit button, then there is no need to name it.

```
</FORM></body></html>
```

Close and complete the form, body, and entire page.

If you have an existing application, you can take out the relevant parts of the above, and insert them in your existing web page. The relevant parts are the `enctype="multipart/form-data"` parameter in the `<FORM>` element, and the `<INPUT type=file>` element.

If you have an existing application that processes the form data submission, you will need to rewrite the CGI function that handles the submitted data. This is because the `enctype` parameter changes the syntax that the browser uses to encode the data. In short, you will need to rewrite the CGI as a “new-style” CGI as described in Step 6: Writing a CGI Function.

Having created the HTML file with the upload form, it is necessary to import it into your main program, so that the HTTP server can present it to the user. This can be done using `#import`, or you can write it directly to the filesystem (although, initially at least, this presents a chicken-and-egg type problem since you might not have established an upload procedure in the first place!)

4.6.2.1 Step 6: Writing a CGI Function

The CGI function is responsible for processing the form submission data as it comes in from the client (browser). In addition, it generally needs to write some sort of response back to the client indicating whether or not the submission was acceptable.

If you start reading the following, and start feeling somewhat overwhelmed, please be aware that there is a default CGI function in the HTTP library that is very useful. The default CGI, called `http_defaultCGI()`, automatically saves uploaded files into the filesystem. If that is all you need to do, then you do not need to fully comprehend this section on first reading.

Note that all of this section is describing *new-style* CGIs. Old-style CGIs are covered in Section 4.5.3.

CGI Syntax

All CGI functions are C functions with the following prototype:

```
int my_CGI (HttpState * s);
```

The `HttpState` parameter is a pointer to the internal state variables of the HTTP server instance that is handling the current request. You can have one or more server instances. If there is more than one, the same CGI may be invoked at the same time for more than one client (if both happen to press the submit button at about the same time). Thus, it is important to write the CGI function so that it is re-entrant. This basically means that the function should not update global or static variables. The CGI should not attempt to modify directly any of the fields in the `HttpState` structure, otherwise the server may become inoperable.

API Functions

The HTTP library provides a set of API functions that can be called safely from the CGI. The list of safe functions is in the index under “Function Reference, CGI.”

It is unwise to make direct calls to TCP/IP functions, especially functions that may not return for a long time such as `sock_read()`.

How to Transfer Form Submission Data

To understand how to write a CGI function, it is necessary to have some understanding of the protocol used to transfer the form submission data. Since the data can consist of one or more files and/or form fields, there needs to be a way of separating them within the one, sequential, stream of data that is sent by the client.

The way this is done is that the client specifies a unique string that separates each item of data. The following text is a dump of the actual data sent by a client (with some irrelevant details omitted, and with comments added in *italics*):

```
POST /upload.cgi HTTP/1.1
```

This indicates that it is POSTed form data, and the target handler is upload.cgi.

```
Content-length: 277
```

This gives the total number of bytes of data following the initial header.

```
Content-Type: multipart/form-data; boundary=3vAL1QsFOUg2GsY3p6n3YQ
```

The multipart/form-data type indicates that this is a multipart form data submission. The boundary parameter specifies a unique character sequence that separates each part. The boundary is deliberately chosen as a long, random, string of characters so that it is unlikely to be confused with the actual data content.

The above blank line is significant; it indicates the end of the initial header lines, and the start of data.

```
--3vAL1QsFOUg2GsY3p6n3YQ
```

This is the first boundary. Boundary strings are always prefixed by an additional -- sequence. The following lines are header lines for the individual part. The actual data follows the first empty line.

```
Content-Disposition: form-data; name="/A/new.htm"; filename="test.txt"
```

The Content-Disposition header indicates the presentation of the data. The only type which is relevant is "form-data". The name= parameter indicates the field name (which was originally part of the name= parameter of the <input> element). The filename= parameter is only set if this is an uploaded file. It gives the name of the file on the remote (client) side. This is not usually relevant to the server. The name of the file as it is stored on the server is not specified (since the browser does not know it or have control over where the file is stored). We are using the convention that the field name indicates the local file name, but this is just a convention!

```
Content-Type: text/plain
```

Content-Type indicates the type of information. The default is plain (i.e. ascii) text, however it could also be set to image/gif for a GIF file, text/html for HTML etc. The following blank line indicates the end of headers for this part.

```
test file contents, first line
```

This is the actual file or form field content.

```
--3vAL1QsFOUg2GsY3p6n3YQ
```

The boundary string terminates the data for the previous part. Headers for the next part immediately follow.

```
Content-Disposition: form-data; name="submit"
```

This is form field data, in this case the submit button itself.

```
upload
```

```
--3vAL1QsFOUg2GsY3p6n3YQ--
```

The boundary terminates the previous form field. Since this is the last boundary, it also has a trailing --.

When writing the CGI, you do not have to worry about parsing the headers and boundary separators. This is already done by the HTTP server. However, you do need to be aware of the stream-oriented nature of the incoming data. The HTTP server separates out the parts (and parses the headers). As it does this, it calls the defined CGI with the data for each section.

Action Codes Received by a CGI Function

The CGI is called in a number of different contexts. It determines the context by calling the `http_getAction()` function. The return value of `http_getAction()` indicates the reason that the CGI is being called by the HTTP server.

For a given upload, the CGI is called with a typical sequence of action codes. The first code is `CGI_START` (for the start of a new part), `CGI_DATA` (for each chunk of data in that part), then `CGI_END` (for the end of the part). Thus, the typical sequence for a single part is

```
CGI_START, CGI_DATA, CGI_DATA, . . . . CGI_DATA, CGI_END
```

Finally, at the end of all the parts, the action code is set to `CGI_EOF`.

Most CGIs should also handle a special action code called `CGI_ABORT`. This code only occurs if the upload is terminated early by a network problem (or by the user pressing the browser's cancel or stop button).

Let's examine a simple CGI that handles these five action codes. This is the minimum requirement; however, there are some additional codes that may be used by more advanced CGIs. The switch statement ignores action codes that are not listed. This is deliberate, since any other action codes may be safely ignored.

```
int my_CGI (HttpState * s)
{
    switch (http_getAction(s)) {
        case CGI_START:
            break;
        case CGI_DATA:
            break;
        case CGI_END:
            break;
        case CGI_EOF:
            break;
        case CGI_ABORT:
            break;
    }
    return 0;
}
```

The above code is a skeleton that does nothing! In other words, all incoming data is sent to the bit-bucket. It is ready to fill out with more useful actions. To avoid repeating the code, we just take each case condition, and fill in the details.

Action Code CGI_START

When the action code CGI_START is received, all of the part headers have been read, so the server knows everything relevant about the data that follows. The CGI can access this information using several of the HTTP API functions. The most important information is the field name on the form, from the <INPUT NAME= "fieldname"> element in the HTML form:

```
case CGI_START:
    if (http_getField(s)[0] == '/') {
        printf("Found a file to upload!\n");
        ...
    }
    break;
```

http_getField() looks at the first character of the field name to see if it is a slash character. We are using the convention that if the field starts with a slash, it is the name of a local file to be overwritten with the following data. Note that the field names are controlled by the server, via the NAME= parameters in the INPUT fields. We can choose any naming convention desired; in this case, using an initial slash seems to make sense for file destinations.

Now let's fill in what happens when there is a file to save. In most cases, when writing or reading a file, it is necessary to "open" the file. When a file is open, it can be read and/or written. Finally, it is closed. All this implies that some sort of state needs to be maintained so that we can refer to the correct open file. It would be very easy if all the data was presented at once to the CGI, so that it could open, write, and close the file in one fell swoop. Unfortunately, that cannot happen since the data is not yet available on the CGI_START call. The CGI has no choice than to return to the HTTP server after doing whatever it can in the CGI_START state.

The solution to this problem is that the CGI opens the file on the CGI_START call, and stores the open file handle somewhere where it can be retrieved on the next (CGI_DATA or CGI_END) call. The recommended method for accomplishing this is to save the handle back with the server. You can use the http_setCond() and http_getCond() functions to do this.

The HTTP server maintains a set of so-called "cond" variables for each CGI instance. Your application decides how many cond variables there are by defining the HTTP_MAX_COND macro, which defaults to 4. Each cond variable is a 16-bit integer.

There is also a single integer variable accessed using http_getState() and http_setState().

Expanding on the above, let's add opening of the file:

```
#define COND_HANDLE 0 // cond variable for storing the handle.
case CGI_START:
    if (http_getField(s)[0] == '/') {
        printf("Found a file to upload!\n");
        http_setCond(s, COND_HANDLE,
            sspec_open(http_getField(s), http_getContext(s),
                O_WRITE|O_CREAT|O_TRUNC, 0));
        if (http_getCond(s, COND_HANDLE) < 0)
            http_skipCGI();
    }
    else
        http_skipCGI(s);
    break;
```

The `sspec_open()` function opens the file (whose name is in the field name) with write access. The `http_getcontext()` function returns a server context structure which is required for the `sspec_open()` call. The context structure contains some details, such as the current user ID, but the details are usually not relevant to the CGI function itself. The file is created if it does not exist, and it is initially truncated if it already exists. The return value from `sspec_open()` is stored in the cond variable `COND_HANDLE`, which is a macro we defined to zero so we wouldn't have to remember hard-coded numbers. The return value is either negative (if there was an error), or not negative in which case it is a valid file handle. We check the cond variable just set, to make sure it has a valid value.

The `else` clause is added so that if the part is not a file to upload the rest of the data for this part is ignored. This is convenient, since we don't want to get called with `CGI_DATA` or `CGI_END` if this is not a file. If `http_skipCGI()` is called, then the next action code will be either `CGI_START` (if there is another part), or `CGI_EOF` (if there were none). Note that we are also calling `http_skipCGI()` in the case that the file could not be opened.

Action Code `CGI_DATA`

Let's now turn to saving the data. For this, we make use of the `CGI_DATA` action code:

```
int handle;
...
case CGI_DATA:
    handle = http_getCond(s, COND_HANDLE);
    sspec_write(handle, http_getData(s), http_getDataLength(s));
    break;
```

First, the open file handle is retrieved from the cond variable. This works because the HTTP server does not touch these variables between calls. The only time the server changes the cond variables is at the start of a completely new form submission, in which case they are usually set to zero. But don't depend on them being zero, since a form submission can sometimes contain syntax that sets them to non-default values. You can rely on `http_getState()` returning zero on the very first call; thereafter, it is not touched, but can be manipulated by the CGI calling the function `http_setState()`.

Having retrieved the open file handle (you didn't save it in a static variable, did you?) it is used in the `sspec_write()` call. `http_getData()` returns the available data, and `http_getDataLength()` returns its length (in bytes). The maximum length that `http_getDataLength()` will return is `HTTP_MAXBUFFER`, which is a macro controlled by the application (defaulting to 256). Often, the available data length will be less than this, even in the middle of a long file.

Note that the return code from `sspec_write()` is not checked. This is a shortcoming that we fix later, since the solution can be slightly complex. For now, we just hope that it works.

Action Code CGI_END

The next thing to consider is closing the file when the upload is complete. For this, we make use of the `CGI_END` action code:

```
case CGI_END:
    handle = http_getCond(s, COND_HANDLE);
    sspec_close(handle);
    break;
```

This is quite simple. We simply retrieve the handle, and close it.

Response to the Client: Redirection

Finally, we have to consider what to do at the end of all parts (`CGI_EOF`), or if the connection was cancelled (`CGI_ABORT`). You may recall that the CGI has two responsibilities: one is to process the incoming data, and the other is to write some results back to the client. We have already done the former, it is only left to do the latter.

Writing results to the client means we have to generate the proper HTTP response, including all the necessary headers and web page content. The CGI can do this itself, by putting strings in the buffer provided by the `http_getData()` call. Alternatively, the CGI can simply redirect back to another local (or even remote) web page and not bother writing anything itself.

If the CGI wants to generate the response itself, then this has the advantage of being slightly more efficient, but the disadvantage of requiring more code in the CGI. Usually, the application already has some sort of web page that can display the necessary results. This is often an "SSI" page (that is, dynamically generated using a specialized function) or may be just a static page (for example, `/index.html`).

Action Code CGI_EOF

Since referring to another web page is easiest, it is shown first:

```
case CGI_EOF:
    cgi_redirectto(s, "/index.html");
    break;
```

The `cgi_redirectto()` function tells the HTTP server to stop calling this CGI function, and tell the client to retrieve its next web page from the specified location (in this case, the `index.html` page on the current server). The onus is on the client (browser) to go and get that page. It will come straight back to this server, but the CGI does not have to worry about it. Easy!

In a similar vein, you can use the `http_switchCGI()` function. Again, the current CGI does not have to generate a response. The difference is that the HTTP server goes straight to the specified web page and presents it to the client on the same connection (rather than requiring the client to come back to the server with a new request).

`http_switchCGI()` can transfer control to any local web page, as if the client had directly requested that resource. If the resource happens to be another new-style CGI (like the one we are describing), then it gets control with the current action code, which will usually be `CGI_EOF`. Otherwise, the resource is processed as if it was directly retrieved by the client, by name. Note: the current CGI must not have written anything back to the client, otherwise the data will not be intelligible to the client). Here is an example:

```
case CGI_EOF:
    http_switchCGI(s, "/index.html");
    break;
```

As you can see, it is very similar to the `cgi_redirectto()` case.

Action Code `CGI_ABORT`

The conventions for having the CGI generate its own response back to the client are covered in the next section, titled, [Writing Responses to the Client from a CGI Function](#). First, we look at the proper handling of a `CGI_ABORT` action code. This code means that the connection has been lost and there is no point in handling any more incoming data or generating any response. Thus, processing of `CGI_ABORT` is necessarily limited to cleaning up any open files or other resources:

```
case CGI_ABORT:
    handle = http_getCond(s, COND_HANDLE);
    sspec_close(handle);
    break;
```

In this example, we simply close the handle, possibly leaving the file with partially written contents. It is important to do this, since if the handle is left open, then that handle is lost forever (or until the next reboot). The `CGI_ABORT` code can happen at any time, so the CGI must handle it if it ever uses “leakable” resources.

If you are alert, you noticed that `CGI_ABORT` may be called when there is no open handle. We must guard against the possibility of trying to close an “invalid” handle, since it may happen to belong to another active CGI. We can do this by ensuring the value in the `cond` variable is “-1” if the handle is not open.

Minimum Required Functionality of CGI

All the above code is pulled together, with the proper tests and comments on the additional code:

```
#define COND_HANDLE 0 // cond variable for storing the handle.
int my_CGI(HttpState * s){
    int handle;

    // Following block ensures that the first time (http_getState() is zero) we set the handle to -1.
    if (http_getState(s) == 0) {
        http_setState(s, 1);
        http_setCond(s, COND_HANDLE, -1);
    }
    switch(http_getAction(s)) {
        case CGI_START:
            if (http_getField(s)[0] == '/') {
                printf("Found a file to upload!\n");
                http_setCond(s, COND_HANDLE,
                    sspec_open(http_getField(s), http_getContext(s),
                        O_WRITE|O_CREAT|O_TRUNC, 0));
                if (http_getCond(s, COND_HANDLE) < 0)
                    http_skipCGI();
            }
            else
                http_skipCGI(s);
            break;

        case CGI_DATA:
            handle = http_getCond(s, COND_HANDLE);
            sspec_write(handle, http_getData(s),
                http_getDataLength(s));
            break;

        case CGI_END:
            handle = http_getCond(s, COND_HANDLE);
            sspec_close(handle);

    // The following statement ensures that the handle is set back to -1 when we know it is closed.
            http_setCond(s, COND_HANDLE, -1);
            break;

        case CGI_EOF:
            http_switchCGI(s, "/index.html");
            break;

        case CGI_ABORT:
            handle = http_getCond(s, COND_HANDLE);

    // The following test is added so we don't try to close the handle if it is already closed.
            if (handle >= 0)
                sspec_close(handle);
            break;
    }
    return 0;
}
```

What Happens if the Write Fails?

There is still one point to cover. That is, the `sspec_write()` call is not guaranteed to swallow all of the data that it was told to write. In fact, `sspec_write()` may completely fail (for example, if the file system runs out of space).

First, let's handle the case where `sspec_write()` returns an error, that is, its return code is negative. In this case, we probably want to return an error indication to the client. This can be done using the `http_switchCGI()` or `cgi_redirectto()` functions. A special page will need to be created for this purpose. If this page is called `"/upld_err.html"`, then the following code could be used:

```
case CGI_DATA:
    handle = http_getCond(s, COND_HANDLE);
    if (sspec_write(handle, http_getData(s),
        http_getDataLength(s)) < 0)
    {
        sspec_close(handle);
        http_switchCGI(s, "/upld_err.html");
    }
    break;
```

In the case of an error, the handle is closed, then the HTTP server presents the `upld_err.html` page to the client. The current CGI is abandoned, including any pending data that is still incoming. This is why the handle is explicitly closed (since `upld_err.html` probably doesn't know anything about it!). Naturally, `upld_err.html` is a web page that tells the user that something went wrong. In practice, this would usually be an SSI rather than a static web page, since you would probably want to give the user different feedback depending on the exact type of error.

The final consideration is what to do if `sspec_write()` can only write some (or perhaps none) of the data it was given. The normal course of action is to just retry later, with the data that was not written. You could just sit in a loop in the CGI function waiting for the data to be written. This may be satisfactory in some cases, but often this will unnecessarily reduce system performance (since nothing else will get a chance to run except interrupts). It is preferable to return to the HTTP server, which in turn can return to the application before coming back into the CGI.

CGI Return Codes

This is where the CGI return code becomes important. Up to now, the return code has always been zero, which means "continue as usual." (However, some of the APIs such as `http_abortCGI()` override this.)

There are several other legitimate values for the return code:

- `CGI_MORE`: Call back again when free space in transmit buffer.
- `CGI_DONE`: CGI has finished writing data to the client.
- `CGI_SEND`: Send the data (null term string) in the main buffer.
- `CGI_SEND_DONE`: combination of the above two.

Action Code CGI_CONTINUE

In the case we are discussing, the CGI_MORE return code is used. This tells the server that the CGI function is busy trying to do something, but it could not complete the task. It wants to be called back again, but without any new incoming data.

Thus, if the CGI function returns CGI_MORE, the HTTP server will eventually come back with a special action code, which has not been mentioned yet, called CGI_CONTINUE. The CGI needs to respond to this code so that it can continue doing what it was trying before. This implies that the CGI will need to remember at least a bit of information (like how many bytes of the total it successfully wrote). For this, it can use the “state” and “cond” variables.

The following code shows the relevant sections for following this protocol:

```
int len, newlen;
#define COND_LEN 1
case CGI_DATA:
    handle = http_getCond(s, COND_HANDLE);
    len = sspec_write(handle, http_getData(s),
        http_getDataLength(s));

    if (len < 0) { //permanent error
        sspec_close(handle);
        http_switchCGI(s, "/upld_err.html");
    }
    else if (len < http_getDataLength(s)) //no error, but not all written
    {
        http_setCond(s, COND_LEN, len); //save place in file
        return CGI_MORE; //tell server we're not done
    }
    break;

case CGI_CONTINUE: //CGI_MORE returned last time
    handle = http_getCond(s, COND_HANDLE); //get file handle
    len = http_getCond(s, COND_LEN); //get place in file

    // Try writing the part that wasn't written.
    newlen = sspec_write(handle, http_getData(s)+len,
        http_getDataLength(s)-len);
    if (newlen < 0) { //permanent error when retrying.
        sspec_close(handle);
        http_switchCGI(s, "/upld_err.html");
    }
    else { //sum the total written count
        len += newlen;
        if (len < http_getDataLength(s)) { //still haven't written all
            http_setCond(s, COND_LEN, len); //save new place
            return CGI_MORE; //tell server we're not done
        }
    }
    break;
```

The important point is that when `CGI_CONTINUE` is the action code, the CGI retries the failed part of the previous operation, then tests whether it is complete. On completion, the usual “0” return code is returned, otherwise the CGI keeps returning `CGI_MORE` until the operation either completes or permanently fails. (The above code does not show the CGI returning zero. Look at the code in the default handler, `http_defaultCGI()`, to see this being done.)

You may notice the repetition of parts of this code, for example the calls to `http_switch CGI()`. This is for clarity; you can condense some of this by factoring out the common parts.

The CGI remembers where it was up to by using another cond variable, `COND_LEN`. This is all that is required, since the contents of `http_getData()` and its length are guaranteed not to be changed on the next call, when the CGI returns `CGI_MORE`.

Writing Responses to the Client from a CGI Function

A CGI function is able to generate all or part of the response to the client. To do this, it has to follow the HTTP specification. That is, it must write the response headers, plus the HTML content. The HTTP headers must be the first thing written. At a minimum, the header lines look like the following:

```
HTTP/1.0 200 OK
Date: Sun, 20 Jan 1980 23:27:10 GMT
Content-Type: text/html
```

NOTE: Each line must be terminated with a CRLF (that is, “\r\n”), and there must be a blank line after the last header. The date string can be constructed using the `http_date_str()` function.

You can create the headers in one hit using the following code:

```
char date[30];
sprintf(http_getData(s),
        "HTTP/1.0 200 OK\r\nDate: %s\r\nContent-Type: text/html\r\n\r\n",
        http_date_str(date));
```

Then send it to the client by returning `CGI_SEND` straight away. `CGI_SEND` tells the HTTP server that the CGI function has put a null-terminated string in the `http_getData()` buffer, and that the server should not call the CGI again until the string has been sent.

This is the most convenient way of sending relatively small amounts of data at a time. It relies on the fact that the CGI is allowed to write to the buffer returned by `http_getData()`. Since `http_getData()` is used to pass incoming data to the CGI, it is important to ensure that the *incoming data has been fully processed* before writing over that buffer. In addition, the buffer’s length is `HTTP_MAXBUFFER` which limits the size of the string (including the null terminator).

The CGI can return `CGI_SEND` for any action code (except `CGI_ABORT`). When the action code is `CGI_EOF`, there is no more incoming data, so strings can be written back to the client indefinitely; the server keeps calling the CGI at `CGI_EOF`. When the CGI has finished generating all the content, it must return `CGI_DONE`.

When the server gets the `CGI_DONE` return code, it closes the client connection normally, and ceases calling the CGI.

If the CGI has one more thing to write before it is “done,” it can return `CGI_SEND_DONE` which combines the `CGI_SEND` and `CGI_DONE` return codes. This can simplify the CGI if it does not have to do much when it first gets the `CGI_EOF` action code.

Using `CGI_SEND` return code has some limitations. In particular, only a limited size of string may be sent to the client on any one call. Also, a null character cannot be sent to the client because the null is interpreted as the end of the string. The null character problem is not usually important, since nulls are rarely (if ever) sent in an HTML document. The length limitation is more important, since some HTML constructs can be very verbose.

The `http_write()` function is designed to overcome these limitations. `http_write()` writes data from an arbitrary buffer (with a higher length limit on any one call), and returns either zero meaning that all data was successfully queued, or it may return `CGI_MORE` if it could not write the data. Either all or none of the data will be written, respectively. In the case that none was written, the CGI returns the `CGI_MORE` return code to the HTTP server. The CGI will then be called back with an action code of `CGI_CONTINUE`, where it should retry the failed `http_write()` call.

If `http_write()` returns zero, it can be called again immediately with more data, or the CGI can return zero to the HTTP server. Otherwise, the CGI function will generally need to remember what it was up to, and retry the `http_write()` on the next call. The following code illustrates use of `http_write()`:

```
static const char * a_very_long_html_fragment = "...."; //512 bytes
case CGI_END:
    return http_write(s, a_very_long_html_fragment,
        strlen(a_very_long_html_fragment));
case CGI_CONTINUE:
    if (was_writing_that_long_fragment)
        return http_write(s, a_very_long_html_fragment,
            strlen(a_very_long_html_fragment));
    break;
```

The details of determining which write was in progress have been glossed over. Basically, you would have to use a cond variable to keep track of which `http_write()` was in progress, if there is more than one possibility.

There is a limit to the amount of data that `http_write()` can possibly write on any given call. This limit is set by the HTTP server socket transmit buffer size. This buffer size is given by `TCP_BUF_SIZE/2`. The transmit buffer is usually at least 1024 bytes, which is considerably larger than the limitation when using the `CGI_SEND` return code (typically 255 bytes). If you try exceeding that limit, `http_write()` will never succeed.

Step 7: Creating the Resource Tables

Web browsers use URLs, which are specially formatted strings, to refer to resources (web pages) on the server. For example, a user may enter `http://rabbit_server/admin/upload.html` to retrieve the `/admin/upload.html` resource from the HTTP server on “rabbit_server.”

When the server receives such a request, it needs to look up the name, open the resource that it refers to, and send the contents back to the client.

CGI functions are no different from other resources, as far as the client is concerned. The server, of course, does entirely different things. The server needs to have a lookup table defined, which translates URLs into the appropriate local type of resource. This is the function of the “resource table,” which is also known as the “flashspec” or “ramspec” table in Dynamic C parlance.

The static resource table is a statically defined, constant, table. The dynamic resource table is generated at runtime. Both types can be used in the same program, with dynamic entries overriding static entries with the same URL.

With this release of Dynamic C, there is no need to put anything in either of these tables, provided that a filesystem (FAT or FS2) is used. However, it is convenient to have at least a few entries in the dynamic table, and it is mandatory to have entries in either or both the static and dynamic tables if CGI functions are used.

When using the HTTP upload facility, you will need at least one CGI function to be defined, and probably another entry for the initial form. The resource table may be defined as follows:

```
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_CGI("upload.cgi", my_CGI)
SSPEC_RESOURCETABLE_END
```

This defines a static resource table with two entries. The first is a static web page for the form (`index.html`) and the second points to the CGI that will be used to process the uploaded data. Important: use `SSPEC_RESOURCE_CGI`, not `SSPEC_RESOURCE_FUNCTION` - this defines the CGI as new-style. `SSPEC_RESOURCE_XMEMFILE` specifies a file that has been imported in the server’s flash memory using the `#ximport` directive. For example,

```
#ximport "samples/tcpip/http/pages/upload.html"    index_html
```

`index_html` is a placeholder (a long int) for the start of the file. This is mentioned in the resource table entry so that the server knows where to get it.

The second entry above specifies a “new-style” CGI function, which has been the subject of the preceding sections. You must use the `SSPEC_RESOURCE_CGI` macro to specify this type of CGI. The URL (string) parameter is whatever is mentioned in the `<form action=...>` parameter of the initial web page. The other parameter is the function pointer to the CGI that will process the upload.

If you do not wish to write a CGI just for handling file uploads, you could specify `http_defaultCGI()` as the CGI function.

Step 8: Create List of Content Type Mappings

The HTTP server needs to recognize different file formats. This is done using file extensions and MIME types. The server shares this information with the browser in its header. In this way, the browser knows how to handle the file.

The following code creates a table that maps file extensions to the appropriate MIME type.

```
SSPEC_MIMETABLE_START
    SSPEC_MIME(".htm", "text/html"),
    SSPEC_MIME(".html", "text/html"),
    SSPEC_MIME(".gif", "image/gif"),
    SSPEC_MIME(".cgi", "")
SSPEC_MIMETABLE_END
```

This method of creating the MIME type mapping table is new with Dynamic C version 8.5.

Step 9: Rule Creation

There must be rules to limit the upload facility to select user groups. This access control adds security to the system by disallowing unauthorized tampering.

This is done by assigning a unique user (or user group) the privilege of uploading new files. All other users will be permitted only read access. To do this, there are several things that need to be coordinated. First, the user(s) need to be created and assigned the correct group bit (which defines the upload privilege). Then, the CGI and the file system need to be protected so that only the privileged group can use the CGI, and only the privileged group can write to a defined subset of the file system.

Let's take this step-by-step. In the main program, define a group bit to represent the privileged user(s):

```
#define ADMIN_GROUP    0x0002
```

Groups are assigned one bit out of 16. In this case, we select bit 1. (Bit 0, or 0x0001, will be used for all other users).

Next, augment the resource table so that the CGI is accessible only to users in ADMIN_GROUP:

```
SSPEC_RESOURCETABLE_START
    SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
    SSPEC_RESOURCE_P_CGI("upload.cgi", my_CGI,
        "newPages", ADMIN_GROUP, 0x0000,
        SERVER_HTTP, SERVER_AUTH_BASIC)
SSPEC_RESOURCETABLE_END
```

The SSPEC_RESOURCE_P_CGI is a macro that allows specification of access control parameters. After the usual URL string and function pointer, the next parameters are:

- "newPages" - this is the so-called "realm" of the CGI resource. This is not particularly significant, except that it notifies the client that this is a restricted resource, and that a userid/password will be required. The user sees this string when prompted for his or her credentials.
- ADMIN_GROUP - this was the group defined above. In this context, it applies to the read access rights. To read this resource (that is, to use the CGI), the user needs to be in this group.

- 0x0000 - this is also a group bit parameter, for write access. CGIs themselves do not have the concept of “writability” (that would imply the ability to change the CGI function!) so this is always zero for a CGI.
- SERVER_HTTP - this specifies the server that can use the CGI function. CGIs are currently only usable by the HTTP server, thus there is no other sensible choice for this parameter.
- SERVER_AUTH_BASIC - this specifies the required (minimum) authentication method. BASIC means that a simple plain-text userid and password will be required. A better choice is SERVER_AUTH_DIGEST since that does not reveal the password to anyone listening in on the conversation; however, older web browsers do not support this.

Next, the file system needs to be protected. Usually, you do not want the entire file system to be writable, even to the privileged group members. To establish this sort of protection, you need to set up a rule-based access control. This is done using the SSPEC_RULETABLE method, or equivalent runtime control:

```
#define SSPEC_FLASHRULES
...
#use "http.lib"
...
SSPEC_RULETABLE_START
    SSPEC_MM_RULE("/A/new", "newPages", 0xFFFF, ADMIN_GROUP,
        SERVER_HTTP, SERVER_AUTH_NONE, NULL)
SSPEC_RULETABLE_END
```

The SSPEC_FLASHRULES macro must be defined before you #use “http.lib”. The rule table has one entry in this example. The parameters to this entry are:

- “/A/new” - this is the string prefix of all file names to which this rule applies. In this example, everything in the first FAT partition (/A/) with a filename starting with “new” is protected according to the remaining parameters. This includes any file in the root directory whose name starts with “new,” or any file in any subdirectory of the root directory where the subdirectory name starts with “new.”
- “newPages” - this is the realm string assigned to these files. This is the same as the CGI realm, but need not be.
- 0xFFFF - this is the user groups who are allowed read access. In this case, everyone is allowed.
- ADMIN_GROUP - this is the writable group: only the one defined for the CGI is allowed.
- SERVER_HTTP - only the HTTP server can access.
- SERVER_AUTH_NONE - this is only relevant when the resource is being read directly by the client. When the file is written (via the CGI) the CGI has already authenticated the user in its own way, and doesn’t need to re-authenticate. In this example, no authentication is required for retrieval (read-only) of the file.
- NULL - this is an additional parameter that is not relevant to this discussion.

By default, every other file in the filesystem(s) that is not covered by this rule is denied write access. In general, a rule is only required when it is desired to permit write access (not deny it).

Step 10: Create Set of User IDs

The last step is to actually define the users. This must be done at runtime, using the `sauth_*()` functions. The following code illustrates:

```
int uid;

uid = sauth_adduser("admin", "upload", SERVER_HTTP);
sauth_setusermask(uid, ADMIN_GROUP, NULL);
sauth_setwriteaccess(uid, SERVER_HTTP);
```

This sets up a single user, with userid “admin” and password “upload.” The user is only “known” to the HTTP server. `sauth_setusermask()` is required when a userid is created (since the default may not be satisfactory). It makes sure the user is placed into the correct group(s), in this case, the `ADMIN_GROUP` that we defined above. Finally, each user must be *individually* granted write access using the `sauth_setwriteaccess()` function. If this is not done, the user will not be able to write the file in spite of passing other tests.

Step 11: Tying It All Together

After performing the above steps, the actual running of the HTTP server and CGI is almost trivial. The main C function should have a loop in it which calls `http_handler()`:

```
void main()
{
    int uid;
    sock_init();                // Initialize the network
    // Mount the FAT filesystem.
    sspec_automount(SSPEC_MOUNT_ANY, NULL, NULL, NULL);
    // Create the authorized user, as described in the previous section.
    uid = sauth_adduser("admin", "upload", SERVER_HTTP);
    sauth_setusermask(uid, ADMIN_GROUP, NULL);
    sauth_setwriteaccess(uid, SERVER_HTTP);

    http_init();                // Initialize the HTTP server
    tcp_reserveport(80);        // Enable smooth handling of multiple HTTP requests
    for (;;) http_handler();    // The big loop! Drives everything.
}
```

All error handling has been pared out of the above code. For full details, please refer to the sample program `samples\tcpip\http\upld_fat.c`.

4.7 API Functions for HTTP Servers

`cgi_continue`

```
int cgi_continue(HttpState * state, char * localurl)
```

DESCRIPTION

Called from a CGI function after processing any data submitted. This function continues creating a response as if from a normal GET request to the specified local URL.

NOTE: the CGI function must NOT have sent any data to the socket.

PARAMETERS

<code>state</code>	A pointer to the HTTP server state structure.
<code>localurl</code>	The URL string, which must be a URL defined in the server spec table (otherwise the browser will see a "not found" message).

RETURN VALUE

The return value from this function should be used as the return value from the CGI handler function that calls it.

LIBRARY

`HTTP.LIB`

cgi_redirectto

```
void cgi_redirectto( HttpState *state, char *url );
```

DESCRIPTION

This utility function may be called in a CGI function to redirect the user to another page. It sends a user to the URL stored in `url`. You should immediately issue a “`return 0;`” after calling this function. The CGI is considered finished when you call this, and will be in an undefined state.

The http samples work correctly with `cgi_redirectto()` because they use macro constants to define the URL parameter. If you manipulate the url string, please be aware of the following issues:

- The library function sets a pointer to the 2nd parameter - url. The calling routine is responsible for ensuring that the location represented by the pointer remains valid after the call. This is because the URL string will not be processed until after the CGI function is finished.
- If the application has `MAX_TCP_SOCKET_BUFFERS` and `HTTP_MAXSERVERS` set to more than one, it is possible that the CGI function will be called successively with different server states serving different client requests. In these circumstances it is necessary to ensure that the pointer to the url is valid for each of the server states.
- After the cgi function has called `cgi_redirectto()` and returns 0, the `http_handler` then causes the server response to be sent to the browser. The information is sent as follows:
 1. HTTP header response containing the redirection information response code 302.
 2. A human readable redirection html page telling the user that redirection has taken place, and to click "here" to go to the new URL. This is for browsers that do not recognize the redirection 302 command in the header.

This may cause a problem for browsers which **do** recognize the 302 redirection command. Some browsers immediately issue a GET request to the new location while still reading in the human readable page. If `MAX_TCP_SOCKET_BUFFERS` and `HTTP_MAXSERVERS` are set to one, the server will not receive the GET request because it is busy sending out the human-readable page. The symptom is that the browser appears to time-out. (This timing problem may be masked when a proxy server is used.) Set `MAX_TCP_SOCKET_BUFFERS` and `HTTP_MAXSERVERS` to a value more than one to prevent this problem.

PARAMETERS

state Current server struct, as received by the CGI function.
url Fully qualified URL to redirect to.

RETURN VALUE

None - sets the state, so the CGI must immediately return with a value of 0.

LIBRARY

HTTP.LIB

SEE ALSO

`cgi_sendstring`

`cgi_sendstring`

```
void cgi_sendstring( HttpState *state, char *str );
```

DESCRIPTION

Sends a string to the user. You should immediately issue a “`return 0;`” after calling this function. The CGI is considered finished when you call this, and will be in an undefined state. This function greatly simplifies a CGI handler because it allows you to generate your page in a buffer, and then let the library handle writing it to the network.

PARAMETERS

state Current server struct, as received by the CGI function.
str String to send.

RETURN VALUE

None - sets the state, so the CGI must immediately return with a value of 0.

LIBRARY

HTTP.LIB

SEE ALSO

`cgi_redirectto`

http_abortCGI

```
int http_abortCGI( HttpState * state );
```

DESCRIPTION

Terminate this CGI request. The client will receive an error message indicating the connection was closed.

The CGI should not make any further HTTP calls after calling this function. It should clean up any resources that it opened, since no further calls are made to this CGI for this request.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

0

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction, http_skipCGI, http_switchCGI, http_finishCGI,
http_write

http_addfile

```
int http_addfile( char *name, long location );
```

DESCRIPTION

Adds a file to the dynamic resource table.

PARAMETERS

name	Name of the file (for example, /index.html).
location	Address of the file data. (Return value from #ximport)

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

HTTP.LIB

SEE ALSO

http_delfile

http_contentencode

```
char *http_contentencode( char *dest, const char *src,
                          int len );
```

DESCRIPTION

Converts a string to include HTTP transfer-coding *tokens* (such as `@` (decimal) for at-sign) where appropriate. Encodes these characters: "`<>@%#&`"

Source string is `NULL`-byte terminated. Destination buffer is bounded by `len`. This function is reentrant.

PARAMETERS

<code>dest</code>	Buffer where encoded string is stored.
<code>src</code>	Buffer holding original string (not changed)
<code>len</code>	Size of destination buffer.

RETURN VALUE

`dest`: There was room for all conversions.
`NULL`: Not enough room.

LIBRARY

`HTTP.LIB`

SEE ALSO

`http_urldecode`

http_date_str

```
char *http_date_str( char *buf );
```

DESCRIPTION

Print the date (time zone adjusted) into the given buffer. This assumes there is room!

PARAMETERS

buf	The buffer to write the date into. This requires at least 30 bytes in the destination buffer.
------------	---

RETURN VALUE

A pointer to the string.

LIBRARY

HTTP.LIB

SEE ALSO

http_handler

http_defaultCGI

```
int http_defaultCGI(HttpState * state);
```

DESCRIPTION

This function should not be called directly by the application. It is intended to be used as a new-style CGI for handling file uploads. See "samples\tcpip\http\upld_fat.c" for an example of using this function.

This CGI function accepts POST requests from the client (browser) which may contain one or more files that are being uploaded. It looks at the field name of the form data in the request. If the field name starts with "/", it is assumed to be the name of a resource which is to be created (if it does not already exist) and overwritten with the uploaded file contents.

There are three steps required to use this CGI:

1. Define a CGI resource in the flash- or ram-spec table. If using flashspec, for example, there would be an entry like

```
SSPEC_RESOURCETABLE_START
SSPEC_RESOURCE_XMEMFILE("/index.html", index_html),
SSPEC_RESOURCE_CGI("/upload.cgi", http_defaultCGI)
SSPEC_RESOURCETABLE_END
```

There may be other resources, but at least two are normally required. One resource is a web page (see below) that contains a form the user can fill in with the name of the file to upload. The other resource (CGI) is a reference to this function, giving it a URL name that identifies it to the browser.

2. Create a web page which contains a form like the following skeleton example:

```
<FORM ACTION="/upload.cgi" METHOD="POST"
      enctype="multipart/form-data">
  <INPUT TYPE="FILE" NAME="/A/incoming/new.htm">
  <INPUT TYPE="SUBMIT" VALUE="Upload">
</FORM>
```

in the <FORM> element, the ACTION= parameter specifies the URL assigned to this CGI. In the <INPUT TYPE="FILE"> element, the NAME= parameter specifies the resource name used to contain the uploaded file contents. In this example, the resource is called "/A/incoming/new.htm", which will work if you are using the FAT filesystem.

If uploading to a subdirectory, "incoming" in the above example, the subdirectory must already exist. If not, the upload will fail.

3. To add user authentication and other facilities there are three possible things to protect:

- The web page containing the form. Give read access only to those users who could conceivably upload the files specified therein.
- The CGI itself (this function). Protect as for (a).
- The uploaded resource. You should set up a rule allowing write access only to the intended user(s).

When defining user IDs which can use the upload, do not forget to give those users overall write access using, for example:

```
sauth_setwriteaccess (uid, SERVER_HTTP)
```

Be aware that “rogue clients” could easily change the resource name to something other than the one that was intended in the original form. This is why resource protection is important.

Having done these three things, the HTTP server is now set up to automatically place uploaded files in the filesystem.

Note that this CGI is limited to placing files into fixed resource locations (as specified by the field name of the INPUT element). If you need more sophisticated control, you may wish to write your own CGI function, using the code of this one as a starting point.

This CGI also presents a default status web page back to the client. This page indicates whether the upload was successful, the number of bytes uploaded, and a link to test out the new file (assuming it is something the browser will understand, such as an HTML document or GIF image). You can use this function as a starting point for generating your own content.

PARAMETERS

state	HTTP state pointer, provided by HTTP server to all CGIs.
newURL	The resource name to present to the client. This may be another CGI, or any other type of resource that could be presented to the client in response to an HTTP GET or POST request. The resource must exist in the flash- or ram-spec table, or in a filesystem.

RETURN VALUE

See documentation for "writing a data handler CGI"

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction, http_skipCGI, http_switchCGI, http_finishCGI, http_write

http_delfile

```
int http_delfile( char *name );
```

DESCRIPTION

Deletes a file from the RAM spec table.

PARAMETERS

name Name of the file, as passed to `http_addfile()`.

RETURN VALUE

0: Success;
1: Failure (not found).

LIBRARY

HTTP.LIB

SEE ALSO

`http_addfile`

http_finderrbuf

```
char *http_finderrbuf( char *name );
```

DESCRIPTION

Finds the occurrence of the given variable in the HTML form error buffer, and returns its location.

PARAMETERS

name Name of the variable.

RETURN VALUE

NULL: Failure.

!NULL: Success, location of the variable in the error buffer.

LIBRARY

HTTP.LIB

http_findname

```
int http_findname(char *name);
```

DESCRIPTION

Finds a spec entry, searching first in RAM, then in flash.

This function is deprecated as of Dynamic C 8.50. Use `sspec_findname()`.

PARAMETERS

name Name, in text, of the spec to find.

RETURN VALUE

The spec entry.

LIBRARY

HTTP.LIB

http_finishCGI

```
int http_finishCGI( HttpState * state );
```

DESCRIPTION

Indicate to the HTTP server that this CGI has finished processing data from this multi-part data stream. The server reads (and discards) data to the end of the entire stream (including epilog). The next call to the CGI function will have an action code of CGI_EOF (or possibly CGI_ABORT if there was a stream error).

PARAMETERS

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

0

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction, http_skipCGI, http_abortCGI, http_switchCGI, ,
http_write

http_getAction

```
char http_getAction( HttpState * state );
```

DESCRIPTION

Return the current CGI action. This should be called only from a CGI function registered as a SSPEC_CGI resource in the zserver resource table.

NOTE: This is implemented as a macro. You must define the macro USE_HTTP_UPLOAD if using this macro, otherwise you will get a compile-time error.

http_getAction() should be called at the top of the CGI function. Other http_get* functions/macros may or may not be valid depending on the action code. The following table shows which functions are applicable:

Table 4-1. Valid Functions per Action Code

CGI Action Code	Valid Functions/Macros
Any action code except CGI_ABORT	http_getContext, http_getURL, http_getState, http_setState, http_getCond, http_setCond, http_getUserState, http_getSocket, http_write, http_abortCGI, http_skipCGI, http_finishCGI, http_switchCGI, http_getHTTPVersion, http_getHTTPMethod, http_getRemainingLength
CGI_START	http_getField, http_getContentLength, http_getContentType, http_getContentDisposition, http_getTransferEncoding
CGI_DATA	http_getField, http_getContentLength, http_getContentType, http_getContentDisposition, http_getTransferEncoding, http_getData, http_getDataLength
CGI_END	http_getField, http_getContentLength, http_getContentType, http_getContentDisposition, http_getTransferEncoding
CGI_HEADER, CGI_PROLOG, CGI_EPILOG, CGI_EOF	http_getData, http_getDataLength
CGI_CONTINUE	Depends on previous action code at time of returning CGI_MORE, however http_getData will NOT be valid.

Table 4-1. Valid Functions per Action Code

CGI Action Code	Valid Functions/Macros
CGI_ABORT	Should only do resource cleanup. http_getContext, http_getURL, http_getState, http_getCond, http_getUserState, http_getHTTPVersion, http_getHTTPMethod

PARAMETER

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

Action code. One of the following values:

- CGI_START - start of a part in a multi-part transfer.
- CGI_DATA - binary data for this part
- CGI_END - end of a part
- CGI_HEADER - header line of a part
- CGI_PROLOG - binary data before the first part
- CGI_EPILOG - line of data after the last part
- CGI_EOF - normal end of all parts and epilog
- CGI_ABORT - abnormal termination. CGI should recover and/or close any open resources.
- CGI_CONTINUE - being called from the HTTP server after the CGI previously returned CGI_MORE.

LIBRARY

HTTP.LIB

SEE ALSO

(functions mentioned above), http_defaultCGI

http_getCond

```
int http_getCond( HttpState * state, int idx );
```

DESCRIPTION

Return the current HTTP condition state variable (aka., cond variable). There are `HTTP_MAX_COND` of these integer state variables, thus `idx` must be between 0 and `HTTP_MAX_COND-1`, inclusive.

Use of cond variables is entirely up to the application; however, they are initialized by the HTTP server under certain conditions. By default, they are set to zero at the start of each request from the client. If the client request includes URL GET-type parameters of the form `http://host/resource.html?A=1&B=2&C=3` etc. then cond state 0 is set to the value for 'A', cond state 1 is set to the value for 'B' etc. The values must be integers, which are coerced into 16 bit signed integers.

NOTE: This is implemented as a macro.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
idx	Index of cond variable: 0.. <code>HTTP_MAX_COND-1</code> . Validity is not checked.

RETURN VALUE

Value of cond variable `idx`.

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`, `http_setCond`

http_getContentDisposition

```
char http_getContentDisposition( HttpState * state );
```

DESCRIPTION

Return the current disposition of the data which is being provided by the client. This is one of the following enumerated values:

- `MIME_DISP_NONE`: unspecified disposition
- `MIME_DISP_INLINE`: the content is to be displayed "inline"
- `MIME_DISP_ATTACHMENT`: the content is only to be displayed if there is some action by the user
- `MIME_DISP_FORMDATA`: the content is form field data (or an uploaded file).

Of these, only `NONE` and `FORMDATA` are really relevant to HTTP. It is only valid to call this when the action code is `CGI_START`, `CGI_DATA` or `CGI_END`.

NOTE: This is implemented as a macro.

PARAMETER

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

Content disposition code, as documented above.

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`

http_getContentLength

```
long http_getContentLength( HttpState * state );
```

DESCRIPTION

Return the length of data in the current part of a multi-part data stream. The return value is interpreted differently, depending on the action code.

It is only valid to call this when the action code is CGI_START, CGI_DATA or CGI_END.

When CGI_START, this returns the value of the ContentLength header for this part (or -1 if there was no such header).

When CGI_DATA or CGI_END, it is the total number of bytes that have actually been read and presented to the CGI. This increases for each CGI_DATA call, until it represents the total content length when action is CGI_END.

NOTE: This is implemented as a macro.

PARAMETER

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

Length of part data.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_getContentType

```
char * http_getContentType( HttpState * state );
```

DESCRIPTION

Return the current content type of the data which is being provided by the client. This is a MIME type string e.g. "text/html" or "image/jpeg".

The CGI might need to look at this to determine the appropriate way to process the data. Normal form fields will usually contain "text/plain", however uploaded files may contain any type of data.

It is only valid to call this when the action code is CGI_START, CGI_DATA or CGI_END.

NOTE: This is implemented as a macro.

PARAMETER

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

Null terminated string containing the MIME type name.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_getcontext

```
ServerContext * http_getcontext( int servno );
```

DESCRIPTION

Return the `ServerContext` struct for the specified HTTP server instance.

NOTE: This structure should not be modified by the application.

PARAMETER

servno Server instance number (0..HTTP_MAXSERVERS-1)

RETURN VALUE

NULL: invalid server instance.

Otherwise, pointer to this server's `ServerContext`.

LIBRARY

HTTP.LIB

http_getContext

```
ServerContext * http_getContext( HttpState * state );
```

DESCRIPTION

Return the current HTTP server context. The context pointer is required by many zserver resource handler functions.

NOTE: This is implemented as a macro.

PARAMETER

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

Pointer to the HTTP server's context structure. See zserver documentation.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_getData

```
char * http_getData( HttpState * state );
```

DESCRIPTION

Return a pointer to the data that is available. It is only valid to call this if the action code is one of CGI_DATA, CGI_PROLOG, CGI_EPILOG, CGI_HEADER or CGI_EOF.

When CGI_DATA, this is the next chunk of data received as the content of the current part of a multi-part transfer. The data arrives in arbitrary amounts. CRLF boundaries (if any) are not respected, and the data may contain NULLs and other binary values. THE CGI MUST CONSUME ALL DATA PROVIDED since the data will not be presented again on the next call.

When CGI_PROLOG, this is data that occurs before the first boundary (part) but after the main HTTP headers. This data (like that for CGI_DATA) is not line-oriented.

When CGI_EPILOG, CGI_HEADER or CGI_EOF, the data will be a complete line of input (with the terminating CRLF stripped off). The returned string will also be null-terminated. When CGI_EOF, the data (if any) is technically part of the epilog.

Prolog data is lines of input that were provided before the first "official" part of the multi-part data. Most HTTP clients will not provide any prolog data. Epilog data is lines of data after the last official part. Again, HTTP clients do not usually generate it. It is always safe to ignore prolog and epilog data, since it is usually provided only for non-MIME compliant servers.

Data provided when the action is CGI_HEADER is a line of header data provided at the start of each part of the multi-part data. It is safe for the CGI to ignore header lines, since the HTTP server also processes the ones that it needs. The CGI is given these header lines so that it can extract useful or customized information if desired.

The length of the data may be obtained using `http_getDataLength()`.

The CGI is allowed to overwrite data at the returned area, provided that it writes no more than `HTTP_MAXBUFFER` bytes.

NOTE: This is implemented as a macro.

PARAMETER

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

Pointer to the first character of data.

SEE ALSO

`http_getAction`

http_getDataLength

```
word http_getDataLength( HttpState * state );
```

DESCRIPTION

Return the length of data that is available. It is only valid to call this if it is valid to call `http_getData()`. That is, if the action code is one of `CGI_DATA`, `CGI_PROLOG`, `CGI_EPILOG`, `CGI_HEADER` or `CGI_EOF`.

NOTE: This is implemented as a macro.

PARAMETER

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

Length of available data. This will range from 0 to `HTTP_MAXBUFFER`. 0 will only be returned for `PROLOG` and `EPILOG` when a blank line is read.

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`

http_getField

```
char * http_getField( HttpState * state );
```

DESCRIPTION

Return the current form field name. This function should only be called when the action code is CGI_START, CGI_DATA or CGI_END.

NOTE: This is implemented as a macro.

PARAMETER

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

Null-terminated string containing the current field name. The field name is the name of a form element, specified using, for example,

```
<INPUT TYPE="TEXT" NAME="srv_file">
```

in the HTML, where `srv_file` is the field name.

If there was no "name=" parameter in the returned form data, this will be an empty string (zero length, not NULL).

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`

http_getHTTPMethod

```
char http_getHTTPMethod( HttpState * state );
```

DESCRIPTION

Return the HTTP request method of the current request protocol. The CGI might need to look at this to generate the correct response headers.

NOTE: This is implemented as a macro.

PARAMETER

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

One of the following codes:

- `HTTP_METHOD_GET` - "GET" i.e., normal retrieval, without making any permanent state update.
- `HTTP_METHOD_POST` - "POST" i.e., uploading some information to be stored, or making some permanent state change. This is the normal method for invoking CGIs.
- `HTTP_METHOD_HEAD` - "HEAD" i.e., the client only wants the headers, not the actual content e.g. it might be trying to determine the most recent modification date.

Other codes may be returned in the future.

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`

http_getHTTPVersion

```
char http_getHTTPVersion( HttpState * state );
```

DESCRIPTION

Return the HTTP version number of the current request protocol. The CGI might need to look at this in order to generate the correct response headers.

NOTE: This is implemented as a macro.

PARAMETER

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

One of the following codes:

- HTTP_VER_09 - version 0.9
- HTTP_VER_10 - version 1.0
- HTTP_VER_11 - version 1.1

Other codes may be returned in the future.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_getRemainingLength

```
long http_getRemainingLength( HttpState * state );
```

DESCRIPTION

Return the remaining length of the incoming data stream. This length includes all parts (not just the current part) and also includes the boundary separators and epilog data. Normally, this value will be zero when the action code is CGI_EOF. If the value is negative, then the client might not have indicated the total data length, or might not have set the right value.

NOTE: This is implemented as a macro.

PARAMETER

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

Length of remaining data, or negative if not known.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_getSocket

```
tcp_Socket * http_getSocket( HttpState * state );
```

DESCRIPTION

Return the current HTTP server socket. The socket may be written/read; however, this is inadvisable since it may interfere with the server's use of it.

NOTE: This is implemented as a macro.

PARAMETER

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

Pointer to the HTTP server's TCP socket structure.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_getState

```
int http_getState( HttpState * state );
```

DESCRIPTION

Return the current primary HTTP CGI state variable.

Use of this state variable is entirely up to the application; however, it is initialized by the HTTP server to zero before calling the CGI for the first time.

NOTE: This is implemented as a macro.

PARAMETER

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

Value of primary state variable.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_getTransferEncoding

```
char http_getTransferEncoding( HttpState * state );
```

DESCRIPTION

Return the current encoding of the data which is being provided by the client. This is one of the following enumerated values:

- CTE_BINARY The default
- CTE_7BIT 7-bit safe ASCII
- CTE_8BIT 8-bit ASCII
- CTE_QP Quoted printable
- CTE_BASE64 Base 64

Of these, the CGI is only likely to see CTE_BINARY, since HTTP is an 8-bit protocol, and most clients (browsers) will not bother to encode the data. Encoding is only an issue for internet mail, which sometimes has to cross interfaces that do not support full 8-bit binary transfers.

If the CGI detects a transfer encoding that requires non-null operation (that is, CTE_QP or CTE_BASE64) then it should either reject the transfer, or decode the data as it comes in.

It is only valid to call this when the action code is CGI_START, CGI_DATA or CGI_END.

NOTE: This is implemented as a macro.

PARAMETER

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

Transfer encoding code, as documented above.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_getURL

```
char * http_getURL( HttpState * state );
```

DESCRIPTION

Return the URL of the current HTTP client request. In a CGI, this will usually be something like `foo.cgi`.

NOTE: This is implemented as a macro.

PARAMETER

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

Null-terminated string containing the URL. Note that GET-style form parameters will be stripped off: for example, the URL, `foo.cgi?A=99&D=-45`, will be returned as `foo.cgi`.

The GET parameters are available using `http_getCond()`.

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`

http_getUserState

```
void * http_getUserState( HttpState * state );
```

DESCRIPTION

Get the "user state" area of the HTTP server structure. This is an area of memory that can be used by the CGI to keep track of its internal state, from call to call.

The size of this area is `HTTP_USERDATA_SIZE`. If that macro is not defined, it defaults to zero, so use of the `http_getUserState` macro will result in a compile-time error.

NOTE: This is implemented as a macro.

Example:

```
typedef struct { ... } myCGIdata;
...
#define HTTP_USERDATA_SIZE sizeof(myCGIdata)
#include "http.lib"
...
int myCGI(HttpState * s) {
    myCGIdata * d;
    d = (myCGIdata *)http_getUserState(state);
    ...
}
```

PARAMETERS

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

Returns the address of the first byte of the user area. This should be cast to the appropriate structure type.

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`

http_handler

```
void http_handler( void );
```

DESCRIPTION

This is the basic control function for the HTTP server, a tick function to run the HTTP daemon. It must be called periodically for the daemon to work. It parses the requests and passes control to the other handlers, either `html_handler`, `shtml_handler`, or to the developer-defined CGI handler based on the request's extension.

LIBRARY

HTTP.LIB

SEE ALSO

`http_init`

http_idle

```
int http_idle( void );
```

DESCRIPTION

Query to see if any HTTP servers are active.

RETURN VALUE

0: at least one HTTP server is active
1: all HTTP servers are idle

LIBRARY

HTTP.LIB

SEE ALSO

http_handler

http_init

```
int http_init( void );
```

DESCRIPTION

Initializes the HTTP daemon. This must be called after `sock_init()`, and before calling `http_handler()` in a loop.

This sets the root directory to "/" and sets the default file name to `index.html`. You can change these defaults by calling `http_set_path()` after this function.

You can override these defaults at compile-time by defining the macros

```
#define HTTP_HOMEDIR "/"
#define HTTP_DFLTFILE "index.html"
```

to be something other than these defaults. If you do this, then there is no need to invoke the `http_set_path()` function.

RETURN VALUE

0: Success.

LIBRARY

HTTP.LIB

SEE ALSO

`http_handler`, `http_shutdown`, `http_status`, `http_set_path`

http_nextfverr

```
void http_nextfverr( char *start, char **name, char **value,  
    int *error, char **next );
```

DESCRIPTION

Gets the information for the next variable in the HTML form error buffer. If any of the last four parameters in the function call are `NULL`, then those parameters will not have a value returned. This is useful if you are only interested in certain variable information.

PARAMETERS

start	Pointer to the variable in the buffer for which we want to get information.
name	Return location for the name of the variable.
value	Return location for the value of the variable.
error	Return location for whether or not the variable is in error (0 if it is not, 1 if it is).
next	Return location for a pointer to the variable after this one.

RETURN VALUE

None, although information is returned in the last four parameters.

LIBRARY

`HTTP.LIB`

http_parseform

```
int http_parseform( int form, HttpState *state );
```

DESCRIPTION

Parses the returned form information. It expects a POST submission. This function is useful for a developer who only wants the parsing functionality and wishes to generate forms herself. Note that the developer must still build the array of `FormVars` and use the `server_spec` table. This function will not, however, automatically display the form when used by itself. If all variables satisfy all integrity checks, then the variables' values are updated. If any variables fail, then none of the values are updated, and error information is written into the error buffer. If this function is used directly, the developer must process errors.

PARAMETERS

form	server_spec index of the form (i.e., location in TCP/IP servers' object list).
state	The HTTP server with which to parse the POSTed data.

RETURN VALUE

0: There is more processing to do;
1: Form processing has been completed.

LIBRARY

HTTP.LIB

http_safe

```
int http_safe( char * to, char * from, int tolen, int fromlen );
```

DESCRIPTION

Convert a http-unsafe string in `from` (length `fromlen`) into a properly escaped string. For example, the string "hello&goodbye<>" would be changed to "hello&goodbye<>".

Returns non-zero if result could not fit in `tolen-1` bytes. A null is always added, thus `tolen` should account for this. Double quotes are escaped since the result may itself be quoted.

Newline characters are turned into HTML line break "
" markup. Control characters (codes less than 32) are turned into "&#xx;" where "xx" is the hexadecimal control char value. The source string can contain null character(s) which is why its length is passed in the parameter `fromlen`.

PARAMETERS

<code>to</code>	Destination buffer for escaped string
<code>from</code>	Source buffer for string to convert
<code>tolen</code>	Length of destination buffer (must be at least equal to <code>fromlen</code> , since string is never smaller than source string).
<code>fromlen</code>	Length of source buffer.

RETURN VALUE

0 on success
non-zero if resulting string (plus its null terminator) could not fit in the provided buffer.

LIBRARY

HTTP.LIB

SEE ALSO

`http_handler`

http_scanpost

```
int http_scanpost( char *tag, char *buffer, char *dest,  
int maxlen );
```

DESCRIPTION

This function allows you to scan a buffer with a POST response for the key/value pairs. This function is reentrant.

PARAMETERS

tag	Buffer holding the tag name.
buffer	Buffer to read data from.
dest	Buffer to store value to.
maxlen	Size of destination buffer.

RETURN VALUE

0: Successful
!0: Not successful

LIBRARY

HTTP.LIB

http_set_anonymous

```
int http_set_anonymous( int uid );
```

DESCRIPTION

Set the “anonymous” user ID. This is the assumed user ID when no credentials are provided by the client (browser). A typical use of this function would be:

```
int anon;
anon = sauth_adduser("anonymous", "",
    SERVER_FTP|SERVER_HTTP);
sauth_setusermask(uid, WORLD_GROUP, NULL);
http_set_anonymous(uid);
ftp_set_anonymous(uid);           // if using FTP too
```

which defines an “anonymous” login for the HTTP and, optionally, the FTP servers. (Since FTP also requires an anon user, you can use the same user ID for both FTP and HTTP).

When a web browser initially requests a resource, it may not pass any user credentials (i.e., user name and password). The HTTP server will assume that the user is anonymous, and apply the access permissions tests on that basis. If access is denied, then the browser will prompt the user for a real user name and password, and the request will be re-tried.

You do not always need to define an anonymous user to HTTP. But it is required if you have some resource which is (say) protected for write access, but you want any user to be able to retrieve the resource without requiring a user name/password.

NOTE: This function is non-reentrant. It sets a global variable which is accessed by all HTTP server instances. For this reason, you should call this function once only before starting to call `http_handler()`.

PARAMETER

uid	The userID to use as the anonymous user. This should have been defined using <code>sauth_adduser()</code> . Pass -1 to set no anonymous user. In this case, only resources which are completely free of any access controls will be accessible to users who do not provide credentials.
------------	---

RETURN VALUE

Same as the uid parameter, except -1 if uid invalid.

LIBRARY

HTTP.LIB

SEE ALSO

`sauth_adduser`, `ftp_set_anonymous`, `sauth_setusermask`

http_setauthentication

```
int http_setauthentication( int auth );
```

DESCRIPTION

Sets the type of authentication to be used globally by the HTTP server. By default, this is set to the strongest available type of authentication available (in order of weakest to strongest: HTTP_NO_AUTH, HTTP_BASIC_AUTH, HTTP_DIGEST_AUTH. This function returns the type of authentication that was actually configured. If the type of authentication that you ask for was not compiled in at compile time, then the type of authentication will not be changed.

NOTE: this function only sets the "default" authentication method for resources who have their authentication method set to SERVER_AUTH_DEFAULT (or, none specified).

PARAMETERS

auth Type of authentication. Choices are:

- HTTP_NO_AUTH
- HTTP_BASIC_AUTH
- HTTP_DIGEST_AUTH

RETURN VALUE

Actual resulting type of authentication.

LIBRARY

HTTP.LIB

http_setCond

```
int http_setCond( HttpState * state, int idx, int val );
```

DESCRIPTION

Set the value of an HTTP condition state variable (aka., cond variable). There are HTTP_MAX_COND of these integer state variables, thus `idx` must be between 0 and HTTP_MAX_COND-1, inclusive.

NOTE: This is implemented as a macro.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
idx	Index of cond variable: 0..HTTP_MAX_COND-1. Validity is not checked.
val	New value.

RETURN VALUE

Returns the new value of the cond variable, i.e., `val`.

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`, `http_getCond`

http_setcookie

```
void http_setcookie( char *buf, char *value );
```

DESCRIPTION

This utility generates a cookie on the client. This will store the text in `value` into a cookie-generation header that will be written to `buf`. The header placed in `buf` is not automatically sent to the web client. It is the caller's responsibility to send the header in `buf`, along with any other HTTP headers, to the client.

When a page is requested from the client, and the cookie is already set, the text of the cookie will be stored in `state->cookie []`. This is a `char*`, and if no cookie was available, `state->cookie [0]` will equal `'\0'`.

PARAMETERS

buf	Buffer to store cookie-generation header, that is, the name of the cookie.
value	Text to store in cookie-generation header, that is, the value of the cookie.

LIBRARY

HTTP.LIB

http_set_path

```
int http_set_path( char * rootdir, char * dfltname );
```

DESCRIPTION

Set the default root directory and resource name for all HTTP server instances. In general, this function should be called once only, after `http_init()` but before `http_handler()`.

The root directory is the base directory and is used as a prefix for all resource requests from clients. For example, if the root directory is set to `"/A/"` then a client request for `http://<hostname>/foo.htm` will look up the resource called `/A/foo.htm` on this server.

The default resource name is used if the client's URL requests a directory. For example, if `dfltname` is set to `"index.htm"` (and `rootdir` is `"/A/"`) then a client request for `"http://<hostname>/admin"` will look up the resource called `"/A/admin"`. If that resource is actually a directory, then it will look up a resource called `"/A/admin/index.htm"`. If it is not a directory, then the default name is not used.

PARAMETERS

rootdir Root directory name to use. This must be a null-terminated string and **MUST** start and end with a forward slash (`/`) character. If this function is not called, the root directory name is set to `"/"` by `http_init()`.

dfltname Default file name to use. This is appended to the directory part of the URL, if the URL actually refers to a directory. If this function is not called, the default file name is set to `index.html` by `http_init()`.

If this parameter is `NULL`, there will be no default name. A request for a directory will generally return a 404 error (not found) to the client. If it is not `NULL`, this parameter must be a null-terminated string. It must not start or end with a `"/` character.

RETURN VALUE

0: OK

-E2BIG: `rootdir` was too long. It should be limited to less than about 12 characters, but you can increase the value of `SSPEC_MAXNAME` if necessary.

-EINVAL: `rootdir` was `NULL`, or did not start and end with a forward slash character.

LIBRARY

`HTTP.LIB`

SEE ALSO

`http_handler`, `http_init`

http_setState

```
int http_setState( HttpState * state, int val );
```

DESCRIPTION

Set the current primary HTTP CGI state variable.

Use of this state variable is entirely up to the application; however, it is initialized by the HTTP server to zero before calling the CGI for the first time.

NOTE: This is implemented as a macro.

PARAMETER

state	HTTP state pointer, as provided in the first parameter to the CGI function.
val	New value for the primary state variable.

RETURN VALUE

Returns the new value, that is, val.

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction

http_shutdown

```
int http_shutdown( int graceful );
```

DESCRIPTION

Shut down the http daemon. Use `http_init()` to restart.

PARAMETER

If non-zero, current connections are allowed to terminate normally. Otherwise, any open connections are reset.

RETURN VALUE

0

LIBRARY

HTTP.LIB

SEE ALSO

`http_handler`, `http_init`, `http_status`

http_skipCGI

```
int http_skipCGI( HttpState * state );
```

DESCRIPTION

Indicate to the HTTP server that the CGI has finished processing this part of a multi-part data stream. The server reads (and discards) data from the stream until the next part is found (or the epilog). When the next part is found, the server continues calling the CGI function as before.

PARAMETERS

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

0

LIBRARY

HTTP.LIB

SEE ALSO

http_getAction, http_abortCGI, http_switchCGI,
http_finishCGI, http_write

http_sock_bytesready

```
int http_sock_bytesready( HttpState *state );
```

DESCRIPTION

HTTP wrapper function for `sock_bytesready()`. This function may be used by CGI applications to determine if there is data waiting on the socket associated with a particular HTTP server.

PARAMETERS

state HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

-1: no bytes waiting to be read
0: in ASCII mode, a blank line is waiting to be read,
 or, for UDP, an empty datagram is waiting to be read
>0: number of bytes waiting to be read

LIBRARY

HTTP.LIB

http_sock_fastread

```
int http_sock_fastread( HttpState *state, byte *dp, int len );
```

DESCRIPTION

HTTP wrapper function for `sock_fastread()`, that is for non-blocking reads (root). This function can be used to read data from a socket associated with a particular HTTP server. This function is intended for use in CGI applications.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
dp	Pointer to return buffer
len	Maximum size of return buffer

RETURN VALUE

>0: the number of bytes read
-1: error

LIBRARY

HTTP.LIB

http_sock_fastwrite

```
int http_sock_fastwrite( HttpState *state, byte *dp, int len );
```

DESCRIPTION

HTTP wrapper function for `sock_fastwrite()`, that is, for non-blocking writes. This function can be used to write data from a root buffer to a socket associated with a particular HTTP server. This function is intended for use in CGI applications.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
dp	Pointer to buffer containing data to be written.
len	Maximum number of bytes to write to the socket.

RETURN VALUE

>0: the number of bytes written
-1: error

LIBRARY

HTTP.LIB

http_sock_gets

```
int http_sock_gets(HttpState *state, byte* dp, int len);
```

DESCRIPTION

HTTP wrapper function for `sock_gets()`. This function can be used by CGI applications to retrieve a string waiting on an ASCII-mode socket associated with a particular HTTP server.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
dp	Pointer to return buffer
len	Maximum size of return buffer

RETURN VALUE

0: if buffer is empty, or
if no “\r” or “\n” is read, but buffer had room *and*
the connection can get more data!
>0: is the length of the string
-1: error

LIBRARY

HTTP.LIB

SEE ALSO

http_sock_mode

http_sock_mode

```
void http_sock_mode( HttpState* state, http_sock_mode_t mode );
```

DESCRIPTION

HTTP socket wrapper function for socket mode. This function can be used by CGI applications to set the mode of a socket associated with a particular HTTP server.

PARAMETERS

- | | |
|--------------|--|
| state | HTTP state pointer, as provided in the first parameter to the CGI function. |
| mode | HTTP mode to use for the socket. Valid values for mode are: <ul style="list-style-type: none">• <code>HTTP_MODE_ASCII</code> - Sets the associated socket to ASCII mode.• <code>HTTP_MODE_BINARY</code> - Sets the associated socket to BINARY. |

RETURN VALUE

None

LIBRARY

`HTTP.LIB`

http_sock_tbleft

```
long http_sock_tbleft(HttpState *state);
```

DESCRIPTION

HTTP wrapper function for `sock_tbleft()`. This function may be used by CGI applications to determine how much space is left in the HTTP socket's transmit buffer.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
--------------	---

RETURN VALUE

Number of bytes of free space remaining in the transmit buffer.

LIBRARY

HTTP.LIB

http_sock_write

```
int http_sock_write( HttpState *state, byte *dp, int len );
```

DESCRIPTION

HTTP wrapper function for blocking writes. This function can be used to write data from a root buffer to a socket associated with a particular HTTP server. This function is intended for use in CGI applications.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
dp	pointer to buffer containing data to be written
len	maximum number of bytes to write to the socket

RETURN VALUE

Number of bytes of written or -1 if there was an error

LIBRARY

HTTP.LIB

http_sock_xfastread

```
int http_sock_xfastread( HttpState *state, long dp, long len );
```

DESCRIPTION

HTTP wrapper function for `sock_fastxread()`, that is, for non-blocking reads (xmem). This function can be used to read data from a socket associated with a particular HTTP server. This function is intended for use in CGI applications.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
dp	Pointer to return xmem buffer.
len	Maximum length of the return xmem buffer.

RETURN VALUE

Number of bytes of read or -1 if there was an error

LIBRARY

HTTP.LIB

http_sock_xfastwrite

```
int http_sock_xfastwrite( HttpState *state, long dp, long len);
```

DESCRIPTION

HTTP wrapper function for `sock_xfastwrite()`, that is for non-blocking writes. This function can be used to write the contents of an xmem buffer to a socket associated with a particular HTTP server.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
dp	Buffer containing data to be written, as an xmem address obtained from, for example, <code>xalloc()</code> .
len	Maximum number of bytes to write to the socket.

RETURN VALUE

Number of bytes of written or -1 if there was an error

LIBRARY

HTTP.LIB

http_status

```
int http_status( void );
```

DESCRIPTION

Determine whether the HTTP server is allowing connections.

RETURN VALUE

0: server is currently disabled
non-zero: server is enabled.

LIBRARY

HTTP.LIB

SEE ALSO

http_handler, http_init, http_shutdown

http_switchCGI

```
int http_switchCGI( HttpState * state, char * newURL );
```

DESCRIPTION

Tell the HTTP server to switch processing to a different CGI function or resource.

The CGI is responsible for generating the correct HTTP response header(s) using `http_write()` etc. If this function is used to pass control to a different CGI, then both CGIs must coordinate so that only one header is written. You can use the HTTP state variable (`http_setState()` and `http_getState()`) and/or `http_getUserState()` to achieve the necessary coordination.

If `newURL` refers to a file or SSI resource (not a CGI), then the CGI function must NOT have already written the HTTP response header(s)—the headers will be generated when the new resource is opened.

If `newURL` refers to a new-style CGI (that is, a CGI resource added using `SSPEC_CGI`, not `SSPEC_FUNCTION`) then that CGI is presented with the remaining content of the current request data stream.

If `newURL` refers to an old-style CGI (that is, a CGI added using `SSPEC_FUNCTION` or `HTTPSPEC_FUNCTION`) then the HTTP server abandons parsing of the request data stream, since old-style CGIs are expected to read the HTTP socket themselves.

Rather than calling `http_switchCGI()`, it is often more convenient to call `cgi_redirectto()`, which tells the client to retrieve the next resource rather than the resource being provided in the current connection. Using `redirect` is less efficient, however.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
newURL	The resource name to present to the client. This may be another CGI, or any other type of resource that could be presented to the client in response to an HTTP GET or POST request. The resource must exist in the flash- or ram-spec table, or in a filesystem.

RETURN VALUE

0

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`, `http_skipCGI`, `http_abortCGI`, `http_finishCGI`,
`http_write`

http_urldecode

```
char *http_urldecode( char *dest, const char *src, int len );
```

DESCRIPTION

Converts a string with URL-escaped "tokens" (such as %20 (hex) for space) into actual values. Changes "+" into a space. String can be NULL terminated; it is also bounded by a specified string length. This function is reentrant.

PARAMETERS

dest	Buffer where decoded string is stored.
src	Buffer holding original string (not changed).
len	Maximum size of string (NULL terminated strings can be shorter).

RETURN VALUE

dest: if all conversion was good.
NULL: if some conversion had trouble.

LIBRARY

HTTP.LIB

SEE ALSO

http_contentencode

http_write

```
int http_write( HttpState * state, char * data, word length );
```

DESCRIPTION

Write data back to the client. This function either sends all of the given data or none of it. If the data cannot be sent (for example, because the socket transmit buffer is already full) then a special return code indicates that the CGI should try again on the next call.

Often, the CGI itself will not need to write anything to the client—the `http_switchCGI()` function takes care of most needs. If this function is used, then the CGI is responsible for generating the correct HTTP response (including headers) and `http_switchCGI()` and similar functions should NOT be called.

Use of this function can often be avoided. Instead, the CGI can copy a string to the pointer provided by `http_getData()`, then return `CGI_SEND`. This will cause the server to send out the (null terminated) string in the buffer, and not call the CGI until the string is sent to the client. See the source to `http_defaultCGI()` for an example of this method.

PARAMETERS

state	HTTP state pointer, as provided to the CGI function.
data	Pointer to first char to transmit. It is OK to make this the same pointer that was returned by <code>http_getData()</code> , since that buffer can be used for output as well as input. In any case, the CGI must ensure that it has processed any incoming data before writing new data to that buffer.
length	Length of data to transmit. There is a limit to the amount of data that <code>http_write()</code> can write on any given call. This limit is set by the HTTP server socket transmit buffer size. This buffer size is given by <code>TCP_BUF_SIZE/2</code> . The transmit buffer is usually at least 1024 bytes. If you try exceeding that limit, <code>http_write()</code> will never succeed.

RETURN VALUE

0: data written (or buffered) successfully.

CGI_MORE: data not written, try again on next call to the CGI. In general, the CGI should pass this code (CGI_MORE) back to the HTTP server. When the server calls the CGI next time, it will set the action code to CGI_CONTINUE which will be a cue to the CGI to try retransmitting the previous data. When CGI_CONTINUE is provided, the contents in the `http_getData()` buffer will not have been altered.

LIBRARY

HTTP.LIB

SEE ALSO

`http_getAction`, `http_skipCGI`, `http_switchCGI`, `http_finishCGI`,
`http_abortCGI`, `http_defaultCGI`

shtml_addfunction

```
int shtml_addfunction( char *name, void (*fptr()) );
```

DESCRIPTION

Adds a CGI/SSI-exec function for making dynamic web pages to the RAM resource table.

PARAMETERS

name	Name of the function (e.g., <code>"/foo.cgi"</code>).
fptr	Function pointer to the handler, that must take <code>HttpState*</code> as an argument. This function should return an <code>int</code> (0 while still pending, 1 when finished).

RETURN VALUE

0: Success;
1: Failure (no room).

LIBRARY

`HTTP.LIB`

SEE ALSO

`shtml_delfunction`

shtml_addvariable

```
int shtml_addvariable( char *name, void *variable, word type,
    char *format );
```

DESCRIPTION

This function adds a variable so it can be recognized by `shtml_handler()`.

PARAMETERS

name	Name of the variable.
variable	Pointer to the variable.
type	Type of variable. The following types are supported: INT8, INT16, INT32, PTR16, FLOAT32.
format	Standard printf format string. (e.g., "%d").

RETURN VALUE

0: Success.
1: Failure (no room).

LIBRARY

HTTP.LIB

SEE ALSO

`shtml_delvariable`

shtml_delfunction

```
int shtml_delfunction( char *name );
```

DESCRIPTION

Deletes a function from the RAM resource table.

PARAMETERS

name Name of the function as given to `shtml_addfunction()`.

RETURN VALUE

0: Success;
1: Failure (not found).

LIBRARY

HTTP.LIB

SEE ALSO

`shtml_addfunction`

shtml_delvariable

```
int shtml_delvariable( char *name );
```

DESCRIPTION

Deletes a variable from the RAM resource table.

PARAMETERS

name Name of the variable, as given to shtml_addvariable().

RETURN VALUE

0: Success;
1: Failure (not found).

LIBRARY

HTTP.LIB

SEE ALSO

shtml_addvariable

5. FTP Client

The library `FTP_CLIENT.LIB` implements the File Transfer Protocol (FTP) for the client side of the connection.

This library supports a single FTP session at any one time since the session state is maintained in a single global structure in root memory.

You can upload and download files to either a static buffer in root data memory (for simple applications) or, starting with Dynamic C version 7.20, you can have the data passed to, or generated by, a data handler callback function that you specify. The data handler function can implement large file transfers in extended memory buffers, or it can be used to generate or process data on-the-fly with minimal buffering.

Starting with Dynamic C 7.20, you can specify “passive” mode transfers. This is most important for clients which are inside a firewall. Passive mode is specified by passing the `FTP_MODE_PASSIVE` option to `ftp_client_setup()`. When passive mode is specified, the client will actively open the data transfer port to the server, rather than the other way around. This avoids the need for the server to penetrate the firewall with an active connection from the outside, which is most often blocked by the firewall. For this reason, it is recommended that your FTP client application uses passive mode by default, unless overridden by an end-user.

5.1 Configuration Macros

The following macros may be defined in a `#define` statement before the inclusion of `FTP_CLIENT.LIB` in an application program. Note that strings must contain the `NULL` byte, so if a maximum string length is 16, the maximum number of characters is 15.

FTP_MAX_DIRLEN

The default is 64, which is the maximum string length of a directory name.

FTP_MAX_FNLEN

The default is 16, which is the maximum string length of a file name.

FTP_MAX_NAMELEN

The default is 16 which is the maximum string length of usernames and passwords.

FTP_MAXLINE

The default is 256, which is both the maximum command line length and data chunk size that can be passed between the FTP data transfer socket and the data handler (if any defined).

FTP_TIMEOUT

The default is 16, which is the number of seconds that pass before a time out occurs.

5.2 API Functions

`ftp_client_setup`

```
int ftp_client_setup( long host, int port, char *username, char
    *password, int mode, char *filename, char *dir, char
    *buffer, int length );
```

DESCRIPTION

Sets up a FTP transfer. It is called first, then `ftp_client_tick()` is called until it returns non-zero. Failure can occur if the host address is zero, if `length` is negative, or if the internal control socket to the FTP server cannot be opened (e.g., because of lack of socket buffers).

PARAMETERS

host	Host IP address of FTP server.
port	Port of FTP server, 0 for default.
username	Username of account on FTP server.
password	Password of account on FTP server.
mode	Mode of transfer: <code>FTP_MODE_UPLOAD</code> or <code>FTP_MODE_DOWNLOAD</code> . You may also OR in the value <code>FTP_MODE_PASSIVE</code> to use passive mode transfer (important if you are behind a firewall).
filename	Filename to get/put.
dir	Directory file is in, <code>NULL</code> for default directory.
buffer	Buffer to get/put the file from/to. Must be <code>NULL</code> if a data handler function will be used. See <code>ftp_data_handler()</code> for more details.
length	On upload, length of file; on download size of buffer. This parameter limits the transfer size to a maximum of 32767 bytes. For larger transfers, it will be necessary to use a data handler function.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

`FTP_CLIENT.LIB`

SEE ALSO

`ftp_client_tick`, `ftp_data_handler`

ftp_client_tick

```
int ftp_client_tick( void );
```

DESCRIPTION

Tick function to run the FTP daemon. Must be called periodically. The return codes are not very specific. You can call `ftp_last_code()` to get the integer value of the last FTP message received from the server. See RFC959 for details. For example, code 530 means that the client was not logged in to the server.

RETURN VALUE

FTPC_AGAIN (0): still pending, call again.
FTPC_OK (1): success (file transfer complete).
FTPC_ERROR (2): failure (call `ftp_last_code()` for more details).
FTPC_NOHOST (3): failure (Couldn't connect to remote host).
FTPC_NOBUF (4): failure (no buffer or data handler).
FTPC_TIMEOUT (5): warning (Timed out on close: data may or may not be OK).
FTPC_DHERROR (6): error (Data handler error in FTPDH_END operation).
FTPC_CANCELLED (7): FTP control socket was aborted (reset) by the server.

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

`ftp_client_setup`, `ftp_client_filesize`, `ftp_client_xfer`,
`ftp_last_code`

ftp_client_filesize

```
int ftp_client_filesize( void );
```

DESCRIPTION

Returns the byte count of data transferred. This function is deprecated in favor of `ftp_client_xfer()`, which returns a long value.

If the number of bytes transferred was over 32767, then this function returns 32767 which may be misleading.

RETURN VALUE

Size, in bytes.

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

`ftp_client_setup`, `ftp_data_handler`, `ftp_client_xfer`

`ftp_client_xfer`

```
longword ftp_client_xfer( void );
```

DESCRIPTION

Returns the byte count of data transferred. Transfers of over 2^{32} bytes (about 4GB) are not reported correctly.

RETURN VALUE

Size, in bytes.

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

`ftp_client_setup`, `ftp_data_handler`, `ftp_client_filesize`

ftp_data_handler

```
void ftp_data_handler( int (*dhnd)(), void *dhnd_data,
    word opts );
```

DESCRIPTION

Sets a data handler for further FTP data transfer(s). This handler is only used if the "buffer" parameter to `ftp_client_setup()` is passed as NULL.

The handler is a function which must be coded according to the following prototype:

```
int my_handler(char *data, int len, longword offset, int
    flags, void *dhnd_data);
```

This function is called with `data` pointing to a data buffer, and `len` containing the length of that buffer. `offset` is the byte number relative to the first byte of the entire FTP stream. This is useful for data handler functions that do not wish to keep track of the current state of the data source. `dhnd_data` is the pointer that was passed to `ftp_data_handler()`.

`flags` contains an indicator of the current operation:

- `FTPDH_IN`: data is to be stored on this host (obtained from an FTP download).
- `FTPDH_OUT`: data is to be filled with the next data to upload to the FTP server.
- `FTPDH_END`: data and `len` are irrelevant: this marks the end of data, and gives the function an opportunity to e.g., close the file. Called after either in or out processing.
- `FTPDH_ABORT`: end of data; error encountered during FTP operation. Similar to `END` except the transfer did not complete. Can use this to e.g., delete a partially written file.

The return value from this function depends on the in/out flag. For `FTPDH_IN`, the function should return `len` if the data was processed successfully and download should continue; -1 if an error has occurred and the transfer should be aborted. For `FTPDH_OUT`, the function should return the actual number of bytes placed in the data buffer, or -1 to abort. If zero is returned, then the upload is terminated normally. For `FTPDH_END`, the return code should be zero for success or -1 for error. If an error is flagged, then this is used as the return code for `ftp_client_tick()`. For `FTPDH_ABORT`, the return code is ignored.

ftp_data_handler (continued)

PARAMETERS

dhnd	Pointer to data handler function, or NULL to remove the current data handler.
dhnd_data	A pointer which is passed to the data handler function. This may be used to point to any further data required by the data handler such as an open file descriptor.
opts	Options word (currently reserved, set to zero).

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

ftp_client_setup

`ftp_last_code`

```
int ftp_last_code( void );
```

DESCRIPTION

Returns the most recent message code sent by the FTP server. RFC959 describes the codes in detail. This function is most useful for error diagnosis in the case that an FTP transfer failed.

RETURN VALUE

Error code; a number between 0 and 999. Codes less than 100 indicate that an internal error occurred e.g., the server was never contacted.

LIBRARY

`FTP_CLIENT.LIB`

SEE ALSO

`ftp_client_setup`, `ftp_client_tick`

5.3 Sample FTP Transfer

Program Name: Samples\tcpip\ftp\ftp_client.c

```
//#define MY_IP_ADDRESS "10.10.6.105"
//#define MY_NETMASK "255.255.255.0"

#define TCPCONFIG 1

#memmap xmem
#use "dcrtcp.lib"
#use "ftp_client.lib"

#define REMOTE_HOST "10.10.6.19"
#define REMOTE_PORT 0

main() {
    char buf[2048];
    int ret, i, j;

    printf("Calling sock_init()...\n");
    sock_init();

    /* Set up the ftp transfer. This is to the host defined above, with a normal
     * anonymous/e-mail password login info. A get of the file bar is requested, which
     * will be stored in buf. */
    printf("Calling ftp_client_setup()...\n");
    if(ftp_client_setup(resolve(REMOTE_HOST), REMOTE_PORT,
        "anonymous", "anon@anon.com", FTP_MODE_DOWNLOAD, "bar",
        NULL, buf, sizeof(buf)))
    {
        printf("FTP setup failed.\n");
        exit(0);
    }
    printf("Looping on ftp_client_tick()...\n");
    while( 0 == (ret = ftp_client_tick()) )
        continue;

    if( 1 == ret ) {
        printf("FTP completed successfully.\n");
        // ftp_client_filesize() returns transfer size, since we asked for download.
        buf[ftp_client_filesize()] = '\0';
        printf("Data => '%s'\n", buf);
    }
    else {
        printf("FTP failed: status == %d\n",ret);
    }
}
```


6. FTP Server

This chapter documents the FTP server. The following information is included:

- configuration macros
- the default file handlers
- how to assign replacement file handlers
- what to do when there is a firewall
- API functions
- commands accepted by the server
- reply codes generated by the server
- sample code demonstrating a working FTP server

The library `FTP_SERVER.LIB` implements the File Transfer Protocol for the server side of a connection. FTP uses two TCP connections to transfer a file. The FTP server does a passive open on well-known port 21 and then listens for a client. This is the command connection. The server receives commands through this port and sends reply codes. The second TCP connection is for the actual data transfer.

Anonymous FTP is supported. Most FTP servers on the Internet use the identifier “anonymous.” So since FTP clients expect it, this is the identifier that is recommended. But any string (with a maximum length of `HTTP_NO_FLASHSPEC SSPEC_NO_STATIC`) may be used.

Dynamic C 8 includes some enhancements that basically let the FTP server act as a full FTP server, where you can create, read and delete files at will. To use these enhancements, the configuration macro `FTP_USE_FS2_HANDLERS` must be defined to enable FS2 support in the default file handler functions. The structure that holds the association of filenames and FS2 file locations is the server spec list—the global array defined in `zserver.lib`. It is stored in the User block and the API functions `ftp_save_filenames()` and `ftp_load_filenames()` are used for support of this.

NOTE: For a demonstration of the enhanced FTP server, see the sample program, `/SAMPLES/TCPIP/FTP/FTP_SERVER_FULL.C`.

6.1 Configuration Macros

The configuration macros control various conditions of the server's operation. Read through them to understand the default conditions. Any changes to these macros may be made in the server application with `#define` statements before inclusion of `FTPSEVER.LIB`.

FTP_CMDPORT

This macro defaults to 21 which is the well-known FTP server port number. You can override this to cause the server to listen on a non-standard port number.

FTP_CREATE_MASK

This macro specifies the mask that is passed into the `servermask` parameter in `sSpec_addfsfile()` calls when a new file is created. In particular, this defines which servers will be allowed to access this file. By default, it is defined to `SERVER_FTP | SERVER_WRITABLE`.

FTP_DTPTIMEOUT

The default is 16, the same as `FTP_TIMEOUT`. This applies to the data transfer port instead of the command port. The data transfer port is involved with `get/store` commands, as well as directory listings.

FTP_EXTENSIONS

The macro is not defined by default. Define it to allow the server to recognize the `DELE`, `SIZE` and `MDTM` commands. If this macro is defined, then the FTP handler structure (`FTPhandlers`) is augmented with pointers to `mdtm` and `delete` handlers.

FTP_INTERFACE

This macro defaults to `IF_DEFAULT`, i.e., the (single) default interface. Define to `IF_ANY` if FTP sessions can be accepted on any active interface, or a specific interface number (e.g., `IF_ETH0`) to allow sessions on that interface only. Note that you are currently limited to a single interface, or all interfaces. This macro is only relevant starting with Dynamic C version 7.30.

FTP_MAXLINE

The default is 256: the number of bytes of the working buffer in each server. This is also the maximum size of each network read/write. The default value of 256 is the minimum value that allows the server to function properly.

FTP_MAXSERVERS

The default is 1: the number of simultaneous connections the FTP server can support. Each server requires a significant amount of RAM (4096 bytes by default, though this can change through `SOCK_BUF_SIZE` or `tcp_MaxBufSize` (deprecated)).

FTP_NODEFAULTHANDLERS

This macro is undefined. Define it to eliminate the code for the default file handlers. You must then provide your own file handlers. This macro is no longer needed starting with Dynamic C version 7.20.

FTP_TIMEOUT

The default is 16: the number of seconds to wait for FTP commands from the remote host before terminating the connection. In a high-latency network this value may need to be increased to avoid premature closures.

FTP_USE_FS2_HANDLERS

Define this to enable the full use of FS2 in the default FTP handler functions. Defining this macro will automatically define `FTP_WRITABLE_FILES` to 1, as well.

FTP_USERBLOCK_OFFSET

This macro should be defined to a number that specifies the offset into the User block at which the list of filenames will be saved. This list correlates the filenames with the locations of the files in the filesystem (FS2). This macro defaults to 0. If the user is putting other information in the User block, this offset may need to be adjusted to prevent clobbering the other data.

FTP_WRITABLE_FILES

The default is 0. Define to 1 to provide support in `ftp_dflt_open()` for authenticating a user for write access before a file is opened. This also provides support in the file listing function, `ftp_dflt_list()`, to show the write permission for writable files.

NOTE: The user will need to override both the write and close default file handlers to provide full support for writing a file.

SSPEC_NO_STATIC

This macro must be defined in any FTP server application compiled with Dynamic C 8.50 or later.

6.2 File Handlers

Default file handlers are provided. The defaults access the server spec list, which is set up using `s_spec_addxmemfile()`, `sauth_adduser()` etc. The default file handlers are used when `NULL` is passed to the initialization function `ftp_init()`.

6.2.1 Replacing the Default Handlers

The `FTPHandlers` structure contains function pointers to the file handlers. This structure may be passed to `ftp_init()` to selectively replace the default file handlers. You may provide a `NULL` pointer for handlers that you do not wish to override. If you have defined `FTP_EXTENSIONS` then there are an additional two function pointers that should be initialized.

```
typedef struct {
    int (*open)();
    int (*read)();
    int (*write)();
    int (*close)();
    long (*getfilesize)();
    int (*dirlist)();
    int (*cd)();
    int (*pwd)();
#ifdef FTP_EXTENSIONS
    long (*mdtm)();
    int (*delete)();
#endif
} FTPHandlers;
```

Starting with Dynamic C 7.30, all FTP server instances share the same set of data handlers. Before this release, there was a separate copy of the handler pointers for each instance of the server. This change does not affect your existing application except to slightly reduce memory usage. This change does add flexibility because it gives any file handler the ability to call any other file handler. In particular, `ftp_dflt_list()` may now call `ftp_dflt_getfilesize()` to get the file's size

6.2.2 File Handlers Specification

Function descriptions for the default handlers are detailed in this section. Additional information is provided in these descriptions when the default handler does not cover the entire function specification.

The default file handlers are in `FTP_SERVER.LIB`.

ftp_dflt_open

```
int ftp_dflt_open( char *name, int options, int uid, int cwd );
```

DESCRIPTION

Opens a file. If a file is successfully opened, the returned value is passed to subsequent handler routines to identify the particular file or resource, as the 'fd' parameter. If necessary, you can use this number to index an array of any other state information needed to communicate with the other handlers. The number returned should be unique with respect to all other open resource instances, so that your handler does not get confused if multiple FTP data transfers are active simultaneously.

Note that the specified file to open may be an absolute or relative path: if the handler supports the concept of directories, then it should handle the path name appropriately and not just assume that the file is in the current directory. If the filename is relative, then the `cwd` parameter indicates the current directory.

PARAMETERS

name	The file to open.
options	File access options: <ul style="list-style-type: none"><code>O_RDONLY</code> (marks file as read-only).<code>O_WRONLY</code> (not currently supported by the default handler).<code>O_RDWR</code> (not used since it's not supported by the FTP protocol).
uid	The userid of the currently logged-in user.
cwd	Current directory (not currently supported by the default handler).

RETURN VALUE

≥ 0 : File descriptor of the opened file.

`FTP_ERR_NOTFOUND`: File not found.

`FTP_ERR_NOTAUTH`: Unauthorized user.

`FTP_ERR_BADMODE`: Requested option (2nd parameter) is not supported.

`FTP_ERR_UNAVAIL`: Resource temporarily unavailable.

In the first case, the returned value is passed to subsequent handler routines to identify the particular file or resource, as the 'fd' parameter. If necessary, you can use this number to index an array of any other state information needed to communicate with the other handlers. The number returned should be unique with respect to all other open resource instances, so that your handler does not get confused if multiple FTP data transfers are active simultaneously. Note that the given file name may be an absolute or relative path: if the handler supports the concept of directories, then it should handle the path name as appropriate and not just assume that the file is in the current directory. If the filename is "relative," then the `cwd` parameter indicates the current directory.

ftp_dflt_getfilesize

```
long ftp_dflt_getfilesize( int fd );
```

DESCRIPTION

Return the length of the specified file. This is called immediately after open for a read file. If the file is of a known constant length, the correct length should be returned. If the resource length is not known (perhaps it is generated on-the-fly) then return -1. For write operations, the maximum permissible length should be returned, or -1 if not known.

PARAMETERS

fd The file descriptor returned when the file was opened.

RETURN VALUE

≥0: The size of the file in bytes.
-1: The length of the file is not known.

ftp_dflt_read

```
int ftp_dflt_read( int fd, char *buf, long offset, int len );
```

DESCRIPTION

Read file identified by `fd`. The file contents at the specified offset should be stored into `buf`, up to a maximum length of `len`. The return value should be the actual number of bytes transferred, which may be less than `len`. If the return value is zero, this indicates normal end-of-file. If the return value is negative, then the transfer is aborted. Each successive call to this handler will have an increasing offset. If the `getfilesize` handler returns a non-negative length, then the read handler will only be called for data up to that length — there is no need for such read handlers to check for EOF since the server will assume that only the specified amount of data is available.

The return value can also be greater than `len`. This is interpreted as "I have not put anything in `buf`. Call me back when you (the server) can accept at least `len` bytes of data." This is useful for read handlers that find it inconvenient to retrieve data from arbitrary offsets, for example a log reader that can only access whole log records. If the returned value is greater than the server can ever offer, then the server aborts the data transfer. The handler should never ask for more than `FTP_MAXLINE` bytes.

PARAMETERS

<code>fd</code>	The file descriptor returned when the file was opened.
<code>buf</code>	Pointer to the buffer to place the file contents.
<code>offset</code>	Offset in the file at which copying should begin.
<code>len</code>	The number of bytes to read.

RETURN VALUE

- 0: EOF.
- >0: The number of bytes read into `buf`.
- 1: Error, transfer aborted.

`ftp_dflt_write`

```
int ftp_dflt_write( int fd, char *buf, long offset, int len );
```

DESCRIPTION

The default write handler does nothing but return zero.

The specification states that the handler may write the file identified by `fd`. `buf` contains data of length `len`, which is to be written to the file at the given offset within the file. The return value must be equal to `len`, or a negative number if an error occurs (such as out of space).

The FTP server does not handle partial writes: the given data must be completely written or not at all. If the return code is less than `len`, an error is assumed to have occurred. Note that it is up to the handler to ensure that another FTP server is not accessing a file which is opened for write. The open call for the other server should return `FTP_ERR_UNAVAIL` if the current server is writing to a file.

PARAMETERS

<code>fd</code>	The file descriptor returned when the file was opened.
<code>buf</code>	Pointer to the data to be written.
<code>offset</code>	Offset in the file at which to start.
<code>len</code>	The number of bytes to write.

RETURN VALUE

≥ 0 : The number of bytes written. If this is less than `len`, an error occurred.
-1: Error.

`ftp_dflt_close`

```
int ftp_dflt_close( int fd );
```

DESCRIPTION

The default close handler does nothing but return zero.

The handler may close the specified file and free up any temporary resources associated with the transfer.

PARAMETERS

`fd` The file descriptor returned when the file was opened.

RETURN VALUE

0

ftp_dflt_list

```
int ftp_dflt_list( int item, char *line, int listing, int uid,
                  int cwd );
```

DESCRIPTION

Returns the next file for the FTP server to list. The file name is formatted as a string.

PARAMETERS

item	Index number starting at zero for the first function call. Subsequent calls should be one plus the return value from the previous call.
line	Pointer to location to put the formatted string.
listing	Boolean variable to control string form: 0: print file name, permissions, date, etc. 1: print file name only.
uid	The currently logged-in user.
cwd	The current working directory.

RETURN VALUE

≥0: File descriptor for last file listed.
-1: Error.

ftp_dflt_cd

```
int ftp_dflt_cd( int cwd, char *dir, int uid );
```

DESCRIPTION

Change to new "directory." This is called when the client issues a CWD command. The FTP server itself has no concept of what a directory is —this is meaningful only to the handler.

PARAMETERS

cwd	Integer representing the current directory.
dir	String that indicates the new directory that will become the current directory. The interpretation of this string is entirely up to the handler. The <code>dir</code> string will be passed as <code>..</code> to move up one level.
uid	The currently logged-in user.

RETURN VALUE

0: No such directory exists.
-1: Root directory.
>0: Anything that is meaningful to the handler.

ftp_dflt_pwd

```
int ftp_dflt_pwd( int cwd, char *buf );
```

DESCRIPTION

Print the current directory, passed as `cwd`, as a string. The result is placed in `buf`, whose length may be assumed to be at least `(FTP_MAXLINE-6)`. The return value is ignored.

PARAMETERS

<code>cwd</code>	The current directory.
<code>buf</code>	Pointer to buffer to put the string.

RETURN VALUE

The return value is ignored.

ftp_dflt_mdtm

```
unsigned long ftp_dflt_mdtm( int fd );
```

DESCRIPTION

This handler function is called when the server receives the FTP command MDTM. The return value of this handler function is the number of seconds that have passed since January 1, 1980. A return value of zero will cause the reply code 213 followed by a space and then the value 19800101000000 (yyyymmddhhmmss) to be sent by the server.

The FTP server assumes that this return value is in UTC (Coordinated Universal Time). If SEC_TIMER is running in local time, the handler should make the necessary time zone adjustment so that the return value is expressed in UTC.

The handler is only recognized if FTP_EXTENSIONS is defined.

PARAMETERS

fd File descriptor for the currently opened file.

RETURN VALUE

The number of seconds that have passed since January 1, 1980. The default handler always returns zero. The number of seconds will be converted to a date and time value of the form yyyymmddhhmmss.

ftp_dflt_delete

```
int ftp_dflt_delete( char *name, int uid, int cwd );
```

DESCRIPTION

The default handler does not support the delete command. It simply returns the error code for an unauthorized user.

The delete handler is only recognized by the server if `FTP_EXTENSIONS` is defined. It is called when the `DELE` command is received. The given file name (possibly relative to `cwd`) should be deleted.

PARAMETERS

name	Pointer to the name of a file.
uid	The currently logged-in user.
cwd	The current directory.

RETURN VALUE

0: File was successfully deleted .
`FTP_ERR_NOTFOUND`: File not found.
`FTP_ERR_NOTAUTH`: Unauthorized user.
`FTP_ERR_BADMODE`: Requested option (2nd parameter) is not supported.
`FTP_ERR_UNAVAIL`: Resource temporarily unavailable.

6.3 API Functions

The API functions described here, initialize and run the FTP server.

`ftp_dflt_is_auth`

```
int ftp_dflt_is_auth( int spec, int options, int uid );
```

DESCRIPTION

Determine amount of access to a file. If the FTP anonymous user has been set, then also checks that. "options" is how to access the file. Currently, this value is ignored. If the anonymous user ID has been set, then files it owns are globally accessible.

Returns whether the user can access it ("owner permission") or if access is because there is an anonymous user ("world permission").

NOTE: This routine only determines accessibility of a name, not whether the user can read and/or write the contents.

PARAMETERS

<code>spec</code>	Handle to SSPEC file (item).
<code>options</code>	How to access O_RDONLY, O_WRONLY or O_RDWR. Currently this value is ignored.
<code>uid</code>	The userID to access as.

RETURN VALUE

0: No access.
1: uid only access.
2: anonymous access (user "anonymous" has been set).

SEE ALSO

`sspec_checkaccess`

ftp_init

```
void ftp_init( FTPhandlers *handlers );
```

DESCRIPTION

Initializes the FTP server. You can optionally specify a set of handlers for controlling what the server presents to the client. This is done with function pointers in the `FTPhandlers` structure. All FTP server instances share the same list of handlers.

The `FTPhandlers` structure is defined as:

```
typedef struct {
    int (*open)(char *name, int options, int uid, int cwd);
    int (*read)(int fd, char *buf, long offset, int len);
    int (*write)(int fd, char *buf, long offset, int len);
    int (*close)(int fd);
    long (*getfilesize)(int fd);

    int (*dirlist)(int item, char *line, int listing, int
        uid, int cwd);

    int (*cd)(int cwd, char *dir, int uid);
    int (*pwd)(int cwd, char *buf);
    [long (*mdtm)(int fd);]
    [int (*delete)(char *name, int uid, int cwd);]
} FTPhandlers;
```

If you always provide all your own handlers, then you can define `FTP_NODEFAULTHANDLER` to eliminate the code for the default handlers. The handlers must be written to the specification described in Section 6.2.2. To use a default handler, leave the field `NULL`. If you pass a `NULL` handlers pointer, then the all default handlers will be used.

The defaults access the server spec list which is set up using the `zserver` functions `sspec_addxmemfile()`, `sauth_adduser()` etc.

PARAMETERS

handlers `NULL` means use default internal file handlers. Otherwise, you must supply a struct of pointers to the various file handlers (`open`, `read`, `write`, `close`, `getfilesize`, `list`). To not override a particular handler, leave it `NULL` in the structure.

LIBRARY

`FTP_SERVER.LIB`

SEE ALSO

`ftp_tick`

`ftp_load_filenames`

```
int ftp_load_filenames(void)
```

DESCRIPTION

This function is used in conjunction with the `FTP_USE_FS2_HANDLERS` macro. It loads the data structure (i.e., the server spec list) that keeps track of the association of filenames to file locations in the file system. The information is loaded from the User block, from the offset given in `FTP_USERBLOCK_OFFSET`.

The function removes any entries from the server spec list that are not FS2 files.

RETURN VALUE

- 0: Success
- 1: Failure (possibly due to the filenames having not yet been saved)

SEE ALSO

`ftp_save_filenames`

`ftp_save_filenames`

```
int ftp_save_filenames(void);
```

DESCRIPTION

This function is used in conjunction with the `FTP_USE_FS2_HANDLERS` macro. This function saves the data structure (i.e., the server spec list) that keeps track of the association of filenames to file locations in the file system. The information is saved to the User block, at the offset given in `FTP_USERBLOCK_OFFSET`.

RETURN VALUE

0: Success.
-1: Failure, the information could not be saved (due to a write error).

SEE ALSO

`ftp_load_filenames`

ftp_set_anonymous

```
int ftp_set_anonymous( int uid );
```

DESCRIPTION

Set the "anonymous" user ID. Resources belonging to this userID may be accessed by any user. A typical use of this function would be

```
ftp_set_anonymous (sauth_adduser("anonymous", "",  
SERVER_FTP) );
```

which defines an "anonymous" login for the FTP server. This only applies to the FTP server. The username "anonymous" is recommended, since most FTP clients use this for hosts that have no account for the user.

PARAMETER

uid	The user ID to use as the anonymous user. This should have been defined using <code>sauth_adduser()</code> . Pass <code>-1</code> to set no anonymous user.
------------	---

RETURN VALUE

Same as the `uid` parameter, except `-1` if `uid` is invalid.

LIBRARY

`FTP_SERVER.LIB`

SEE ALSO

`sauth_adduser`

ftp_shutdown

```
void ftp_shutdown( int bGraceful );
```

DESCRIPTION

Close and cancel all FTP connections. If the server is connected to a client, forces the QUIT state. If the application has called `tcp_reserveport()`, then it must call `tcp_clearreserve()`. For a graceful shutdown, the application must call `tcp_tick()` a few more times.

After the FTP sockets close, the application must call `ftp_init()` to again start the server running.

PARAMETER

bGraceful (boolean) zero to immediately abort all open connections, or non-zero to simulate the QUIT command.

RETURN VALUE

None

LIBRARY

`FTP_SERVER.LIB`

SEE ALSO

`ftp_init`

ftp_tick

```
void ftp_tick( void );
```

DESCRIPTION

Once `ftp_init()` has been called, `ftp_tick()` must be called periodically to run the server. This function is non-blocking.

LIBRARY

FTP_SERVER.LIB

SEE ALSO

`ftp_init`

6.4 Sample FTP Server

This code demonstrates a simple FTP server, using the ftp library. The user "anonymous" may download the file "rabbitA.gif," but not "rabbitF.gif." The user "foo" (with password "bar") may download "rabbitF.gif," but also "rabbitA.gif," since files owned by the anonymous user are world-readable.

File Name: Samples\tcpip\ftp_server.c

```
#define TCPCONFIG 101
#define SSPEC_NO_STATIC //Required for DC 8.50 or later
#include <memmap.h>
#include <dcrtcp.h>
#include <ftp_server.h>
#include <http/pages/rabbit1.gif>
main() {
    int file, user;
    /* Set up the first file and user */
    file = sspec_addxmemfile("rabbitA.gif", rabbit1_gif,
        SERVER_FTP);
    user = sauth_adduser("anonymous", "", SERVER_FTP);
    ftp_set_anonymous(user);
    sspec_setuser(file, user);
    sspec_setuser(sspec_addxmemfile("test1", rabbit1_gif,
        SERVER_FTP), user);
    sspec_setuser(sspec_addxmemfile("test2", rabbit1_gif,
        SERVER_FTP), user);
    /* Set up the second file and user */
    file = sspec_addxmemfile("rabbitF.gif", rabbit1_gif,
        SERVER_FTP);
    user = sauth_adduser("foo", "bar", SERVER_FTP);
    sspec_setuser(file, user);
    sspec_setuser(sspec_addxmemfile("test3", rabbit1_gif,
        SERVER_FTP), user);
    sspec_setuser(sspec_addxmemfile("test4", rabbit1_gif,
        SERVER_FTP), user);
    sock_init();
    ftp_init(NULL); // use default handlers
    tcp_reserveport(FTP_CMDPORT); // Port 21
    while(1) {
        ftp_tick();
    }
}
```


Each user may execute the "dir" or "ls" command to see a listing of the available files. The listing shows only the files that the logged-in user can access.

Notice the definition for TCP_CONFIG. When the value for this macro exceeds 100, a special configuration file is pulled in that will not be overridden by future updates of Dynamic C. In the file CUSTOM_CONFIG.LIB, you may specify any network configuration that suits your purposes. Please see /LIB/TCPIP/TCP_CONFIG.LIB for examples of setting up a library of configuration options.

6.5 Getting Through a Firewall

If a client is behind a firewall, it is incumbent upon the client to request that the server do a passive open on its data port instead of the normal active open. This is so that the client can then do an active open using the passively opened data port of the server, thus getting through the firewall.

Typically the server would not be behind a firewall.

6.6 FTP Server Commands

The following commands are recognized by the FTP server. The reply codes sent in response to these commands are detailed in Section 6.7 on page 301. They are noted here to associate them with the commands that may cause them to be sent.

Table 6.2 Recognized FTP Server Commands

Command	Description	Possible Reply Codes
ABOR	The current data transfer completes before the abort command is read by the server.	226
CDUP	A special case of CWD (Change Working Directory); the parent of the working directory is changed to be the working directory.	250, 431
CWD	Changes working directory.	250, 431
DELE	Delete the specified file.	250, 450, 550
LIST	Displays list of files requested by its argument in ls -l format. This gives extra information about the file.	150, 226, 425
MDTM	Shows the last modification time of the specified file.	213, 250, 450, 550
MODE	Confirms the mode of data transmission. Only stream mode is supported.	200, 504
NLST	Displays list of files requested by its argument, with names only. This allows an application to further process the files.	150, 226, 425
NOOP	Specifies no action except that the server send an OK reply. It does not affect any parameters or previously entered commands.	200

Table 6.2 Recognized FTP Server Commands

Command	Description	Possible Reply Codes
PASS	Password for the user name (sent in clear text). It is accepted only after USER returns code 331	230, 530
PASV	Requests a passive open on a port that is not the default data port. The server responds with the host and port address on which it is listening.	227, 452
PORT	Changes the data port from the default port to the port specified in the command's argument. The argument is the concatenation of a 32-bit internet host address and a 16-bit TCP port address.	200
PWD	Prints the working directory name.	257
QUIT	Closes the control connection. If a data transfer is in progress, the connection will not be closed until it has completed.	221
RETR	Transfers a copy of the file specified in the pathname argument from the server to the client.	150, 226, 425, 550
SIZE	Returns the size of the specified file.	213, 250, 450, 550
STOR	Stores a file from the client onto the server. The file will be overwritten if it already exists at the specified pathname, or it will be created if it does not exist.	150, 226, 250, 425, 450, 452, 550
STRU	Confirms the supported structure of a file. Only file-structure is supported: a continuous stream of data bytes.	200, 504
SYST	Sends the string "RABBIT2000."	215
TYPE	Confirms the transfer type. The types IMAGE (binary), ASCII and Local with 8-bit bytes are all supported and are treated the same.	200, 504
USER	User name to use for authentication.	331, 530

6.7 Reply Codes to FTP Commands

The FTP server replies to all of the commands that it receives. The reply consists of a 3-digit number followed by a space and then a text string explaining the reply. All reply codes sent from the FTP server are listed here.

Table 6.3

Reply Code	Reply Text
150	File status okay; about to open data connection.
200	Command okay.
202	Command not implemented, superfluous at this site.
211	System status, or system help reply.
213	File status
214	Help message. On how to use the server or the meaning of a particular non-standard command. This reply is useful only to the human user.
215	System type.
220	Service ready for new user.
221	Service closing connection.
226	Closing data connection. Requested file action successful (for example, file transfer or file abort).
227	Entering Passive Mode (h1,h2,h3,h4,p1,p2).
230	User logged in, proceed
250	Requested file action okay, completed.
257	"PATHNAME" created.
331	User name okay, need password.
425	Can't open data connection.
450	Requested file action not taken. File unavailable (e.g., file busy).
452	Requested action not taken. Insufficient storage space in system.
502	Command not implemented.
504	Command not implemented for that parameter.
530	Not logged in.
550	Requested action not taken. File unavailable (e.g., file not found, no access).

The text used for the reply codes, may be slightly different than what is shown here. It will be context specific.

7. TFTP Client

TFTP . LIB implements the Trivial File Transfer Protocol (TFTP). This standard protocol (internet RFC783) is a lightweight protocol typically used to transfer bootstrap or configuration files from a server to a client host, such as a diskless workstation. TFTP allows data to be sent in either direction between client and server, using UDP as the underlying transport.

This library fully implements TFTP, but as a client only.

Compared with more capable protocols such as FTP, TFTP:

- has no security or authentication
- is not as fast because of the step-by-step protocol
- uses fewer machine resources.

Because of the lack of authentication, most TFTP servers restrict the set of accessible files to a small number of configuration files in a single directory. For uploading files, servers are usually configured to accept only certain file names that are writable by any user. If these restrictions are acceptable, TFTP has the advantage of requiring very little 'footprint' in the client host.

7.1 BOOTP/DHCP

In conjunction with DHCP/BOOTP and appropriate server configuration, TFTP is often used to download a kernel image to a diskless host. The target TCP/IP board does not currently support loading the BIOS in this way, since the BIOS and application program are written to non-volatile flash memory. However, the downloaded file does not have to be a binary executable - it can be any reasonably small file, such as an application configuration file. TFTP and DHCP/BOOTP can thus be used to administer the configuration of multiple targets from a central server.

Using TFTP with BOOTP/DHCP requires minimal additional effort for the programmer. Just `#define` the symbol `DHCP_USE_TFTP` to an integer representing the maximum allowable boot file size (1-65535). See the description of the variables `_bootpsize`, `_bootpdata` and `_bootperror` in volume 1 of the TCP/IP User's Manual for further details.

7.2 Data Structure for TFTP

This data structure is used to send and receive. The `tftp_state` structure, which is required for many of the API functions in `TFTP.LIB`, may be allocated either in root data memory or in extended memory. This structure is approximately 155 bytes long.

```
typedef struct tftp_state {
    byte state;           // Current state. LSB indicates read (0)
                        // or write(1). Other bits determine
                        // state within this (see below).
    long buf_addr;       // Physical address of buffer
    word buf_len;        // Length of buffer
    word buf_used;       // Amount Tx or Rx from/to buffer
    word next_blk;       // Next expected block #, or next to Tx
    word my_tid;         // UDP port number used by this host
    udp_socket *sock;    // UDP socket to use
    longword rem_ip;     // IP address of remote host
    longword timeout;    // ms timer value for next timeout
    char retry;          // retransmit retry counter
    char flags;          // miscellaneous flags (see below).

    // Following fields not used after initial request has been acknowledged.
    char mode;           // Translation mode (see below).
    char file[129];      // File name on remote host (TFTP server)
                        // - NULL terminated. This field will be
                        // overwritten with a NULL-term error message
                        // from the server if an error occurs.
};
```

The following macros are valid for `tftp_state->mode`.

```
#define TFTP_MODE_NETASCII 0 // ASCII text
#define TFTP_MODE_OCTET 1 // 8-bit binary
#define TFTP_MODE_MAIL 2 // Mail (remote file name is email address,
// e.g., user@host.blob.org)
```

7.3 API Functions

Any of the following functions will require approximately 600-800 bytes of free stack. The data buffer for the file to put or to get is always allocated in `xram` (see `xalloc()`).

TFTP Session

A session can be either a single download (`get`) or upload (`put`). The functions ending with 'x' are versions that use a data structure allocated in extended memory, for applications that are constrained in their use of root data memory.

tftp_init

```
int tftp_init( struct tftp_state *ts );
```

DESCRIPTION

This function prepares for a TFTP session and is called to complete initialization of the TFTP state structure. Before calling this function, some fields in the structure `tftp_state` must be set up as follows:

```
ts->state      = <0 for read, 1 for write>
ts->buf_addr   = <physical address of xmem buffer>
ts->buf_len    = <length of physical buffer, 0-65535>
ts->my_tid     = <UDP port number. Set 0 for default>
ts->sock       = <address of UDP socket (udp_Socket *), or NULL to
                  use DHCP/BOOTP socket>
ts->rem_ip     = <IP address of TFTP server host, or zero to use
                  default BOOTP host>
ts->mode       = <one of the following constants:
                  TFTP_MODE_NETASCII (ASCII text)
                  TFTP_MODE_OCTET (8-bit binary)
                  TFTP_MODE_MAIL (Mail)>
strcpy(ts->file, <remote filename or mail address>)
```

Note that mail mode can only be used to write mail to the TFTP server, and the file name is the e-mail address of the recipient. The e-mail message must be ASCII-encoded and formatted with [RFC822 headers](#). Sending e-mail via TFTP is deprecated. Use SMTP instead since TFTP servers may not implement mail.

PARAMETERS

ts Pointer to `tftp_state`.

RETURN VALUE

0: OK.
-4: Error, default socket in use.

LIBRARY

TFTP.LIB

tftp_initx

```
int tftp_initx( long ts_addr );
```

DESCRIPTION

This function is called to complete initialization of the TFTP state structure, where the structure is possibly stored somewhere other than in the root data space. This is a wrapper function for `tftp_init()`. See that function description for details.

PARAMETERS

ts_addr Physical address of TFTP state (struct `tftp_state`)

RETURN VALUE

0: OK
-1: Error, default socket in use

LIBRARY

TFTP.LIB

tftp_tick

```
int tftp_tick( struct tftp_state *ts );
```

DESCRIPTION

This function is called periodically in order to take the next step in a TFTP process. Appropriate use of this function allows single or multiple transfers to occur without blocking. For multiple concurrent transfers, there must be a unique `tftp_state` structure, and a unique UDP socket, for each transfer in progress. This function calls `sock_tick()`.

PARAMETERS

ts Pointer to TFTP state. This must have been set up using `tftp_init()`, and must be passed to each call of `tftp_tick()` without alteration.

RETURN VALUE

- 1: OK, transfer not yet complete.
- 0: OK, transfer complete
- 1: Error from remote side, transfer terminated. In this case, the `ts_addr->file` field will be overwritten with a NULL-terminated error message from the server.
- 2: Error, could not contact remote host or lost contact.
- 3: Timed out, transfer terminated.
- 4: (not used)
- 5: Transfer complete, but truncated -- buffer too small to receive the complete file.

LIBRARY

TFTP.LIB

tftp_tickx

```
int tftp_tickx( long ts_addr );
```

DESCRIPTION

This function is a wrapper for calling `tftp_tick()`, where the structure is possibly stored somewhere other than in the root data space. See that function description for details.

PARAMETERS

ts_addr Physical address of TFTP state (struct `tftp_state`).

RETURN VALUE

- 1: OK, transfer not yet complete.
- 0: OK, transfer complete
- 1: Error from remote side, transfer terminated. In this case, the `ts_addr->file` field will be overwritten with a NULL-terminated error message from the server.
- 2: Error, could not contact remote host or lost contact.
- 3: Timed out, transfer terminated.
- 4: (not used)
- 5: Transfer complete, but truncated -- buffer too small to receive the complete file.

LIBRARY

TFTP.LIB

tftp_exec

```
int tftp_exec( char put, long buf_addr, word *len, int mode,
              char *host, char *hostfile, udp_Socket *sock );
```

DESCRIPTION

Prepare and execute a complete TFTP session, blocking until complete. This function is a wrapper for `tftp_init()` and `tftp_tick()`. It does not return until the complete file is transferred or an error occurs. Note that approximately 750 bytes of free stack will be required by this function.

PARAMETERS

put	0: get file from remote host; 1: put file to host.
buf_addr	Physical address of data buffer.
len	Length of data buffer. This is both an input and a return parameter. It should be initialized to the buffer length. On return, it will be set to the actual length received (for a get), or unchanged (for a put).
mode	Data representation: 0=NETASCII, 1=OCTET (binary), 2=MAIL.
host	Remote host name, or NULL to use default BOOTP host.
hostfile	Name of file on remote host, or e-mail address for mail.
sock	UDP socket to use, or NULL to re-use BOOTP socket if available.

RETURN VALUE

- 0: OK, transfer complete.
- 1: Error from remote side, transfer terminated. In this case, `ts_addr->file` will be overwritten with a NULL-terminated error message from the server.
- 2: Error, could not contact remote host or lost contact.
- 3: Timed out, transfer terminated
- 4: sock parameter was NULL, but BOOTP socket was unavailable.

LIBRARY

TFTP.LIB

8. SMTP Mail Client

SMTP (Simple Mail Transfer Protocol) is one of the most common ways of sending e-mail. SMTP is a simple text conversation across a TCP/IP connection. The SMTP server usually resides on TCP port 25 waiting for clients to connect. (Define `SMTP_PORT` to override the default port number.)

Sending mail with the Dynamic C SMTP client library is a simple process, demonstrated in the sample program shown in [Section 8.3](#). Dynamic C 9 introduced SMTP authentication, described below in [Section 8.2](#).

8.1 Sample Conversation

The following is a typical listing of mail from the controller (`me@somewhere.com`) to `someone@somewhereelse.com`. The mail server that the controller is talking to is `mail.somehost.com`. The lines that begin with a numeric value are coming from the mail server. The other lines were sent by the controller. More information on the exact specification of SMTP and the meanings of the commands and responses can be found in RFC821 at <http://www.ietf.org>.

```
220 mail.somehost.com ESMTP Service (WorldMail 1.3.122)
ready
HELO 10.10.6.100

250 mail.somewhere.com
MAIL FROM: <me@somewhere.com>

250 MAIL FROM:<me@somewhere.com> OK
RCPT TO: <someone@somewhereelse.com>

250 RCPT TO:<someone@somewhereelse.com> OK
DATA

354 Start mail input; end with <CRLF>.<CRLF>
From: <me@somewhere.com>
To: <someone@somewhereelse.com>
Subject: test mail

test mail
.

250 Mail accepted
QUIT

221 mail.somehost.com QUIT
```

You can see a listing of the conversation between your controller and the mail server by defining the `SMTP_DEBUG` macro at the top of your program. Note that there must be a blank line after the line “Subject: test mail.”

8.2 SMTP Authentication

In most situations, Internet access is provided by an Internet Service Provider (ISP). Usually the ISP runs an email server that will accept emails without authentication from customers that are within their network. Users outside of their network are not allowed to send email through their servers because the mail server would quickly become a gateway for spam. With more people on the go with laptops, SMTP authentication allows them to send email through a trusted server without being directly on the network.

An informative tutorial on SMTP authentication is available at:

www.fehcom.de/qmail/smtppauth.html

Default behavior prior to Dynamic C 9.21 was for the login to fail if authentication failed. With Dynamic C 9.21 the SMTP library will fall back on unauthenticated login if authentication fails. To restore the old behavior when using Dynamic C 9.21, define the macro `SMTP_AUTH_FAIL_IF_NO_AUTH`.

Three methods of authentication are recognized by the implementation of an SMTP client.

AUTH PLAIN

The client sends "AUTH PLAIN <token>" where <token> is the Base64-encoded string "username\0username\0password" that will look something like:

```
AUTH PLAIN dGVzdAB0ZXN0AHRlc3RwYXNz
```

The server responds with a message indicating whether authentication was successful.

AUTH LOGIN

Client sends "AUTH LOGIN" command; server responds with "334 VXNlcm5hbWU6" (Base64-encoded "Username:"); client responds with its Base64-encoded username; server responds with "334 UGFzc3dvcmQ6"; client responds with its Base64-encoded password. At this point, the server should respond with a message indicating whether authentication was successful. This method is only slightly more complex than AUTH PLAIN.

AUTH CRAM-MD5

Client sends "AUTH CRAM-MD5"; server responds with "334 <challenge>" where <challenge> is a unique Base64-encoded challenge string (for example, "<4994.1088035610@zworld.com>").

The client generates a digest using the following MD5 hashing algorithm (where password is null-padded to a length of 64 bytes, ipad is 0x36 repeated 64 times and opad is 0x5C repeated 64 times):

```
digest = MD5((password XOR opad), MD5((password XOR  
ipad), challenge))
```

The client responds with the string "<username> <response>" Base64-encoded; <username> is in plaintext, and <response> is the 16-byte digest in hex form. This method is the most secure, since someone sniffing the connection would be unable to determine the cleartext password used to authenticate.

8.3 Sample Sending of an E-mail

This program, `smtp.c`, sends an e-mail. To have the client query the server for authentication, define the macro `USE_SMTP_AUTH` and call `smtp_setauth()` before calling `smtp_sendmail()` (or `smtp_sendmailxmem()`). If the mail server does not support authentication, either do not define `USE_SMTP_AUTH` or pass empty strings (“”) as the parameters to `smtp_setauth()`.

Program Name: `Samples\tcpip\smtp\smtp.c`

```
#define TCPCONFIG 1 // pick network configuration
#define FROM "myaddress@mydomain.com"
#define TO "myaddress@mydomain.com"
#define SUBJECT "You've got mail!"
#define BODY "Visit the Rabbit Semiconductor web site.\r\n"
/* SMTP_SERVER identifies the mail server. This can be name or IP address. */
#define SMTP_SERVER "mymailserver.mydomain.com"

#define USE_SMTP_AUTH

#memmap xmem
#use dcrtcp.lib
#use smtp.lib

main() {
    sock_init();

    while (ifpending(IF_DEFAULT) == IF_COMING_UP) {
        tcp_tick(NULL);
    }

#ifdef USE_SMTP_AUTH
    smtp_setauth("myusername", "mypassword");
#endif

    smtp_sendmail(TO, FROM, SUBJECT, BODY);

    while (smtp_maintick() == SMTP_PENDING)
        continue;

    if (smtp_status() == SMTP_SUCCESS)
        printf("Message sent\n");
    else
        printf("Error sending message\n");
}
```

8.4 Configuration Macros

The SMTP client is configured by using compiler macros.

SMTP_AUTH_FAIL_IF_NO_AUTH

Defaults to undefined. This macro was introduced in Dynamic C 9.21. If it is defined, the login will fail if authentication fails. Otherwise, the library will fall back on an unauthenticated login if authentication fails. Prior to Dynamic C 9.21, the login failed if authentication failed, so the macro is restoring that behavior.

SMTP_DEBUG

This macro tells the SMTP code to log events to the STDIO window in Dynamic C. This provides a convenient way of troubleshooting an e-mail problem.

SMTP_DOMAIN

This macro defines the text to be sent with the HELO client command. Many mail servers ignore the information supplied with the HELO, but some e-mail servers require the fully qualified name in this field (i.e., somemachine.somedomain.com). If you have problems with e-mail being rejected by the server, turn on `SMTP_DEBUG`. If it is giving an error message after the HELO line, talk to the administrator of the machine for the appropriate value to place in `SMTP_DOMAIN`. If you do not define this macro, it will default to `MY_IP_ADDRESS`.

```
#define SMTP_DOMAIN "somemachine.somedomain.com"
```

SMTP_MAX_DATALEN

Defaults to 256. Maximum buffer size for server responses and short client requests.

SMTP_MAX_PASSWORDLEN

Defaults to 16. Maximum length of the password used in authentication.

SMTP_MAX_USERNAMELEN

Defaults to 64. Maximum length of the user name used in authentication.

SMTP_MAX_SERVERLEN

Defaults to `MAX_STRING`, which defaults to 50. Maximum length of mail server name.

SMTP_SERVER

This macro defines the mail server that will relay the controller's mail. This server must be configured to relay mail for your controller. You can either place a fully qualified domain name or an IP address in this field.

```
#define SMTP_SERVER "mail.mydomain.com"  
or  
#define SMTP_SERVER "10.10.6.19"
```


SMTP_TIMEOUT

This macro tells the SMTP code how long in seconds to try to send the e-mail before timing out. It defaults to 20 seconds.

```
#define SMTP_TIMEOUT 10
```

USE_SMTP_AUTH

Define this macro to enable SMTP authentication.

8.5 API Functions

The user-callable functions described in this section are found in the Dynamic C library `Lib\tcpip\smtp.lib`.

`smtp_data_handler`

```
void smtp_data_handler( int (*dhnd)(), void * dhnd_data, word
    opts );
```

DESCRIPTION

Sets a data handler for generating mail message content. This function should be called after calling `smtp_sendmail()` etc. It overrides any message parameter set by the `smtp_sendmail()` call, since the message is generated dynamically by the callback function.

Note: you can use the same data handler as used for the FTP library (see the `ftp_data_handler()` description). The flags values are numerically equivalent to those of the same meaning for `ftp_data_handler()`. The SMTP data handler is only used to generate data, not receive it.

The handler is a function that must be coded according to the following prototype:

```
int my_handler(char *data, int len, longword offset,
    int flags, void *dhnd_data);
```

The data handler function must be called with the following parameters:

data	Pointer to a data buffer
len	The length of the above data buffer. This parameter is set to <code>SMTP_MAX_DATALEN</code> (256) by default. You can override that macro to allow larger "chunks".
offset	The byte number relative to the first byte of the entire message stream. This is useful for data handler functions that do not wish to keep track of the current state of the data source.
flags	Contains an indicator of the current operation: <code>SMTPDH_OUT</code> : data is to be filled with the next data to send to the mail server. The maximum allowable chunk of data is specified by 'len'. The data must not contain the sequence <code><CRLF>.<CRLF></code> since that will confuse the process. <code>SMTPDH_ABORT</code> : end of data; error encountered during SMTP operation. The mail was probably not delivered.
dhnd_data	The pointer that was passed to <code>ftp_data_handler()</code> .

PARAMETERS

<code>dhnd</code>	Pointer to data handler function, or <code>NULL</code> to remove the current data handler.
<code>dhnd_data</code>	A pointer that is passed to the data handler function. This may be used to point to any further data required by the data handler such as an open file descriptor.
<code>opts</code>	Options word (currently reserved, set to zero).

RETURN VALUE

The return value from this function should be the actual number of bytes placed in the data buffer, or -1 to abort. If 0 is returned, then this is considered to be the end of data. You can write up to and including "len" bytes into the buffer, but at least one byte must be written otherwise it is assumed that no more data is following.

For `SMTPPDH_ABORT`, the return code is ignored.

SEE ALSO

`smtp_sendmail`, `smtp_sendmailxmem`, `smtp_maintick`

EXAMPLE

The program `Samples/tcpip/smtp/smtp_dh.c` makes use of this function.

smtp_maintick

```
int smtp_maintick( void );
```

DESCRIPTION

Repetitively call this function until e-mail is completely sent.

RETURN VALUE

SMTP_SUCCESS - e-mail sent.

SMTP_PENDING - e-mail not sent yet call smtp_maintick again.

SMTP_TIME - e-mail not sent within SMTP_TIMEOUT seconds.

SMTP_UNEXPECTED - received an invalid response from SMTP server.

SMTP_DNSERROR - cannot resolve server name

SMTP_ABORTED - transaction aborted (by data handler)

If using SMTP AUTH, the following values are also possible:

SMTP_AUTH_UNAVAILABLE - unable to attempt authentication|

SMTP_AUTH_FAILED - attempts to authenticate failed

LIBRARY

SMTP.LIB

SEE ALSO

smtp_sendmail, smtp_status

smtp_sendmail

```
void smtp_sendmail( char *to, char *from, char *subject, char
    *message );
```

DESCRIPTION

Start an e-mail being sent. This function is intended to be used for short messages that are entirely constructed prior to being sent.

If you have previously installed a data handler via `smtp_data_handler()`, then you must call `smtp_data_handler()` with a NULL data handler, otherwise this message will not get sent.

NOTE: The strings pointed to by the parameters must not be changed until the entire process is completed. Also, if the first character of any line of the message is a period (`.`), then this character will be deleted as part of normal mail processing. Thus, to actually send a line starting with `.`, you must start the line with `..` i.e. double up an initial period.

PARAMETERS

to	String containing the e-mail address of the destination. Maximum of 192 characters. Currently, only one recipient is supported.
from	String containing the e-mail address of the source. Maximum of 192 characters for a return address. If no return should be sent by receiver, then pass an empty string (<code>""</code>).
subject	String containing the subject of the message. This may be NULL in which case no subject line will be sent. This string may also contain embedded <code>\r\n</code> sequences so that additional mail header lines may be inserted. The length of this string is unlimited.
message	String containing the message. (This string must <i>not</i> contain the byte sequence <code>"\r\n\r\n"</code> (CRLF.CRLF), as this is used to mark the end of the e-mail, and will be appended to the e-mail automatically.) This message must be null terminated, and is only allowed to contain 7-bit characters. You can pass NULL if a data handler is to be used to generate the message.

RETURN VALUE

None.

SEE ALSO

`smtp_maintick`, `smtp_status`, `smtp_sendmailxmem`

smtp_sendmailxmem

```
void smtp_sendmailxmem( char *to, char *from, char *subject,
    long message, long messagelen );
```

DESCRIPTION

Start an e-mail being sent. This is intended for moderately long, fixed messages that are stored in extended memory (e.g., via #ximport'ed file).

See `smtp_sendmail()` for more details.

PARAMETERS

to	String containing the e-mail address of the destination.
from	String containing the e-mail address of the source.
subject	String containing the subject of the message.
message	Physical address in xmem containing the message. (The message must NOT contain the byte sequence "\r\n.\r\n" (CRLF.CRLF), as this is used to mark the end of the e-mail, and will be appended to the e-mail automatically.)
messagelen	Length of the message in xmem.

RETURN VALUE

None

LIBRARY

SMTP.LIB

SEE ALSO

`smtp_maintick`, `smtp_status`, `smtp_sendmail`

smtp_setauth

```
int smtp_setauth( char* username, char* password );
```

DESCRIPTION

Sets the username and password to use for SMTP AUTH (Authentication). You must `#define USE_SMTP_AUTH` in your program if you want to use SMTP AUTH on your outbound connections. To disable SMTP authentication, set both `username` and `password` to "" (empty strings).

PARAMETERS

username	This is copied into the SMTP state structure. Note that some SMTP servers require a full email address while others just want a username.
password	This is copied into the SMTP state structure.

RETURN VALUE

SMTP_OK: server name was set successfully
SMTP_USERNAMETOOLONG: the username was too long
SMTP_PASSWORDTOOLONG: the username was too long

SEE ALSO

`smtp_sendmail`, `smtp_maintick`

smtp_setserver

```
int smtp_setserver( char* server );
```

DESCRIPTION

Sets the SMTP server. This value overrides SMTP_SERVER and the results of any previous call to smtp_setserver_ip().

PARAMETER

server Server name string. This is copied into the SMTP state structure. This name is not resolved to an IP address until you start calling smtp_mailtick().

RETURN VALUE

SMTP_OK: Server name was set successfully
SMTP_NAMETOOLONG: The server name was too long

SEE ALSO

smtp_sendmail, smtp_setserver_ip, smtp_mailtick

smtp_setserver_ip

```
int smtp_setserver_ip( longword server );
```

DESCRIPTION

Sets the SMTP server. This value overrides the value set by smtp_setserver(), and is used when the IP address of the mail server is known.

PARAMETER

server Server IP address.

RETURN VALUE

SMTP_OK: server IP was set successfully

SEE ALSO

smtp_sendmail, smtp_setserver, smtp_mailtick

smtp_status

```
int smtp_status( void );
```

DESCRIPTION

Return the status of the last e-mail processed.

RETURN VALUE

SMTP_SUCCESS - e-mail sent.

SMTP_PENDING - e-mail not sent yet call smtp_maintick again.

SMTP_TIME - e-mail not sent within SMTP_TIMEOUT seconds.

SMTP_UNEXPECTED - received an invalid response from SMTP server.

LIBRARY

SMTP.LIB

9. POP3 Client

Post Office Protocol version 3 (POP3) is probably the most common way of retrieving e-mail from a remote server. Most e-mail programs, such as Eudora, MS-Outlook, and Netscape's e-mail client, use POP3. The protocol is a fairly simple text-based chat across a TCP socket, normally using TCP port 110.

There are two ways of using `POP3.LIB`. The first method provides a raw dump of the incoming e-mail. This includes all of the header information that is sent with the e-mail, which, while sometimes useful, may be more information than is needed. The second method provides a parsed version of the e-mail, with the sender, recipient, subject line, and body text separated out.

In both methods, each line of e-mail has CRLF stripped from it and `'\0'` appended to it.

9.1 Configuration

The POP3 client can be configured through the following macros:

POP_BUFFER_SIZE

This will set the buffer size for `POP_PARSE_EXTRA` in bytes. These are the buffers that hold the sender, recipient and subject of the e-mail. `POP_BUFFER_SIZE` defaults to 64 bytes.

POP_DEBUG

This will turn on debug information. It will show the actual conversation between the device and the remote mail server, as well as other useful information.

POP_NODELETE

This will stop the POP3 library from removing messages from the remote server as they are read. By default, the messages are deleted to save storage space on the remote mail server.

POP_PARSE_EXTRA

This will enable the second mode, creating a parsed version of the e-mail as mentioned above. The POP3 library parses the incoming mail more fully to provide the Sender, Recipient, Subject, and Body fields as separate items to the call-back function.

9.2 Steps to Receive E-mail.

1. `pop3_init()` is called to provide the POP3 library with a call-back function. This call-back will be used to provide you the incoming data. This function is usually called once.
2. `pop3_getmail()` is called to start the e-mail being received, and to provide the library with e-mail account information.
3. `pop3_tick()` is called as long as it returns `POP_PENDING`, to actually run the library. The library will call the function you provided `pop3_init()` several times to give you the e-mail.

9.3 Call-Back Function

There are two types of call-back functions, which are described here.

9.3.1 Normal call-back

When not using `POP_PARSE_EXTRA`, you need to provide a function with the following prototype:

```
int storemail(int number, char *buf, int size);
```

The parameter `number` is the number of the e-mail being transferred, usually 1 for the first, 2 for the second, but not necessarily. The numbers are only guaranteed to be unique between all e-mails transferred.

The `buf` parameter is the text buffer containing one line of the incoming e-mail. This must be copied out immediately, as the buffer will be different when the next line comes in, and your call-back is called again. `size` is the number of bytes in `buf`.

The sample program `Samples\tcpip\pop3\ pop.c` provides an example of this style of call-back.

9.3.2 POP_PARSE_EXTRA call-back

If `POP_PARSE_EXTRA` is defined, you need to provide a call-back function with the following prototype:

```
int storemail(int number, char *to, char *from, char *subject,  
             char *body, int size);
```

`number`, `body`, and `size` are the same as before.

`to` has the e-mail address of who this e-mail was sent to.

`from` has the e-mail address of who sent this e-mail.

`subject` has the subject line of the e-mail.

These new fields should be used only the first time your call-back is called with a new `number` field. In subsequent calls, these fields are not guaranteed to have accurate information.

See `parse_extra.c` in Section 9.5 for an example of this type of call-back.

9.4 API Functions

pop3_init

```
int pop3_init(int (*storemail)());
```

DESCRIPTION

This function must be called before any other POP3 function is called. It will set the call-back function where the incoming e-mail will be passed to. This probably should only be called once.

PARAMETERS

storemail A function pointer to the call-back function.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

POP3.LIB

pop3_getmail

```
int pop3_getmail(char *username, char *password, long server);
```

DESCRIPTION

This function will initiate receiving e-mail (a POP3 request to a remote e-mail server).
IMPORTANT NOTE - the buffers for `username` and `password` must NOT change until `pop3_tick()` returns something besides `POP_PENDING`. These values are not saved internally, and depend on the buffers not changing.

PARAMETERS

username	The username of the account to access.
password	The password of the account to access.
server	The IP address of the server to connect to, as returned from <code>resolve()</code> .

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

POP3.LIB

pop3_tick

```
int pop3_tick(void)
```

DESCRIPTION

A standard tick function, to run the daemon. Continue to call it as long as it returns POP_PENDING.

RETURN VALUE

POP_PENDING: Transfer is not done; call pop3_tick again.
POP_SUCCESS: All e-mails were received successfully.
POP_ERROR: Unknown error occurred.
POP_TIME: Session timed-out. Try again, or use POP_TIMEOUT to increase the time-out length.

LIBRARY

POP3.LIB

9.5 Sample Receiving of E-mail

This program connects to a POP3 server and downloads e-mail from it.

Program Name: Samples\tcpip\pop3\parse_extra.c

```
#define TCPCONFIG 1
#define POP_HOST mail.domain.com" // Name of your POP3 server
#define POP_USER "myname" // Username for POP3 account
#define POP_PASS "secret" // Password for POP3 account
#define POP_PARSE_EXTRA
#memmap xmem
#use "dcrtcp.lib"
#use "pop3.lib"
int n;

int storemsg(int num, char *to, char *from, char *subject,
char *body, int len){
    #GLOBAL_INIT{n = -1;}
    if(n != num) {
        n = num;
        printf("RECEIVING MESSAGE <%d>\n", n);
        printf("\tFrom: %s\n", from);
        printf("\tTo: %s\n", to);
        printf("\tSubject: %s\n", subject);
    }
    printf("MSG_DATA> '%s'\n", body);
    return 0;
}

main(){
    static long address;
    static int ret;

    sock_init();
    pop3_init(storemsg); //set up call-back
    printf("Resolving name...\n");
    address = resolve(POP_HOST);
    printf("Calling pop3_getmail()...\n");
    pop3_getmail(POP_USER, POP_PASS, address); // Request to server
    printf("Entering pop3_tick()...\n");
    while((ret = pop3_tick()) == POP_PENDING)
        continue;
    if(ret == POP_SUCCESS)
        printf("POP was successful!\n");
    if(ret == POP_TIME)
        printf("POP timed out!\n");
    if(ret == POP_ERROR)
        printf("POP returned a general error!\n");
    printf("All done!\n");
}
```


9.5.1 Sample Conversation

The following is an example POP3 session from the specification in RFC1939. For more information see:

<http://www.rfc-editor.org/rfc/std/std53.txt>

In the following example, lines starting with 'S:' are from the server, and lines starting with 'C:' are from the client.

```
S: <wait for connection on TCP port 110>
C: <open connection>
S: +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>
C: APOP mrose c4c9334bac560ecc979e58001b3e22fb
S: +OK mrose's maildrop has 2 messages (320 octets)
C: STAT
S: +OK 2 320
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S: <the POP3 server sends message 1>
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S: <the POP3 server sends message 2>
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
C: <close connection>
S: <wait for next connection>
```

For debugging purposes, you can observe this conversation by defining POP_DEBUG at the top of your program.

10. Telnet

The library `Vserial.lib` implements the telecommunications network interface known as telnet. The implementation is a telnet-to-serial and serial-to-telnet gateway. This chapter is divided into two parts. The first part describes the library from Dynamic C version 7.05 and later. The second part describes the library prior to 7.05.

10.1 Telnet (Dynamic C 7.05 and later)

This implementation is more general than the previous one. Any of the four serial ports can be used and other I/O streams can be added. Multiple connections are supported by the use of unique gateway identifiers.

10.1.1 Setup

To use a serial port, the circular buffers must be initialized. For instance, if serial port A is used by an application, then the following macros must be defined in the program:

```
#define AINBUFSIZE 31
#define AOUTBUFSIZE 31
```

It might be necessary to have bigger buffers for some applications.

10.1.1.1 Low-Level Serial Routines

A table to hold the low-level I/O routines must be defined as type `VSerialSpec`.

```
typedef struct {
    int id; // unique ID to match with calls to listen/open
    int (*open)(); // serial port routines, or
    int (*close)(); // serial port compatible routines.
    int (*tick)();
    int (*rdUsed)();
    int (*wrFree)();
    int (*read)();
    int (*write)();
} VSerialSpec;
```

For each serial port A, B, C and D, there is a pre-defined macro in `VSERIAL.LIB`:

```
#define VSERIAL_PORTA(id) { (id), serAopen, serAclose, NULL,
    serArdUsed, serAwrFree, serAread, serAwrite }
```

The parameter passed to `VSERIAL_PORTA` is the unique gateway identifier mentioned earlier. This value is chosen by the developer when entries are made to the array of type `VSerialSpec` (also known as the spec table).

Dynamic C 9.21 includes support for serial ports E and F on all Rabbit 3000 based boards.

10.1.1.2 Configuration Macros

VSERIAL_DEBUG

Turns on debug messages.

VSERIAL_NUM_GATEWAYS

The number of telnet sessions must be defined and must match the number of entries in the spec table.

10.1.2 API Functions (Dynamic C 7.05 and later)

The following functions compose the latest telnet API. A sample program demonstrating their use is available at `Samples\tcpip\telnet\vserial.c`.

`vserial_close`

```
int vserial_close( int id );
```

DESCRIPTION

Closes the specified gateway. This will not only terminate any network activity, but will also close the serial port.

PARAMETERS

`id` ID of the gateway to change, as specified in the spec table.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

VSERIAL.LIB

`vserial_init`

```
int vserial_init ( void );
```

DESCRIPTION

Initializes the daemon and parses the spec table.

RETURN VALUE

0: Success;
1: Failure.

LIBRARY

VSERIAL.LIB

vserial_keealive

```
int vserial_keealive ( int id, long timeout );
```

DESCRIPTION

This function sets the keealive timer to generate TCP keealives after **timeout** periods of inactivity. This helps detect if the connection has gone bad.

Keealives should be used at the application level, but if that is not possible, then **timeout** should be set so as to not overload the network. The standard timeout is two hours, and should be set sooner than that only for a Very Good Reason.

PARAMETERS

id	Unique gateway identifier.
timeout	Number of seconds of inactivity allowed before a TCP keealive is sent. A value of 0 shuts off keealives.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

VSERIAL.LIB

vserial_listen

```
int vserial_listen( int id, long baud, int port, long
    remote_host, int flags );
```

DESCRIPTION

Listens on the specified port for a telnet connection. The gateway process is started when a connection request is received. On disconnect, re-listen happens automatically.

PARAMETERS

id	ID of the gateway to change, as specified in the spec table.
baud	The parameter to send to the <code>open ()</code> serial port command; it's usually the baud rate.
port	The local TCP port to listen on.
remote_host	The remote host from whom to accept connections, or 0 to accept a connection from anybody.
flags	Option flags for this gateway. Currently the only valid bit flags are <code>VSERIAL_COOKED</code> to strip out telnet control codes, or 0 to leave it a raw data link.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

`VSERIAL.LIB`

vserial_open

```
int vserial_open( int id, long baud, int port, long remote_host,  
                 int flags, long retry );
```

DESCRIPTION

Opens a connection to a remote host and maintains it, starting the gateway process.

PARAMETERS

id	ID of the gateway to change, as specified in the spec table.
baud	The parameter to send to the <code>open ()</code> serial port command; it's usually the baud rate.
port	The TCP port on the remote host to connect to.
remote_host	The remote host to connect to.
flags	Option flags for this gateway. Currently the only valid bit flags are <code>VSERIAL_COOKED</code> to strip out telnet control codes, or 0 to leave it a raw data link.
retry	The retry time-out, in seconds. When a connection fails, or if the connection was refused, we will wait this number of seconds before retrying.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

`VSERIAL.LIB`

vserial_tick

```
int vserial_tick( void );
```

DESCRIPTION

Runs the telnet daemon - must be called periodically.

RETURN VALUE

0: Success;

1: Failure.

But call it periodically no matter the return value! An error message can be seen when 1 is returned if you define `VSERIAL_DEBUG` at the top of your program.

LIBRARY

`VSERIAL.LIB`

10.2 Telnet (pre-Dynamic C 7.05)

The API available for telnet changed with Dynamic C version 7.05. This is the old API

10.2.1 Configuration Macros

SERIAL_PORT_SPEED

The baud rate of the serial port. Defaults to 115,200 bps.

TELNET_COOKED

`#define` this to have telnet control codes stripped out of the data stream.

This is useful if you are actually telnetting to the device from another box. It should not be defined if you are using two devices as a transparent bridge over the Ethernet.

10.2.2 API Functions

telnet_init

```
int telnet_init( int which, longword addy, int port );
```

DESCRIPTION

Initializes the connection. This function must not be called by an application program starting with Dynamic C 7.05.

PARAMETERS

which	Is one of the following: TELNET_LISTEN—Listens on a port for incoming connections. TELNET_RECONNECT—Connects to a remote host, and reconnects if the connection dies. TELNET_CONNECT—Connects to a remote host, and terminates if the connection dies.
addy	IP address of the remote host, or 0 if we are listening.
port	Port to bind to if we are listening, or the port of the remote host to connect to.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

VSERIAL.LIB

telnet_tick

```
int telnet_tick( void );
```

DESCRIPTION

Must be called periodically to run the daemon.

RETURN VALUE

0: Success (call it again);
1: Failure; TELNET_CONNECT died, or a fatal error occurred.

LIBRARY

VSERIAL.LIB

telnet_close

```
void telnet_close( void );
```

DESCRIPTION

Terminates any connections currently open, and shuts down the daemon.

LIBRARY

VSERIAL.LIB

10.2.3 An Example Telnet Server

The following code implements a telnet server. It listens on well-known port 23 for a connection request. Data is passed transparently via serial port C.

```
#define MY_IP_ADDRESS "10.10.6.105"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.19"
#define MY_NAMESERVER "10.10.6.19"

#define SERIAL_PORT_SPEED 115200 // Set serial port speed.
#undef TELNET_COOKED // This is a raw data port.

#mmap xmem
#use "dcrtcp.lib"
#use "vserial.lib"

#define SERVER_PORT 0 // Defaults to port 23.
main() {
    sock_init(); // Initialize stack.
    telnet_init(TELNET_LISTEN, 0, SERVER_PORT); // Initialize server
    while(!telnet_tick()) // Run server; this is non-blocking
        continue;
    telnet_close(); // Error, close telnet connection
}
```

10.2.4 An Example Telnet Client

This sample code implements a client that can connect to the above telnet server.

```
#define MY_IP_ADDRESS "10.10.6.106"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.19"
#define MY_NAMESERVER "10.10.6.19"
#define SERIAL_PORT_SPEED 115200 // Must match server.
#undef TELNET_COOKED
#mmap xmem
#use "dcrtcp.lib"
#use "vserial.lib"
#define SERVER_PORT 0 // Defaults to port 23; must match server.
#define REM_HOST "10.10.6.19" // Remote IP to connect to.
main() {
    sock_init(); // Initialize the stack
    // Tell the server to connect and reconnect if the connection is lost
    telnet_init(TELNET_RECONNECT, resolve(REM_HOST), SERVER_PORT);
    while(!telnet_tick()) // Run client; this is non-blocking
        continue;
    telnet_close(); // Error, close telnet connection
}
```

11. General Purpose Console

The library, `zconsole.lib`, implements a serial-based console that we call Zconsole. It can be used to:

- Configure a board.
- Upload and download web pages.
- Change web page variables without re-uploading the page.
- Send e-mail.
- Calls subsystem initialization for ARP, TCP, UDP and DNS (if applicable).

11.1 Zconsole Features

Recognizing that embedded control systems are wide-ranging in their requirements, `zconsole.lib` was designed with flexibility and extensibility in mind. Designers can choose the available functionality they want and leave the rest alone. Zconsole includes:

- Login name and password protection.
- Default and custom Zconsole commands.
- Default and custom error messages.
- Help text for Zconsole commands, including custom commands
- Multiple I/O streams that can be used simultaneously.
- A fail-safe backup system for configuration data.

11.1.1 File System Requirement

Prior to Dynamic C 7.30, an application program using Zconsole must include the lines:

```
#use "filesystem.lib" // If using the improved file system available with
                        // Dynamic C 7.05, substitute "fs2.lib" for "filesystem.lib"
#use "zconsole.lib"
```

Using the file system is no longer necessary.

11.1.2 TCP/IP and Zconsole

Dynamic C TCP/IP functionality may be used by a Zconsole application program by including the statement

```
#use "dcrtcp.lib"
```

in the program. Other TCP/IP protocols may be added with `#use` statements of the appropriate libraries.

11.2 Login Name and Password

There is a sample program, `Samples\tcpip\LOGINCONSOLE.C`, that demonstrates the use of the login name/password functionality for Zconsole. Zconsole command functions: `con_loginname()`, `con_loginpassword()` and `con_logout()` are described in Section 11.4.1.1 starting on page 347. The structure that saves the name and password information can be backed up using the backup macro `CONSOLE_BACKUP_LOGIN`. Please see Section 11.7 starting on page 368 for details on the backup system.

11.3 Zconsole Commands and Messages

Zconsole is a command-driven application. A command is issued either at the keyboard using a terminal emulator or a command is generated and sent from an attached machine. Zconsole carries out the command, and either the message “OK” \r\n is returned, or an error is returned in the form of:

```
ERROR XXXX This is an error message.\r\n
```

Note that the carriage return and new line characters (\r\n) are always returned by Zconsole whether the command completed successfully or not.

11.3.1 Zconsole Command Data Structure

The command system is set up at compile time with an array of `ConsoleCommand` structures. There is one array entry for each command recognized by Zconsole.

```
typedef struct {
    char *command;
    int (*cmdfunc)();
    long helptext;
} ConsoleCommand
```

command

This field is a string like the following: “SET MAIL FROM.” That is, each word of the command is separated by a space. The case of the command does not matter. Entering this string is how the command is invoked.

cmdfunc

This field is a function pointer to the function that implements the command. The functions that come with Zconsole are listed in Section 11.4.1.1 on page 347.

helptext

This field points to a text file. The text file contains help information for the associated command. When `HELP COMMAND` is entered, this text file (the help information for `COMMAND`) will be printed to Zconsole. The help text comes from `#ximported` text files.

11.3.1.1 Help Text for General Cases

There are two cases in `Zconsole.lib` where help text is needed, but is not associated with a particular command. It is still necessary to allocate a `ConsoleCommand` structure to access the help text. The first case is the help overview given when `HELP` is entered by itself. The `command` field should be "" and the `cmdfunc` field should be `NULL`.

```
{ "", NULL, help_txt },
```

The second case is `HELP SET`. This is an overview of the family of `SET` commands, i.e., commands that set configuration values. For `HELP SET`, the `command` field should be "SET" and the `cmdfunc` field should be `NULL`.

```
{ "SET", NULL, help_set_txt },
```

This second case illustrates the general case of displaying help for a family of commands. The family name can not be the name of a command.

11.4 Zconsole Command Array

An array of `ConsoleCommand` structures must be defined in an application program as a constant global variable named `console_commands []`. All commands available at the console, those provided in `Zconsole.lib` and custom commands, must have an entry in this array.

11.4.1 Zconsole Commands

The following is an example of a list of commands that may be defined in a Zconsole application. When the command name {i.e., the string in the `command` field) is received by the console, the function pointed to in the `cmdfunc` field is executed. When the console receives the command, **HELP** <command name>, the text file located at physical address `helptext` will be displayed.

```
const ConsoleCommand console_commands[] =
{
  { "HELLO WORLD", hello_world, 0 },
  { "ECHO", con_echo, help_echo_txt },
  { "HELP", con_help, help_help_txt },
  { "", NULL, help_txt },
  { "SET", NULL, help_set_txt },
  { "SET PARAM", con_set_param, 0 },
  { "SET IP", con_set_ip, help_set_txt },
  { "SET NETMASK", con_set_netmask, help_set_txt },
  { "SET GATEWAY", con_set_gateway, help_set_txt },
  { "SET NAMESERVER", con_set_nameserver, help_set_txt },
  { "SET MAIL", NULL, help_set_mail_txt },
  { "SET MAIL SERVER", con_set_mail_server, help_set_mail_server_txt },
  { "SET MAIL FROM", con_set_mail_from, help_set_mail_from_txt },
  { "SHOW", con_show, help_show_txt },
  { "PUT", con_put, help_put_txt },
  { "GET", con_get, help_get_txt },
  { "DELETE", con_delete, help_delete_txt },
  { "LIST", NULL, help_list_txt },
  { "LIST FILES", con_list_files, help_list_txt },
  { "LIST VARIABLES", con_list_variables, help_list_txt },
  { "CREATEV", con_createv, help_createv_txt },
  { "PUTV", con_putv, help_putv_txt },
  { "GETV", con_getv, help_getv_txt },
  { "MAIL", con_mail, help_mail_txt },
  { "RESET FILES", con_reset_files, 0 },
  { "RESET VARIABLES", con_reset_variables, help_reset_variables }
};
```


11.4.1.1 Default Command Functions

The following functions are provided in `Zconsole.lib`. Each one takes a pointer to a `ConsoleState` structure as its only parameter, following the prototype for custom functions described in Section 11.4.1.2 on page 352. Each of these functions return 0 when it has more processing to do (and thus will be called again), 1 for successful completion of its task, and -1 to report an error.

Parameters needed by the commands using these functions are passed on the command line.

con_add_nameserver()

This function adds a name server to the list of name servers (unlike `con_set_nameserver()` that clears the list of name servers and adds one name server). A command that use this function takes one parameter: the IP address of the name server in dotted quad notation.

con_createv()

This function creates a variable that can be used with SSI commands in SHTML files. Certain SSI commands can be replaced by the value of this variable, so that a web page can be dynamically altered without re-uploading the entire page. Note, however, that the value of the variable is not preserved across power cycles, although the variable entry is still preserved. That is, the value of the variable may change after a power cycle. It can be changed again, though, with a `putv` command. It works in the following fashion (if the command is called "CREATEV"):

```
usage: "createv <varname> <vartype> <format> <value> [strlen] "
```

A web variable that can be referenced within web files is created.

`<varname>` is the name of the variable

`<vartype>` is the type of the variable (INT8, INT16, INT32, FLOAT32, or STRING)

`<format>` is the printf-style format specifier for outputting the variable (such as "%d")

`<value>` is the value to assign the variable.

`[strlen]` is only used if the variable is of type STRING. It is used to give the maximum length of the string.

con_delete()

This function deletes a file from the file system. A command that uses this function takes one parameter: the name of the file to delete.

con_echo()

This function turns on or off the echoing of characters on a particular I/O stream. That is, it does not affect echoing globally, but only for the I/O stream on which it is issued. A command that uses this function takes one parameter: **ON | OFF**.

con_get()

This function displays a file from the file system. It works in the following fashion (if the command is called "GET"):

- ASCII mode: usage: "get <filename>"
The file is then sent, followed by the usual OK message.
- BINARY mode: usage: "get <filename> <size in bytes>"
The message "LENGTH <len>" will be sent, indicating length of the file to be sent, and then the file will be sent, but not more than <size in bytes> bytes.

con_getv()

This function displays the value of the given variable. The variable is displayed using the printf-style format specifier given in the `createv` command. A command that uses this function takes one parameter: the name of the variable.

con_help()

This function implements the help system for Zconsole. A command that uses this function takes one parameter: the name of another command. Zconsole outputs the associated help text for the requested command. The help text is the text file referenced in the third field of the `Console-Command` structure.

con_list_files()

This function lists the files in the file system and their file sizes. A command that uses this function takes no parameters.

con_list_variables()

This function displays the names and types of all variables. A command that uses this function takes no parameters.

con_loginname()

This function stores an identifier that will be remembered across power cycles (with battery-backed RAM). The existence of the identifier will be used to prompt the user of a new console session. Before console access to the controller is allowed, a valid identifier must be entered in response to the prompt. A command that uses this function takes one parameter: an identifier that will be used as the login name.

con_loginpassword()

This function stores an identifier that will be remembered across power cycles (with battery-backed RAM). The existence of the identifier will be used to prompt the user for a password after a login name has been entered. Before console access to the controller is allowed, a valid identifier must be entered in response to the prompt. A command that uses this function takes no parameters on the command line, but requires a series of user inputs in response to prompts. In the following screen shot, the command is named "login password," and is typed in by the user. All other screen text shown here was printed by Zconsole.



If no identifier is stored for the password, a <CR> must be sent in response to the prompt for the old password.

NOTE: A login name must be stored by a command using `con_loginname()` for a login password to be applicable, i.e., if a password has been stored but no login name, new console sessions will not prompt for the password or a login name. If a login name is applicable, but there is no password, new console sessions will prompt for the login name and grant access after a valid name is entered without prompting for a password.

con_logout()

This function exits the current console session and begins a new session by entering the initialization state of Zconsole. A command that uses this function takes no parameters.

con_mail()

This function sends e-mail to the server specified by `con_mail_server()`, with the return address specified by `set_mail_from()`. A command that uses this function takes one parameter: the destination e-mail address. If the command is named mail, the usage is:

```
"mail destination@where.com"
```

The first line of the message will be used as the subject, and the other lines are the body. The body is terminated with a ^D or ^Z (0x04 or 0x1A).

con_put()

This function creates a new file in the file system for use with the HTTP server. It works in the following fashion (if the command is called "PUT"):

- ASCII mode: usage: "put <filename>"
The file is then sent, terminating with a ^D or ^Z (0x04 or 0x1A).
- BINARY mode: usage: "put <filename> <size in bytes>"
The file is then sent, and must be exactly the specified number of bytes in length.

Note that ASCII mode is only useful for text files, since the console will ignore non-displayable characters. In binary mode, the put command will time out after CON_TIMEOUT seconds of inactivity (60 by default).

con_putv()

This function updates the value of a variable. A command that uses this function takes two parameters: the name of the variable, and the new value for the variable.

con_reset_files()

This function removes all web files.

con_reset_variables()

This function removes all web variables.

con_set_dhcp()

This function turns DHCP configuration for an interface "on" or "off." Currently this command only works with the default interface. After DHCP has been turned on, ZConsole will undertake reacquiring the lease should it be dropped. (For example, a lease might be dropped if the DHCP server is unavailable for an extended period of time.)

con_set_gateway()

This function changes the gateway of the board. A command that uses this function takes one parameter: the new gateway in dotted quad notation, e.g., 192.168.1.1.

con_set_icmp_config()

This function configures an interface to use directed ICMP echo request (ping) packets for configuration. A command that uses this function takes two parameters. The first is "on" or "off" to turn this feature on or off. The second parameter is optional, and specifies the intended interface (ETH0 or ETH1). Only non-PPPoE Ethernet may be used for ping configuration.

con_set_icmp_config_reset()

Normally, when an interface has been configured via a directed ping packet, further configuration via a directed ping packet is disabled (until the next power cycle). This function allows the interface to be configured via a ping packet again. A command that uses this function takes an optional interface argument (ETH0 or ETH1).

con_set_ip()

This function changes the IP address of the board. A command that uses this function takes one parameter: the new IP address in dotted quad notation, e.g., 192.168.1.112.

con_set_param()

This function sets the parameter for the current I/O device. Depending on the I/O device, this value could be a baud rate, a port number or a channel number. A command that uses this function takes one parameter: the value for the I/O device parameter.

con_set_mail_from()

This function sets the return address for all e-mail messages. This address will be added to the outgoing e-mail and should be valid in case the e-mail needs to be returned. A command that uses this function takes one parameter: the return address.

con_set_mail_server()

This functions identifies the SMTP server to use. A command that uses this function takes one parameter: the IP address of the SMTP server.

con_set_nameserver()

This function changes the name server for the board. A command that uses this function takes one parameter: the IP address of the new name server in dotted quad notation, e.g., 192.168.1.1.

con_set_netmask()

This function changes the netmask of the board. A command that uses this function takes one parameter: the new netmask in dotted quad notation, e.g., 255.255.255.0.

con_set_tcpip_debug()

This function is intended to aid in development and debugging. A command that uses this function takes one parameter: the numerical level of debugging messages. The higher the number, the more verbose the TCP/IP debugging messages will be.

con_show()

This function displays the current configuration of the board (IP address, netmask, and gateway). If the developer's application has configuration options she would like to show other than the IP address, netmask, and gateway, she will probably want to implement her own version of the show command. The new show command can be modelled after `con_show()` in `ZConsole.lib`. A command that uses this function takes no parameters.

con_show_multi()

Like the `con_show()` function, this function shows the current console configuration. This command will, however, show more network configuration than is available via `con_show()`.

Interface-specific configuration information is separated out. A command that uses this function takes an optional parameter (ETH0, ETH1, PPP0, PPP1, PPP2, etc.) to display the interface specific configuration for the specified interface. If the optional parameter is missing, the current console configuration for all valid interfaces is displayed.

11.4.1.2 Custom Zconsole Commands

Developers are not limited to the default commands. A custom command is easy to add to Zconsole; simply create an entry for it in `console_commands []`. The three fields of this entry were described in Section 11.3.1. The first field is the name of the command. The second field is the function that implements the command. This function must follow this prototype:

```
int function_name ( ConsoleState *state );
```

The parameter passed to the function is a structure of type `ConsoleState`. Some of the fields in this structure must be manipulated by your custom command function, other fields are used by `Zconsole.lib` and must not be changed by the your program.

```
typedef struct {
    int console_number;
    ConsoleIO *conio;
    int state;
    int laststate;

    char command[CON_CMD_SIZE];
    char *cmdptr;
    char buffer[CON_BUF_SIZE]; // Use for reading in data.
    char *bufferend; // Use for reading in data.

    ConsoleCommand *cmdspec;
    int sawcr;
    int sawesc;
    int echo; // Check if echo is enabled, or change it.
    int substate;
    unsigned int error;
    int numparams; // Number of parameters on command line.
    int commandparams; // Number of commands issued on cmd line
    char cmddata[CON_CMD_DATA_SIZE];

#ifdef CON_NO_FS_SUPPORT // File processing not needed with DC 7.30
    FileNumber filenum; // Use for file processing.
    File file; // Use for file processing.
#endif
    int spec; // Use for working with Zserver entities
    long timeout; // Use for extending the time out.
} ConsoleState;
#endif
```

To accomplish its tasks, the function should use `state->substate` for its state machine (which is initialized to zero before dispatching the command handler), and `state->command` to read out the command buffer (to get other parameters to the command, for instance). In case of error, the function should set `state->error` to the appropriate value.

The buffer at `state->cmddata` is available for the command to preserve data across invocations of the command's state machine. The size of the buffer is adjustable via the `CON_CMD_DATA_SIZE` macro (set to 16 bytes by default). Generally this buffer area will be cast into a data structure appropriate for the given command state machine.

Both `state->numparams` and `state->commandparams` are read-only. The latter was introduced in Dynamic C 7.30. It indicates the number of arguments in the command line that are NOT part of the command name itself. For instance, for the command

```
SET IP 10.10.6.112 ETH0
```

`state->commandparams` would be 2, but `state->numparams` would be 4. This distinction is made to allow the commands in Zconsole to be insensitive to the number of words that make up the name of the command itself, but still maintain backwards compatibility with custom commands that use `state->numparams`.

The function that implements the custom command should return 0 when it has more processing to do (and thus will be called again), 1 for successful completion of its task, and -1 to report an error.

The third and final field of the `console_commands []` entry is the physical address of the help text file for the custom command in question. This file must be `#ximported`, along with all of the default command function help files that are being used.

IMPORTANT: The fields discussed in the previous paragraph and the fields that have comments in the structure definition are the only ones that an application program should change. The other fields must not be changed.

11.4.2 Zconsole Error Messages

`ZCONSOLE.LIB` provides a list of default error messages for the default Zconsole commands. An application program must define an array for these error messages, as well as for any custom error messages that are desired. To include only the default error messages, the following array is sufficient:

```
const ConsoleError console_errors[] = {
    CON_STANDARD_ERRORS           // includes all default error messages
}
```

11.4.2.1 Default Error Messages

These are the error codes for the default error messages and the text that will be displayed by the console if the error occurs.

```
#define CON_ERR_TIMEOUT 1
#define CON_ERR_BADCOMMAND 2
#define CON_ERR_BADPARAMETER 3
#define CON_ERR_NAMETOOLONG 4
#define CON_ERR_DUPLICATE 5
#define CON_ERR_BADFILESIZE 6
#define CON_ERR_SAVINGFILE 7
#define CON_ERR_READINGFILE 8
#define CON_ERR_FILENOTFOUND 9
#define CON_ERR_MSGTOOLONG 10
#define CON_ERR_SMTPELOR 11
#define CON_ERR_BADPASSPHRASE 12
#define CON_ERR_CANCELRESET 13
#define CON_ERR_BADVARTYPE 14
#define CON_ERR_BADVARVALUE 15
#define CON_ERR_NOVARSPACE 16
#define CON_ERR_VARNOTFOUND 17
#define CON_ERR_STRINGTOOLONG 18
#define CON_ERR_NOTAFILE 19
#define CON_ERR_NOTAVAR 20
#define CON_ERR_COMMANDTOOLONG 21
#define CON_ERR_BADIPADDRESS 22
#define CON_ERR_INVALIDPASSWORD 23
#define CON_ERR_BADIFACE 24
#define CON_ERR_BADNETWORKPARAM 25
```



```

#define CON_STANDARD_ERRORS \
{CON_ERR_TIMEOUT,          "Timed out." },\
{CON_ERR_BADCOMMAND,      "Unknown command." },\
{CON_ERR_BADPARAMETER,    "Bad or missing parameter." },\
{CON_ERR_NAMETOOLONG,     "Filename too long." },\
{CON_ERR_DUPLICATE,       "Duplicate object found." },\
{CON_ERR_BADFILESIZE,     "Bad file size." },\
{CON_ERR_SAVINGFILE,      "Error saving file." },\
{CON_ERR_READINGFILE,     "Error reading file." },\
{CON_ERR_FILENOTFOUND,    "File not found." },\
{CON_ERR_MSGTOOLONG,      "Mail message too long." },\
{CON_ERR_SMTPELOR,        "SMTP server error." },\
{CON_ERR_BADPASSPHRASE,   "Passphrases do not match!" },\
{CON_ERR_CANCELRESET,     "Reset cancelled." },\
{CON_ERR_BADVARTYPE,      "Bad variable type." },\
{CON_ERR_BADVARVALUE,     "Bad variable value." },\
{CON_ERR_NOVARSPACE,      "Out of variable space." },\
{CON_ERR_VARNOTFOUND,     "Variable not found." },\
{CON_ERR_STRINGTOOLONG,   "String too long." },\
{CON_ERR_NOTAFILE,        "Not a file." },\
{CON_ERR_NOTAVAR,         "Not a variable." },\
{CON_ERR_COMMANDTOOLONG,  "Command too long." },\
{CON_ERR_BADIPADDRESS,    "Bad IP address." },\
{CON_ERR_INVALIDPASSWORD, "Invalid Password." },\
{CON_ERR_BADIFACE,        "Bad interface name." },\
{CON_ERR_BADNETWORKPARAM, "Error setting network parameter." }

```

11.4.2.2 Custom Error Messages

Developers can create their own error messages by following the format of the default error messages. The error code numbers should be greater than 1,000 to save room for expansion of built-in error messages.

```

#define NEW_ERROR 1001
const ConsoleError console_errors[] = {
    CON_STANDARD_ERRORS, // includes all default error messages
    { NEW_ERROR, "Any error message I want." }
}

```

The default error messages should be included in `console_errors[]` along with any custom error messages that are used since the commands that come with `Zconsole.lib` each expect their own particular error message.

11.5 Zconsole I/O Interface

Multiple I/O methods are supported, as well as the ability to add custom I/O methods. An array of ConsoleIO structures must be defined in the application program and named `console_io[]`. This structure holds handlers for common I/O functions for the I/O method.

```
typedef struct {
    long param;           // Baud for serial, port for telnet, etc.
    int (*open) ();
    void (*close) ();
    int (*tick) ();
    int (*puts) ();
    int (*rdUsed) ();
    int (*wrUsed) ();
    int (*wrFree) ();
    int (*read) ();
    int (*write) ();
} ConsoleIO;
```

11.5.1 How to Include an I/O Method

Each supported I/O method is determined at compile time, i.e., each supported I/O method must have an entry in `console_io[]`.

11.5.2 Predefined I/O Methods

Several predefined I/O methods are in `Zconsole.lib`. They will be included by entering their respective macros in `console_io[]`.

```
const ConsoleIO console_io[] = {
    CONSOLE_IO_SERA(baud rate),
    CONSOLE_IO_SERB(baud rate),
    CONSOLE_IO_SERC(baud rate),
    CONSOLE_IO_SERD(baud rate),
    CONSOLE_IO_SP(channel number),
    CONSOLE_IO_TELNET(port number),
}
```

The macros expand to the appropriate set of pre-defined handler functions, e.g.,

```
#define CONSOLE_IO_SERA(param) { param, serAopen, serAclose,
    NULL, conio_serAputs, serArdUsed, serAwrdUsed, serAwrdFree,
    serAread, serAwrite}
```

11.5.2.1 Serial Ports

There are predefined I/O methods for all four of the serial ports on a Rabbit board. The baud rate is set by passing it to the macro. See above.

11.5.2.2 Telnet

Zconsole runs a telnet server. The port number is passed to the macro `CONSOLE_IO_TELNET`. The user telnets to the controller that is running the console.

11.5.2.3 Slave Port

The Rabbit slave port is an 8-bit bidirectional data port. Zconsole runs on the slave processor. Two drivers are needed.

11.5.2.3.1 Slave Port Driver

The slave port driver is implemented by `SLAVE_PORT.LIB`. For an application to use the slave port:

- The driver must be installed by including the library in the program.
- A call to `SPinit(mode)` must be made to initialize the driver.
- A function to process Zconsole commands sent to the slave port must be provided.

The slave port has 256 channels, separate port addresses that are independent of one another. A handler function for each channel that is used must be provided. For details on how to do this, please see the *Dynamic C User's Manual*.

A stream-based handler, `SPShandler()`, to process Zconsole commands for the slave is provided in `SP_STREAM.LIB`. The handler is set up automatically by the console when the slave port I/O method is included. The macro, `CONSOLE_IO_SP`, expands to the I/O functions defined in `SP_STREAM.LIB`.

11.5.2.3.2 Master Connected to Rabbit Slave Port

The master controller board can be another Rabbit processor or something else.

The master also needs a driver for its end of the slave port connection. An example of the software needed on the master is given in `MASTER_SERIAL.LIB`. The software on the master controller is, of course, specific to the task at hand. In the example driver provided, most of the work is done by the slave, making minimal changes necessary to the code on the master.

11.5.2.4 Custom I/O Methods

To define a custom I/O method, you must add a structure of type `ConsoleIO` to `console_io[]`. This structure holds the common handler functions for the I/O method. The tick function may have a `NULL` pointer, but the rest of the function pointers must be valid pointers to functions.

11.5.3 Multiple I/O Streams

Each I/O method has its own state machine in Zconsole. That means that each I/O method is independent of the others and they can all be used simultaneously. This imposes the important restriction that all command handlers be able to run simultaneously on different I/O streams or support proper locking for functions that cannot be performed simultaneously.

11.6 Zconsole Execution

Normally, Zconsole will communicate over a serial link. The physical connection will differ slightly from board to board. Basically, you will need a 3 wire (GND, RXD, TXD) serial cable. Several initialization steps must be taken at the beginning of an application program to execute the console.

11.6.1 File System Initialization

Prior to Dynamic C 7.30, Zconsole depended on the flash file system included with Dynamic C. There are actually two file systems: FS1 was the first Dynamic C file system. The second one, FS2 (introduced with Dynamic C 7.05), is an improved file system.

Besides defining the macro that directs the file system to EEPROM memory and including the appropriate library, i.e.,

```
#define FS_FLASH
#use "filesystem.lib" // If using the improved file system available with
                    // Dynamic C 7.05, substitute "fs2.lib" for "filesystem.lib"
```

the application program must initialize the file system with a call to `fs_init()`. Starting with Dynamic C 7.30 none of this is necessary because Zconsole saves configuration information to the User block. See the *Rabbit 2000 Microprocessor User's Manual* or the *Rabbit 3000 Microprocessor User's Manual* for more information about the User block.

11.6.2 Serial Buffers

If the pre-defined serial I/O methods are used, the circular buffers used for I/O data can be resized from their default values of 31 bytes by using macros. For example, if `CONSOLE_IO_SERIALC` is included in `console_io[]`, then lines similar to the following can be in the application program:

```
#define CINBUFSIZE 1023
#define COUTBUFSIZE 255
```

In general, these buffers can be smaller for slower baud rates, but must be larger for faster baud rates.

11.6.3 Using TCP/IP

To use the TCP/IP functionality of Zconsole you must have the following line in your application program:

```
#use "dcrtcp.lib"
```

If you are serving web pages you must also include `http.lib`, and if you are sending e-mail you must include `smtplib`.

11.6.4 Required Zconsole Functions

To run the console, the following two functions are required.

console_init

```
int console_init( void );
```

DESCRIPTION

This function will initialize Zconsole data structures. It must be called before `console_tick()` is called for the first time. This function also loads the configuration information from the file system.

RETURN VALUE

- 0: Success;
- 1: No configuration information found.
- <0: Indicates an error loading the configuration data;
 - 1 indicates an error reading the 1st set of information,
 - 2 the 2nd set, and so on.

LIBRARY

`zconsole.lib`

console_tick

```
void console_tick( void );
```

DESCRIPTION

This function needs to be called periodically in an application program to allow Zconsole time for processing.

LIBRARY

`zconsole.lib`

11.6.5 Useful Zconsole Function

Most of the following functions are only useful for creating custom commands.

`con_backup`

```
int con_backup( void );
```

DESCRIPTION

This function backs up the current configuration.

RETURN VALUE

0: Success
1: Failure

LIBRARY

`zconsole.lib`

SEE ALSO

`con_backup_reserve`, `con_load_backup`

`con_backup_bytes`

```
long con_backup_bytes( void );
```

DESCRIPTION

Returns the number of bytes necessary for each backup configuration file. Note that enough space for two of these files needs to be reserved. This function is most useful when `ZCONSOLE.LIB` is being used with `FS2.LIB`.

RETURN VALUE

Number of bytes needed for a backup configuration file.

LIBRARY

`zconsole.lib`

SEE ALSO

`con_backup_reserve`

con_backup_reserve

```
void con_backup_reserve( void );
```

DESCRIPTION

Reserves space for the configuration information in the file system. For more information on the file system see the *Dynamic C User's Manual*.

LIBRARY

zconsole.lib

SEE ALSO

con_backup, con_load_backup, con_backup_bytes

con_chk_timeout

```
int con_chk_timeout( unsigned long timeout );
```

DESCRIPTION

Checks whether the given timeout value has passed.

RETURN VALUE

0: Timeout has not passed
!0: Timeout has passed

LIBRARY

zconsole.lib

SEE ALSO

con_set_timeout

con_load_backup

```
int con_load_backup( void );
```

DESCRIPTION

Loads the configuration from the file system.

RETURN VALUE

0: Success
1: No configuration information found
<0: Failure
-1: error reading 1st set of information
-2: error reading 2nd set of information, and so on

LIBRARY

zconsole.lib

SEE ALSO

con_backup, con_backup_reserve

con_reset_io

```
void con_reset_io( void );
```

DESCRIPTION

Resets all I/O methods by calling `close()` and `open()` on each of them.

LIBRARY

zconsole.lib

con_set_backup_lx

```
void con_set_backup_lx( FSLXnum backuplx );
```

DESCRIPTION

Sets the logical extent (LX) that will be used to store the backup configuration data. For more information on the file system see the *Dynamic C User's Manual*. This is only useful in conjunction with `FS2.LIB`. This should be called once before `console_init()`. Care should be taken that enough space is available in this logical extent for the configuration files. See `con_backup_bytes()` for more information.

PARAMETER

backuplx LX number to use for backup

LIBRARY

`zconsole.lib`

SEE ALSO

`con_set_files_lx`, `con_backup_bytes`

con_set_files_lx

```
void con_set_files_lx( FSLXnum fileslx );
```

DESCRIPTION

Sets the logical extent (LX) that will be used to store files. For more information on the file system see the *Dynamic C User's Manual*. This is only useful in conjunction with `FS2.LIB`. This should be called once before `console_init()`.

PARAMETER

fileslx LX number to use for files.

LIBRARY

`zconsole.lib`

SEE ALSO

`con_set_backup_lx`

con_set_user_idle

```
void con_set_user_idle( void (*funcptr)() );
```

DESCRIPTION

Sets a user-defined function that will be called when the console (for a particular I/O channel) is idle. The user-defined function should take an argument of type `ConsoleState*` .

LIBRARY

`zconsole.lib`

SEE ALSO

`con_set_user_timeout`

con_set_timeout

```
unsigned long con_set_timeout( unsigned int seconds );
```

DESCRIPTION

Returns the value that `MS_TIMER` should have when the number of seconds given have elapsed.

LIBRARY

`zconsole.lib`

SEE ALSO

`con_chk_timeout`

con_set_user_timeout

```
void con_set_user_timeout( void (*funcptr)() );
```

DESCRIPTION

Sets a user-defined function that will be called when a timeout event has occurred. The user-defined function should take an argument of type `ConsoleState*`.

LIBRARY

`zconsole.lib`

SEE ALSO

`con_set_user_idle`

console_disable

```
void console_disable( int which );
```

DESCRIPTION

Disable processing for the designated console in the `console_io[]` array. This function, along with `console_enable()`, allows the sharing of the Zconsole port with some other processing.

PARAMETER

which The console to disable.

LIBRARY

`zconsole.lib`

SEE ALSO

`console_init`, `console_enable`

console_enable

```
void console_enable( int which );
```

DESCRIPTION

Enable processing for the designated console in the `console_io[]` array. This function, along with `console_disable()`, allows the sharing of the Zconsole port with some other processing.

PARAMETER

which The console to enable.

LIBRARY

`zconsole.lib`

SEE ALSO

`console_init`, `console_disable`

11.6.6 Zconsole Execution Choices

Zconsole can be used interactively with a terminal emulator or by sending commands from a program running on a device connected to the controller that is running the console.

11.6.6.1 Terminal Emulator

To manually enter Zconsole commands from a keyboard and view results in the Stdio window you must:

1. Run Dynamic C 7.05 or later.
2. Open a terminal emulator. Windows HyperTerminal comes with Windows. It does not work with binary files, only ASCII. Tera Term can handle both ASCII and binary. It is available for free download at

`http://hp.vector.co.jp/authors/VA002416/teraterm.html`

3. Configure the terminal emulator as follows:

COM port: (1 or 2) to which 3-wire serial cable is connected

Baud Rate: 57,600 bps

Data Bits: 8

Parity: None

Stop Bits: 1

Flow Control: None

The terminal emulator should now accept Zconsole commands.

To avoid losing a <LF> at the beginning of a file when using the `con_put` command function, select **Setup->Terminal** from the Tera Term menu and set the Transmit option to **CR+LF**. This option might be located elsewhere if you are using a different terminal emulator.

11.7 Backup System

Zconsole can save configuration parameters to the file system or, starting with Dynamic C 7.30, to the User block. The configuration is then available across power cycles. The backup process is done by `con_backup()`. Unlike the other Zconsole command functions, `con_backup()` does not take a parameter and it returns 0 if the backup was successful and 1 if it was not. This function is called by several of the Zconsole command functions that change configuration parameters, or that add or delete files or variables from the file system. Caution is advised when calling `con_backup()` since it writes to flash memory.

11.7.1 Data Structure for Backup System

The developer must define an array called `console_backup[]` of `ConsoleBackup` structures.

```
typedef struct {
    void *data;
    int len;
    void (*postload)();
    void (*presave)();
} ConsoleBackup;
```

data

This is a pointer to the data to be backed up.

len

This is how many bytes of data need to be backed up.

postload

This is a function pointer to a function that is called after configuration data is loaded, in case the developer needs to do something with the newly loaded configuration data.

presave

This is a function pointer that is called just before the configuration data is saved so that the developer can fill in the data structure to be saved. The functions referenced by `postload()` and `presave()` should have the following prototype:

```
void my_preload(void *dataptr);
```

The `dataptr` parameter is the address of the configuration data (the same as the data pointer in the `ConsoleBackup` structure).

11.7.2 Array Definition for Backup System

```
const ConsoleBackup console_backup[] = {
    CONSOLE_BASIC_BACKUP,          // echo state, baud rate/port number
    CONSOLE_TCPIP_BACKUP,
    CONSOLE_TCP_MULTI_BACKUP,
    CONSOLE_HTTP_BACKUP,
    CONSOLE_SMTTP_BACKUP,
    CONSOLE_BACKUP_LOGIN,
    { my_data, my_data_len, my_preload, my_presave }
}
```

CONSOLE_BASIC_BACKUP

Causes backup of the echo state (on or off), baud rate and port number information.

CONSOLE_TCPIP_BACKUP

Causes backup of the IP addresses of the controller board and the IP address of its net-mask, gateway and name server.

Note that only one of the `CONSOLE_TCP_*` structures should be used.

CONSOLE_TCP_MULTI_BACKUP

Using this structure causes `ifconfig()` to save and restore network configuration. In addition to the information saved by `CONSOLE_TCP_BACKUP`, multiple name servers, DHCP configuration, ICMP (Ping) configuration, and multiple interface configuration are all saved by `CONSOLE_TCP_MULTI_BACKUP`.

Some built-in console functions are for use with `CONSOLE_TCP_MULTI_BACKUP`. In general, except for backwards compatibility issues, `CONSOLE_TCP_MULTI_BACKUP` should be used instead of `CONSOLE_TCP_BACKUP`.

Note that only one of the `CONSOLE_TCP_*` structures should be used.

CONSOLE_HTTP_BACKUP

Causes backup of the files and variables visible to the HTTP server.

CONSOLE_SMTTP_BACKUP

Causes backup of the mail configuration.

CONSOLE_BACKUP_LOGIN

Causes backup of the `ConsoleLogin` structure which stores the login name and password strings.

11.8 Zconsole Macros

Many macros are available to change the behavior of Zconsole. They are all listed here. Starting with Dynamic C 7.30 additional macros are available to support saving configuration information to the User block, DHCP, ping configuration, and multiple interfaces.

CON_BACKUP_FILE1

The file number used for the first backup file. For FS1, this number must be in the range 128-143, so that `fs_reserve_blocks()` can be used to guarantee free space for the backup files. Defaults to 128 for FS1. Defaults to 254 for FS2.

CON_BACKUP_FILE2

Same as above, except this is for the second backup file. Two files are used so that configuration information is preserved even if the power cycles while configuration data is being saved. For FS1, this number must be in the range 128-143. Defaults to 129 for FS1. Defaults to 255 for FS2.

CON_BACKUP_USER_BLOCK

Defaults to not defined. If this is defined, then configuration information for the console will be saved to the User block instead of to the flash file system. Note that the configuration is only safe in the case of power failures with a version 3 or higher System ID block.

CON_BUF_SIZE

Changes the size of the data buffer that is allocated for each I/O method. If the baud rate or transfer speed is too great for the console to keep up, then increasing this value may help avoid dropped characters. It is allocated in root data space. It defaults to 1024 bytes.

CON_CMD_SIZE

Changes the size of the command buffer that is allocated for each I/O method. This limits the length of a command line. It is allocated in root data space. Defaults to 128 bytes.

CON_CMD_DATA_SIZE

Default is 16. Adjusts the size of the user data area within the state structure so that user commands may preserve arbitrary information across calls. The user data area is allocated in root data space.

CON_DHCP_ACQUIRE_RETRY_TIMEOUT

Defaults to 120 seconds. If DHCP is enabled, then Zconsole will maintain the DHCP lease. This macro specifies the number of seconds after which a DHCP lease has been dropped that the board will attempt to reacquire the lease. Note that in the normal course of operation, a lease will never be dropped. Generally, that will only happen if the DHCP server is inoperable for an extended period of time (subject to the lengths of the leases that the DHCP server issues).

CON_HELP_VERSION

This macro should be defined if the developer wants a version message to be displayed when the HELP command is issued with no parameters. If this macro is defined, then the macro CON_VERSION_MESSAGE must also be defined.

CON_INIT_MESSAGE

Defines the message that is displayed on all Zconsole I/O methods upon startup. Defaults to “Console Ready\r\n”.

CON_MAIL_BUF_SIZE

Maximum length of a mail message. Defaults to 1024.

CON_MAIL_FROM_SIZE

Maximum length of mail from address to NULL terminator. Default to 51.

CON_MAIL_SERV_SIZE

Maximum length of mail server name and NULL terminator. Defaults to 51.

CON_MAX_NAME

Default is 10: maximum number of characters for a login name. This value must be equal to or less than CON_CMD_DATA_SIZE.

CON_MAX_PASSWORD

Default is 10: maximum number of characters for a login password.

CON_NO_FS_SUPPORT

This macro is defined by default only if no filesystem libraries have been used. Even if a filesystem library has been used, this can still be explicitly defined by the user. When this is defined, then the console will not save configuration information to the filesystem, and no filesystem function calls will be included.

CON_SP_RDBUF_SIZE

Size of the slave port read buffer. Defaults to 255.

CON_SP_WRBUF_SIZE

Size of the slave port write buffer. Defaults to 255.

CON_TIMEOUT

Adjusts the number of seconds that the console will wait before cancelling the current command. The timeout can be adjusted in user code in the following manner:

```
state->timeout = con_set_timeout(CON_TIMEOUT);
```

This is useful for custom user commands so that they can indicate when something “meaningful” has happened on the console (such as some data being successfully transferred).

CON_VAR_BUF_SIZE

Adjusts the size of the variable buffer, in which values of variables can be stored for use with the HTTP server. It is allocated in xmem space. Defaults to 1024 bytes.

CON_VERSION_MESSAGE

This defines the version message to display when the HELP command is issued with no parameters. It is not defined by default, so has no default value.

11.9 Sample Program

The sample program `Samples\zconsole\tcpipconsole.c` demonstrates many of the features of `zconsole.lib`. Among the features this application supports is network configuration, uploading web pages, changing variables for use with web pages, sending mail, and access to the console through a telnet client. Please note that all libraries needed by `zconsole.lib` must be included with `#use` statements before the `#use` statement for the `Zconsole` library.

The following code is taken from `tcpipconsole.c`.

```
/*
 * Size of the buffers for serial port C. If you want to use another serial port, you should
 * change the buffer macros below appropriately (and change the console_io[] array below).
 */
#define CINBUFSIZE 1023
#define COUTBUFSIZE 255

/*
 * Maximum number of connections to the web server. This indicates the number of sockets
 * that the web server will use.
 */
#define HTTP_MAXSERVERS 2

/*
 * Maximum number of sockets this program can use. The web server is taking two sockets:
 * the mail client uses one socket, and the telnet interface uses the other socket.
 */
#define MAX_SOCKETS 4

/*
 * All web server content is dynamic, so we do not need http_flashspec[].
 */
#define HTTP_NO_FLASHSPEC

/*
 * The file system that the console uses should be located in flash.
 */
#define FS_FLASH

/*
 * The function prototype for a custom command must be declared before the
 * console_command[] array.
 */
int hello_world ( ConsoleState *state);
```

The following code is for Zconsole configuration.

```
/*
 * The number of console I/O streams that this program supports. Since we are supporting
 * serial port C and telnet, there are two I/O streams.
 */
#define NUM_CONSOLES 2

/*
 * If this macro is defined, then the version message will be shown with the help command,
 * when the help command has no parameters.
 */
#define CON_HELP_VERSION

/*
 * Defines the version message that will be displayed in the help command if
 * CON_HELP_VERSION is defined.
 */
#define CON_VERSION_MESSAGE "TCP/IP Console Version 1.0\r\n"

/*
 * Defines the message that is displayed on all I/O channels when the console starts.
 */
#define CON_INIT_MESSAGE CON_VERSION_MESSAGE

/*
 * The ximport directives include the help texts for the console commands. Having the help text
 * in xmem helps save root code space.
 */
#import "samples\zconsole\tcpipconsole_help\help.txt" help_txt
...
/* The rest of the #import statements may be seen in tcpipconsole.c. */
```

The following code sets up all the data structures needed by the console.

```
/* The console will be available to the I/O streams given in the following array. The I/O streams
 * are defined through macros as documented in Section 11.5.2. The parameter for the first macro
 * represents the initial baud rate for serial port C. The second macro is passed the port number
 * for telnet. If you change the number of I/O streams, update NUM_CONSOLES above.*/

const ConsoleIO console_io[] = {
    CONSOLE_IO_SERC(57600),
    CONSOLE_IO_TELNET(23)
};

/* This array defines the commands that are available in the console. The first parameter for the
 * ConsoleCommand structure is the command specification, i.e., how the console
 * recognizes a command. The second parameter is the function to call when the command
 * is recognized. The third parameter is the location of the #ximport'ed help file for the command.
 * Note that the second parameter can be NULL, which is useful if help information is needed
 * for something that is not a command (like for the "SET" command below--the help file for
 * "SET" contains a list of all of the set commands). Also note the entry for the command "",
 * which is used to set up the help text that is displayed when the help command is used by
 * itself (that is, with no parameters).*/

const ConsoleCommand console_commands[] = {
    { "HELLO WORLD", hello_world, 0 },
    { "ECHO", con_echo, help_echo_txt },
    { "HELP", con_help, help_help_txt },
    { "", NULL, help_txt },
    { "SET", NULL, help_set_txt },
    { "SET PARAM", con_set_param, help_set_param_txt },
    ...
};

/* This array sets up the error messages that can be generated. CON_STANDARD_ERRORS is
 * a macro that expands to the standard errors used by the built-in commands in zconsole.lib.
 * Users can define their own errors here, as well.*/

const ConsoleError console_errors[] = {
    CON_STANDARD_ERRORS
};

/* This array defines the information (such as configuration) that will be saved to the file system.
 * Note that if, for example, the HTTP or SMTP related commands are included in the
 * console_commands array above, then the backup information must be included in
 * this array. The entries below are macros that expand to the appropriate entry for each set of
 * functionality. Users can also add their own information to be backed up here by adding
 * more ConsoleBackup structures.*/

const ConsoleBackup console_backup[] = {
    CONSOLE_BASIC_BACKUP,
    CONSOLE_TCP_BACKUP,
    CONSOLE_HTTP_BACKUP,
    CONSOLE_SMTP_BACKUP
};
```

The following code defines the MIME types that the web server will handle.

```
const HttpType http_types[] = {
    { ".shtml", "text/html", shtml_handler}, // ssi
    { ".html", "text/html", NULL},          // html
    { ".gif", "image/gif", NULL},
    { ".jpg", "image/jpeg", NULL},
    { ".jpeg", "image/jpeg", NULL},
    { ".txt", "text/plain", NULL}
};
```

The function for the custom command is defined here and the main program finishes up the program. To see the complete sample, look in `Samples\zconsole\tcpipconsole.c`.

```
/* This is a custom command. Custom commands always take a ConsoleState* as an
 * argument (a pointer to the state structure for the given I/O stream), and return an int.
 * The return value should be 0 when the command wishes to be called again on the next
 * console_tick(), 1 when the command has successfully finished processing, or -1
 * when the command has finished due to an error.*/

int hello_world(ConsoleState *state) {
    state->conio->puts("Hello, World!\r\n");
    return 1;
}

void main(void) {
    /* Initialize TCP/IP, clients, servers, and I/O prior to using any console functions.*/
    sock_init();

    tcp_reserveport(80); // Start a listen queue and disable the 2MSL wait.
    http_init();

    if (fs_init(0, 64))
        printf("Filesystem not present!\n");

    if (console_init() != 0) {
        printf("Console did not initialize!\n");
        fs_format(0, 64, 1);

        /* After the file system has been initialized or formatted, space must be reserved in
         * the file system for the backup information. */

        con_backup_reserve();
        con_backup(); // Save the backup information to the console.
    }

    while (1) {
        console_tick();
        http_handler();
    }
}
```

Notice to Users

Z-WORLD PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE-SUPPORT DEVICES OR SYSTEMS UNLESS A SPECIFIC WRITTEN AGREEMENT REGARDING SUCH INTENDED USE IS ENTERED INTO BETWEEN THE CUSTOMER AND Z-WORLD PRIOR TO USE. Life-support devices or systems are devices or systems intended for surgical implantation into the body or to sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling and user's manual, can be reasonably expected to result in significant injury.

No complex software or hardware system is perfect. Bugs are always present in a system of any size. In order to prevent danger to life or property, it is the responsibility of the system designer to incorporate redundant protective mechanisms appropriate to the risk involved.

The Dynamic C TCP/IP software is designed for use only with Rabbit Semiconductor chips.

Index

Symbols

#echo var 168
#exec cmd 168
#include file 168

A

anonymous login 277
application protocols
 FTP client 267
 FTP server 277
 HTTP 153
 POP3 client 325
 SMTP client 311
 telnet 333
 TFTP client 303
authentication
 HTTP 161
 SMTP 312

B

basic authentication 161
BOOTP/DHCP
 used with TFTP 303

C

callbacks
 FTP data transfers 272
 sending HTTP headers 159
CGI 154, 169
console, serial-based 343–376

D

daemons
 FTP client 269
 FTP server 297
 HTTP server 234
 POP3 client 329
 telnet 340
 tftp_tick 307
 Zconsole 359
DHCP/BOOTP, See BOOTP/DHCP
directory listing 79
dynamic web pages 165

E

e-mail
 POP3 client 325–331
 SMTP client 311–323
entries in directory 79

F

file extensions 32, 164
file handlers 280
file size 270
file transfer 268
firewall 299
flow control 367
FTP client 267–275
FTP server 277–301
FTP server commands 299–300

Function Reference

Authentication and Identification

sauth_adduser 46
sauth_authenticate 47
sauth_getpassword 48
sauth_getserver 48
sauth_getuserid 49
sauth_getusermask 50
sauth_getusername 51
sauth_getwriteaccess 52
sauth_removeuser 53
sauth_setpassword 54
sauth_setserver 55
sauth_setusermask 56
sauth_setwriteaccess 57

CGI

cgi_continue 202
cgi_redirectto 203
cgi_sendstring 204
http_abortCGI 205
http_defaultCGI 209
http_finishCGI 214
http_getAction 215
http_getCond 217
http_getContentDisposition 218
http_getContentLength 219
http_getContentType 220
http_getContext 222
http_getData 223
http_getDataLength 224
http_getField 225
http_getHTTPMethod 226
http_getHTTPVersion 227
http_getRemainingLength 228
http_getSocket 229
http_getState 230
http_getTransferEncoding 231
http_getURL 232
http_getUserState 233
http_setCond 243
http_setState 246

http_skipCGI	248	smtp_sendmailxmem	320
http_sock_bytesready	249	smtp_status	323
http_sock_fastread	250	File System Specifics	
http_sock_gets	252	sspec_automount	72
http_sock_mode	253	sspec_fatregister	82
http_sock_tbleft	254	sspec_fatregistered	83
http_sock_write	255	FTP Client	
http_sock_xfastread	256	ftp_client_filesize	270
http_sock_xfastwrite	257	ftp_client_setup	268
http_switchCGI	259	ftp_client_tick	269
http_write	261	ftp_client_xfer	271
Console		ftp_data_handler	272
con_backup	360	ftp_last_code	274
con_backup_bytes	360	FTP Server	
con_backup_reserve	361	ftp_init	292
con_chk_timeout	361	ftp_set_anonymous	295
con_load_backup	362	ftp_shutdown	296
con_reset_io	362	ftp_tick	297
con_set_backup_lx	363	HTML Forms	
con_set_files_lx	363	http_finderrbuf	212
con_set_timeout	364	http_nextfverr	237
con_set_user_idle	364	http_parseform	238
con_set_user_timeout	365	http_scanpost	240
console_init	359	sspec_addfv	63
console_tick	359	sspec_findfv	84
Cookie		sspec_getformtitle	90
http_setcookie	244	sspec_getfvdesc	92
Data Conversion		sspec_getfventrytype	93
http_contentencode	207	sspec_getfvlen	94
http_date_str	208	sspec_getfvname	95
http_urldecode	260	sspec_getfvnum	96
Directory Navigation		sspec_getfvopt	97
sspec_cd	74	sspec_getfvoptlistlen	98
sspec_dirlist	79	sspec_getfvreadonly	99
sspec_pwd	119	sspec_getfvspec	100
Dynamic (RAM) Resource Table		sspec_getpreformfunction	105
sspec_addform	60	sspec_setformepilog	131
sspec_addfsfile	61	sspec_setformfunction	132
sspec_addfunction	62	sspec_setformprolog	133
sspec_addrootfile	64	sspec_setformtitle	134
sspec_addvariable	68	sspec_setfvcheck	135
sspec_addxmemfile	69	sspec_setfvdesc	136
sspec_addxmemvar	70	sspec_setfventrytype	137
sspec_aliasspec	71	sspec_setfvfloatrange	138
sspec_resizerootfile	126	sspec_setfvlen	139
Dynamic Rule Table		sspec_setfvname	140
sspec_addrule	65	sspec_setfvoptlist	141
sspec_removevrule	124	sspec_setfvrange	142
E-mail		sspec_setfvreadonly	143
pop3_getmail	328	sspec_setpreformfunction	145
pop3_init	327	HTTP Server	
pop3_tick	329	http_findname	213
smtp_mailltick	318	http_getcontext	221
smtp_sendmail	319	http_handler	234

http_idle	235	sspec_needsauthentication	115
http_init	236	sspec_readfile	121
http_safe	239	sspec_remove	123
http_set_anonymous	241	sspec_removeuser	125
http_set_path	245	sspec_restore	127
http_setauthentication	242	sspec_save	129
http_shutdown	247	sspec_setsavedata	147
http_status	258	sspec_setuser	148
MIME Types		Telnet	
sspec_getMIMEtype	102	telnet_close	340
Resource Access Control		telnet_init	339
sspec_access	58	telnet_tick	340
sspec_checkpermissions	76	vserial_close	334
sspec_getpermissions	104	vserial_init	334
sspec_getrealm	106	vserial_keepalive	335
sspec_setpermissions	144	vserial_listen	336
sspec_setrealm	146	vserial_open	337
Resource Location and Information		vserial_tick	338
sspec_findfsname	86	TFTP Client	
sspec_findname	85	tftp_exec	309
sspec_findnextfile	87	tftp_init	305
sspec_getfileloc	88	tftp_initx	306
sspec_getfiletype	89	tftp_tick	307
sspec_getfunction	91	tftp_tickx	308
sspec_getlength	101		
sspec_getname	103	H	
sspec_getservermask	107	HTML forms	45, 170–181
sspec_gettype	108	HTTP configuration macros	157
sspec_getvaraddr	111	HTTP server	153–266
sspec_getvarkind	112	HttpState	154
sspec_getvartype	113		
sspec_getxvaraddr	113	L	
Resource Retrieval and Update		listing directory entries	79
sspec_close	77		
sspec_delete	78	M	
sspec_mkdir	114	macros	
sspec_open	116	FTP server	278
sspec_readvariable	122	HTTP server	157
sspec_rmdir	128	serial ports for telnet	333
sspec_seek	130	SMTP client	314
sspec_stat	149	telnet (7.05 and later)	334
sspec_tell	151	telnet (pre 7.05)	339
sspec_write	152	Zconsole	370
Server Resource Management		MIME types	32, 164
http_addfile	206		
http_delfile	211	P	
shtml_addfunction	263	passive open	277
shtml_addvariable	264	password protection	46, 54
shtml_delfunction	265	permissions	
shtml_delvariable	266	defaults	40
sspec_adduser	67	POP_BUFFER_SIZE	325
sspec_checkaccess	75	POP_DEBUG	325
sspec_getuserid	109	POP_NODELETE	325
sspec_getusername	110		

POP_PARSE_EXTRA	325
POP3 client	
configuration	325
POST command	172
R	
resources	
access controls	16
rule table	16
S	
sample programs	
FTP server	298
POP3 client	330
SMTP client	313
telnet client	342
telnet server	341
Zconsole	373
SAUTH_MAXNAME	39
SAUTH_MAXUSERS	40
security	161
SERIAL_PORT_SPEED	339
server spec list	33, 36
SERVER_PASSWORD_ONLY	40
SMTP client	311–323
SMTP configuration macros	314
SSI	158, 168
SSL	161
SSPEC_MAX_FATDRIVES	40
SSPEC_MAX_OPEN	41
SSPEC_MAXNAME	40
SSPEC_MAXRULES	40
SSPEC_MAXSPEC	40
SSPEC_USERSPERRESOURCE	41
SSPEC_XMEMVARLEN	41
stack	
free space for TFTP functions	304
static resource table	43
static web pages	163

T	
telnet	333–342
TELNET_COOKED	339
TFTP client	303–309
time zone	162
TIMEZONE	159
TLS	161
U	
URL-encoded data	173
user table	16
users list	35
V	
VSERIAL_DEBUG	334
VSERIAL_NUM_GATEWAYS	334
W	
web browser control	3–29
well-known ports	
FTP server	277
HTTP server	158
POP3	325
SMTP server	311
Z	
Zconsole	343–376
backup system	368
circular buffers	358
commands	344
custom commands	352
error messages	353
I/O interfaces	356
macros	370–372
physical connection	358
terminal emulator	367
using TCP/IP	358