



## *Dynamic C*

### **TCP/IP User's Manual**

000423-A

---

# **Dynamic C TCP/IP User's Manual**

Part Number 019-0100 • 000423-A

Printed in U.S.A.

## **Copyright**

© 2001 Z-World, Inc. • All rights reserved.

The TCP/IP software used in the Rabbit 2000 TCP/IP Development Kit is designed for use only with Rabbit Semiconductor chips, and is used under licence from Erick Engelke.

Z-World, Inc. 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

## **Notice to Users**

When a system failure may cause serious consequences, protecting life and property against such consequences with a backup system or safety device is essential. The buyer agrees that protection against consequences resulting from system failure is the buyer's responsibility.

This device is not approved for life-support or medical systems.

All Z-World products are 100 percent functionally tested. Additional testing may include visual quality control inspections or mechanical defects analyzer inspections. Specifications are based on characterization of tested sample units rather than testing over temperature and voltage of each unit. Rabbit Semiconductor may qualify components to operate within a range of parameters that is different from the manufacturer's recommended range. This strategy is believed to be more economical and effective. Additional testing or burn-in of an individual unit is available by special arrangement.

## **Company Address**

### **Z-World, Inc**

2900 Spafford Street

Davis, California 95616-6800

USA

Telephone: (530) 757-3737

Facsimile: (530) 753-5141

Web site: <http://www.zworld.com>

# Table of Contents

- 1 Introduction .....1
- 2 TCP/IP Engine .....3
  - 2.1 TCP/IP Configuration.....3
    - IP Addresses Set Manually .....3
    - IP Addresses Set Dynamically .....3
      - BOOTP/DHCP Control
      - BOOTP/DHCP

|  |     |                                 |                                    |
|--|-----|---------------------------------|------------------------------------|
| tcp_open.....                            | 79  | sspec_getfvopt .....            | 110                                |
| tcp_reserveport .....                    | 81  | sspec_getfvoptlistlen .....     | 110                                |
| tcp_tick .....                           | 82  | sspec_getfvreadonly .....       | 111                                |
| udp_open .....                           | 83  | sspec_getfvspec .....           | 111                                |
| 2.8 Macros .....                         | 85  | sspec_getlength.....            | 112                                |
| DISABLE_DNS .....                        | 85  | sspec_getname .....             | 112                                |
| MAX_SOCKETS .....                        | 85  | sspec_getrealm.....             | 113                                |
| MY_DOMAIN .....                          | 85  | sspec_gettype.....              | 113                                |
| MY_GATEWAY.....                          | 85  | sspec_getusers. 113             |                                    |
| MY_IP_ADDRESS .....                      | 85  | ..1.6(p)9(ec_ch)9(ecec_ch)..... | 101                                |
| MY_NAMESERVER.....                       | 85  | ssp(s)6.4 [(s)6.3tname .....    | 0511.c_gett...6a8..... 112.8(..)-1 |
| MY_NETMASK .....                         | 85  |                                 |                                    |
| SOCK_BUF_SIZE .....                      | 86  |                                 |                                    |
| tcp_MaxBufSize .....                     | 86  |                                 |                                    |
| 3 Server Utility Library.....            | 87  |                                 |                                    |
| 3.1 Data Structures for Zserver.lib..... | 87  |                                 |                                    |
| ServerSpec Structure.....                | 87  |                                 |                                    |
| ServerAuth Structure.....                | 87  |                                 |                                    |
| FormVar Structure.....                   | 87  |                                 |                                    |
| 3.2 Constants Used in Zserver.lib .....  | 88  |                                 |                                    |
| ServerSpec Type Field .....              | 88  |                                 |                                    |
| ServerSpec Vartype Field.....            | 88  |                                 |                                    |
| Servermask field .....                   | 88  |                                 |                                    |
| Configurable Constants .....             | 88  |                                 |                                    |
| 3.3 HTML Forms .....                     | 89  |                                 |                                    |
| 3.4 Functions.....                       | 90  |                                 |                                    |
| sauth_adduser .....                      | 90  |                                 |                                    |
| sauth_authenticate .....                 | 91  |                                 |                                    |
| sauth_getusername.....                   | 92  |                                 |                                    |
| sauth_getwriteaccess .....               | 92  |                                 |                                    |
| sauth_setwriteaccess .....               | 93  |                                 |                                    |
| sspec_addform .....                      | 94  |                                 |                                    |
| sspec_addfsfile.....                     | 95  |                                 |                                    |
| sspec_addfunction .....                  | 96  |                                 |                                    |
| sspec_addfv .....                        | 96  |                                 |                                    |
| sspec_addrootfile .....                  | 97  |                                 |                                    |
| sspec_addvariable .....                  | 98  |                                 |                                    |
| sspec_addxmemfile .....                  | 99  |                                 |                                    |
| sspec_addxmemvar.....                    | 100 |                                 |                                    |
| sspec_aliasspec .....                    | 101 |                                 |                                    |
| sspec_checkaccess .....                  | 102 |                                 |                                    |
| sspec_findfv.....                        | 102 |                                 |                                    |
| sspec_findname .....                     | 103 |                                 |                                    |
| sspec_findnextfile .....                 | 104 |                                 |                                    |
| sspec_getfileloc .....                   | 104 |                                 |                                    |
| sspec_getfiletype .....                  | 105 |                                 |                                    |
| sspec_getformtitle.....                  | 105 |                                 |                                    |
| sspec_getfunction .....                  | 106 |                                 |                                    |
| sspec_getfvdesc .....                    | 107 |                                 |                                    |
| sspec_getfventrytype .....               | 108 |                                 |                                    |
| sspec_getfvlen .....                     | 108 |                                 |                                    |
| sspec_getfvname.....                     | 109 |                                 |                                    |
| sspec_getfvnum .....                     | 109 |                                 |                                    |
|  |     | HTTP                            |                                    |

|                                    |     |   |     |
|------------------------------------|-----|---|-----|
| CGI Feature .....                  | 145 | tftp_exec .....                         | 185 |
| Web Pages With HTML Forms .....    | 145 | 8 SMTP Mail Client .....                | 187 |
| Sample HTML Page .....             | 146 | 8.1 Sample Conversation.....            | 187 |
| POST-style form submission ....    | 148 | 8.2 Configuration.....                  | 188 |
| URL-encoded Data .....             | 148 | 8.3 Functions .....                     | 189 |
| Sample of a CGI Handler .....      | 150 | 8.4 Sample Sending of an E-mail.....    | 192 |
| HTML Forms Using Zserver.lib ..... | 152 | 9 POP3 Client.....                      | 193 |
| 4.4 Functions .....                | 158 | 9.1 Configuration.....                  | 193 |
| cgi_redirectto.....                | 158 | 9.2 Three Steps to Receive E-mail. .... | 193 |
| cgi_sendstring.....                | 159 | 9.3 Call-Back Function.....             | 194 |
| http_addfile.....                  | 159 | Normal call-back .....                  | 194 |
| http_contentencode.....            | 160 | POP_PARSE_EXTRA call-back.....          | 194 |
| http_delfile .....                 | 161 | 9.4 Functions .....                     | 195 |
| http_finderrbuf .....              | 161 | pop3_init .....                         | 195 |
| http_nextfverr .....               | 162 | pop3_getmail.....                       | 196 |
| http_handler.....                  | 162 | pop3_tick.....                          | 196 |
| http_init .....                    | 163 | 9.5 Sample receiving of e-mail.....     | 197 |
| http_parseform .....               | 163 | Sample Conversation.....                | 198 |
| http_setcookie.....                | 164 | 10 Telnet.....                          | 199 |
| http_urldecode.....                | 165 | 10.1 Configuration Macros.....          | 199 |
| shtml_addfunction.....             | 166 | 10.2 Functions .....                    | 199 |
| shtml_addvariable .....            | 167 | 10.3 An Example Telnet Server.....      | 201 |
| shtml_delfunction.....             | 168 | A Sample Client To Connect to the       |     |
| shtml_delvariable .....            | 168 | Server.....                             | 202 |
| 5 FTP CLIENT.....                  | 169 | 11 General Purpose Console .....        | 203 |
| 5.1 Configuration Macros.....      | 169 | 11.1 Introduction .....                 | 203 |
| 5.2 Functions .....                | 170 | 11.2 Console Features.....              | 203 |
| ftp_client_setup .....             | 170 | Using other Dynamic C Libraries.....    | 203 |
| ftp_client_tick.....               | 171 | 11.3 Console Commands and Messages..... | 204 |
| ftp_client_filesize .....          | 171 | Console Command Data Structure ..       | 204 |
| 5.3 Sample FTP Transfer.....       | 172 | Help Text for General Cases ....        | 204 |
| 6 FTP Server .....                 | 173 | Console Command Array .....             | 205 |
| 6.1 Configuration Constants .....  | 173 | Console Commands .....                  | 205 |
| File Options .....                 | 173 | Default Command Functions ....          | 206 |
| 6.2 File Handlers .....            | 174 | Custom Console Commands ....            | 209 |
| open .....                         | 174 | Console Error Messages .....            | 210 |
| getfilesize .....                  | 175 | Default Error Messages .....            | 210 |
| read.....                          | 175 | Custom Error Messages .....             | 211 |
| write.....                         | 176 | 11.4 Console I/O Interface.....         | 212 |
| close.....                         | 176 | How to Include an I/O Method.....       | 212 |
| 6.3 Functions .....                | 177 | Predefined I/O Methods.....             | 212 |
| 6.4 Sample FTP Server.....         | 178 | Serial Ports .....                      | 212 |
| 7 TFTP Client.....                 | 179 | Telnet .....                            | 212 |
| BOOTP/DHCP .....                   | 179 | Slave Port .....                        | 213 |
| Data Structure for TFTP.....       | 180 | Custom I/O Methods .....                | 213 |
| Function Reference.....            | 180 | Multiple I/O Streams .....              | 213 |
| TFTP Session .....                 | 180 | 11.5 Console Execution.....             | 213 |
| tftp_init.....                     | 181 | File System Initialization.....         | 214 |
| tftp_initx.....                    | 182 | Serial Buffers .....                    | 214 |
| tftp_tick .....                    | 183 |   |     |
| tftp_tickx .....                   | 184 |   |     |

|                                     |     |
|-------------------------------------|-----|
| Using TCP/IP .....                  | 214 |
| Required Console Functions .....    | 215 |
| console_init.....                   | 215 |
| console_tick .....                  | 215 |
| Console Execution Choices .....     | 215 |
| Terminal Emulator .....             | 215 |
| 11.6 Backup System.....             | 216 |
| Data Structure for Backup System... | 216 |
| Array Definition for Backup System  | 217 |
| 11.7 Console Macros.....            | 217 |
| 11.8 Sample Program.....            | 218 |
| Index.....                          | 225 |

# Introduction 1

This manual is intended for embedded system designers and support professionals who are using an Ethernet-enabled controller board. 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**. It implements IP, TCP, and UDP (User Datagram Protocol). This, along with the libraries **ARP.LIB** and **ICMP.LIB**, make up the transport and network layers of the TCP/IP protocol stack. The remaining libraries implement application-layer protocols.

All user-callable functions are listed and described in their appropriate chapter. Example programs throughout 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.





# TCP/IP Engine 2

This chapter describes the main library file, **DCRTCP.LIB**, which comprises the configuration macros, the data structures and the functions used to initialize and drive TCP/IP. IP version 4 is supported by **DCRTCP.LIB**.

## 2.1 TCP/IP Configuration

To run the TCP/IP engine, a host (i.e., the controller board) needs to know its IP address, netmask and default gateway. If DNS (Domain Name System) lookups are needed, a host will also need to know the IP address of the local DNS server.

### Media Access Control (MAC) address

Some ISPs require that the user provide them with a MAC address for their device. Run the utility program, **samples/tcpip/display\_mac.c**, to display the MAC address of your controller board.

### 2.1.1 IP Addresses Set Manually

The necessary IP addresses can be set at compile time by defining the configuration macros: **MY\_IP\_ADDRESS**, **MY\_NETMASK**, **MY\_GATEWAY** and **MY\_NAMESERVER** respectively. At runtime, the configuration functions, **tcp\_config**, **sethostid** and **sethostname** can override the configuration macros.

### 2.1.2 IP Addresses Set Dynamically

The library **BOOTP.LIB** allows a target board to be a BOOTP or DHCP client. The protocol used depends on what type of server is installed on the local network. BOOTP and DHCP servers are usually centrally located on a local network and operated by the network administrator.

Both protocols allow a number of configuration parameters to be sent to the client, including:

- Client's IP address
- Net mask
- List of gateways
- Host and default domain name
- List of name servers

Both protocols also provide some inessential but useful information:

- Various standard servers, such as NTP, NIS, cookie, etc.
- A bootstrap server address
- The name of a bootstrap file

To use these protocols, include:

```
#define USE_DHCP
#use DCRTCP.LIB
```

in your program.

BOOTP assigns permanent IP addresses. DHCP can “lease” an IP address to a host, i.e., assign the IP address for a limited amount of time. The lease can also be specified as permanent by setting `_dhcplife` to `~0UL` (i.e. `0xFFFFFFFF`).

### 2.1.2.1 BOOTP/DHCP Control Macros

Various macros control the use of DHCP. They must be set before the line `#use "dcrtcp.lib"` in the application program.

#### **USE\_DHCP**

If this macro is defined, the target uses BOOTP or DHCP to configure the required parameters. If `USE_DHCP` is not defined, then `MY_IP_ADDRESS`, `MY_NETMASK`, `MY_GATEWAY` and (possibly) `MY_NAMESERVER` must be defined in the application program.

#### **DHCP\_USE\_BOOTP**

If defined, the target uses the first BOOTP response it gets. If not defined, the target waits for the first DHCP offer and only if none comes in the time specified by `_bootptimeout` does it accept a BOOTP response (if any). Use of this macro speeds up the boot process, but at the expense of ignoring DHCP offers if there is an eager BOOTP server on the local subnet.

#### **DHCP\_CLASS\_ID "Rabbit2000-TCPIP:Z-World:Test:1.0.0"**

This macro defines a class identifier by which the OEM can identify the type of configuration parameters expected. DHCP servers can use this information to direct the target to the appropriate configuration file. Z-World recommends the standard format: “hardware:vendor:product code:firmware” version.

#### **DHCP\_USE\_TFTP**

If this and `USE_DHCP` are defined, the library will use the BOOTP filename and server to obtain an arbitrary configuration file that will be accessible in a buffer at physical address `_bootpdata`, with length, `_bootpsize`. The global variables, `_bootpdone` and `_bootperror` indicate the status of the boot file download. `DHCP_USE_TFTP` should be defined to the maximum file size that may be downloaded.

### 2.1.2.2 BOOTP/DHCP Global Variables

The following list of global variables may be accessed by application code to obtain information about DHCP or BOOTP. These variable are only accessible if `USE_DHCP` is defined.

#### **\_bootpon**

Runtime control of whether to perform DHCP/BOOTP. This is initially set to 'true'. It can be set to false before calling `sock_init` (the function that initializes the TCP/IP engine), causing static configuration to be used. Static configuration uses the values defined for the configuration macros, `MY_IP_ADDRESS` etc. If BOOTP fails during initialization, this will be reset to 0. If reset, then you can call `dhcp_acquire()` at some later time.

**\_survivebootp**

Set to one of the following values:

**0**

### **\_bootpsize**

Indicates how many bytes of the boot file have been downloaded. Only exists if **DHCP\_USE\_TFTP** is defined.

### **\_bootpdata**

Physical starting address of boot data. The length of this area will be **DHCP\_USE\_TFTP** bytes, however, the actual amount of data in the buffer is given by **\_bootpsize**. This variable only exists if **DHCP\_USE\_TFTP** is defined and is only valid if **\_bootpdone** is 1. You can access the data using **xmem2root()** and related functions.

### **\_bootperror**

Indicates any error which occurred in a TFTP process. This variable only exists if **DHCP\_USE\_TFTP** is defined and is only valid when **\_bootpdone** is 1, in which case **\_bootperror** is set to one of the following values (which are also documented with the **tftp\_tick()** function):

- 0: No error.
- 1: Error from boot file server, transfer terminated. This usually occurs because the server is not configured properly, and has denied access to the nominated file.
- 2: Error, could not contact boot file server or lost contact.
- 3: Timed out, transfer terminated.
- 4: (not used)
- 5: Transfer complete, but truncated because buffer too small to receive the complete file.

### 2.1.2.3 BOOTP/DHCP Functions

#### **dhcp\_acquire**

```
int dhcp_acquire( void );
```

#### **DESCRIPTION**

This function acquires a DHCP lease which has not yet been obtained, or has expired, or was relinquished using `dhcp_release()`. Normally, DHCP leases are renewed automatically, however if the DHCP server is down for an extended period then it might not be possible to renew the lease in time, in which case the lease expires and TCP/IP should not be used. When the lease expires, `tcp_tick()` will return 0, and the global variable for the IP address will be reset to 0. At some later time, this function can be called to try to obtain an IP address.

This function blocks until the lease is renewed, or the process times out.

#### **RETURN VALUE**

**0**: OK, lease was not expired, or an IP address lease was acquired with the same IP address as previously obtained.

**-1**: An error occurred, no IP address is available. TCP/IP functionality is thus not available. Usual causes of an error are timeouts because a DHCP or BOOTP server is not available within the timeout specified by the global variable `_bootptimeout` (default 30 seconds).

**1**: Lease was re-acquired, however the IP address differs from the one previously obtained. All existing sockets must be re-opened. Normally, DHCP servers are careful to re-assign the same IP address previously used by the client, however this is sometimes not possible.

#### **LIBRARY**

BOOTP.LIB

```
int dhcp_release( void );
```

#### DESCRIPTION

This function relinquishes a lease obtained from a DHCP server. This allows the server to re-use the IP address which was allocated to this target. After calling this function, the global variable for the IP address is set to 0, and it is not possible to call any other TCP/IP function which requires a valid IP address. Normally, `dhcp_release()` would be used on networks where only a small number of IP addresses are available, but there are a large number of hosts which need sporadic network access.

This function is non-blocking since it only sends one packet to the DHCP server and expects no response.

#### RETURN VALUE

**0:** OK, lease was relinquished.

**1:** Not released, because an address is currently being acquired, or because a boot file (from the BOOTP or DHCP server) is being downloaded, or because some other network resource is in use e.g. open TCP socket. Call `dhcp_release()` again after the resource is freed.

**-1:** Not released, because DHCP was not used to obtain a lease, or no lease was acquired.

#### LIBRARY

BOOTP.LIB

#### 2.1.2.4 DHCP Sample Program

The following sample is a very basic TCP/IP program, that will initialize the TCP/IP interface, and allow the device to be 'pinged' from another computer on the network. DHCP or BOOTP will be used to obtain IP addresses and other network sample

```

// Main define to cause BOOTP or DHCP to be used.
#define USE_DHCP

/* These values may be used as a fallback if _survivebootp is set true.
Otherwise, they will be ignored. Note that in a 'real' application,
setting fallbacks as hard-coded addresses would be unwise.*/

#define MY_IP_ADDRESS "10.10.6.179"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"

#memmap xmem
#use dcrtcp.lib

/* Print some of the DHCP or BOOTP parameters received. */
void print_results(void){
    printf("Network Parameters:\r\n");
    printf(" My IP Address = %08lX\r\n", my_ip_addr);
    printf(" Netmask = %08lX\r\n", sin_mask);
    if (_dhcphost != ~0UL) {
        if (_dhcptime == DHCP_ST_PERMANENT) {
            printf(" Permanent lease\r\n");
        } else {
            printf("Remaining lease= %ld (sec)\r\n", _dhcplife -
                SEC_TIMER);
            printf("Renew lease in %ld (sec)\r\n", _dhcpt1 - SEC_TIMER);
        }
        printf(" DHCP server = %08lX\r\n", _dhcphost);
        printf(" Boot server = %08lX\r\n", _bootphost);
    }
    if (gethostname(NULL,0))
        printf(" Host name = %s\r\n", gethostname(NULL,0));
    if (getdomainname(NULL,0))
        printf(" Domain name = %s\r\n", getdomainname(NULL,0));
}

main(){
    _survivebootp = 1; // So we can print our own message
    _bootptimeout = 6; // Short timeout for testing
    sock_init();
    if (_dhcphost != ~0UL)
        printf("Lease obtained\r\n");
    else {
        printf("Lease not obtained. DHCP server may be down.\r\n");
        printf("Using fallback parameters...\r\n");
    }
    print_results();
    for (;;)
        tcp_tick(NULL);
}

```

### **2.1.3 Sizes for TCP/IP I/O Buffers**

There are two macros that can define the size of the buffer that is used for UDP



To handle multiple simultaneous connections, each new connection will require its own `tcp_socket` and a separate call to `tcp_listen()`, but using the same local port number (`lport` value). `tcp_listen()` will immediately return, and you must poll for the incoming connection. You can use the `sock_wait_established` macro, which calls `tcp_tick()` and blocks until the connection is established or manually poll the socket using `sock_established()`.

### 2.2.3 Active Open

When your Web browser retrieves a page, it actively opens one or more connections to the server's passively opened sockets. To actively open a connection, you call `tcp_open()`, which uses parameters that are similar to the ones used in `tcp_listen()`. Supply exact parameters for `ina` and `port`, which are the IP address and port number you want to connect to; the `lport` parameter can be zero, which tells `DCR_TCP_LIB` to select an unused local port between 1024 and 65535. If `tcp_open()` returns zero, no connection was made. This could be due to routing difficulties, such as an inability to resolve the remote computer's hardware address with ARP.

### 2.2.4 Delay a Connection

To accept a connection request when the resources to actually process the request are not available, use the function `tcp_reserveport()`. It takes one parameter, the port number where you want to accept connections. When a connection to that port number is requested, the 3-way handshaking is done even if there is not yet a socket available. When replying to the connection request, the window parameter in the TCP header is set to zero, meaning, "I can take no bytes of data at this time." The other side of the connection will wait until the value in the window parameter indicates that data can be sent. Using the companion function, `tcp_clearreserve(port number)`, causes TCP/IP to treat a connection request to the port in the conventional way. The macro `USE_RESERVEDPORTS` is defined by default. It allows the use of these two functions.

When using `tcp_reserveport`, the 2MSL (Maximum Segment Lifetime) waiting period for closing a socket is avoided.

### 2.2.5 TCP Socket Functions

There are many functions that can be applied to an open TCP socket. They fall into three main categories: Control, Status, and I/O.

#### 2.2.5.1 Control Functions

These functions change the status of the socket or its I/O buffer.

- `sock_abort`
- `sock_close`
- `sock_flush`
- `sock_flushnext`
- `tcp_listen`
- `tcp_open`

`tcp_open()` and `tcp_listen()` have been explained in previous sections.

Call `sock_close()` to end a connection. This call may not immediately close the connection because it may take some time to send the request to end the connection and receive the acknowledgements. If you want to be sure that the connection is completely closed before continuing, call `tcp_tick()` with the socket structure's address. When `tcp_tick()` returns zero, then the

socket is completely closed. Please note that if there is data left to be read on the socket, the socket will not completely close.

Call `sock_abort()` to cancel an open connection. This function will cause a TCP reset to be sent to the other end, and all future packets received on this connection will be ignored.

For performance reasons, data may not be immediately sent from a socket to its destination. If your application requires the data to be sent immediately, you can call `sock_flush()`. This function will cause `DCRTCP.LIB` to try sending any pending data immediately. If you know ahead of time that data needs to be sent immediately, call `sock_flushnext()` on the socket. This function will cause the next set of data written to the socket to be sent immediately, and is more efficient than `sock_flush()`.

### 2.2.5.2 Status Functions

These functions return useful information about the status of either a socket or its I/O buffers.

- `sock_bytesready`
- `sock_dataready`
- `sock_established`
- `sock_rbleft`
- `sock_rbsize`
- `sock_rbused`
- `sock_tbleft`
- `sock_tbsize`
- `sock_tbused`
- `tcp_tick`

`tcp_tick()` is the daemon that drives the TCP/IP engine, but it also returns status information. When you supply `tcp_tick()` with a pointer to a `tcp_socket` (a structure that identifies a particular socket), it will first process packets and then check the indicated socket for an established connection. `tcp_tick()` returns zero when the socket is completely closed. You can use this return value after calling `sock_close()` to determine if the socket is completely closed.

```
sock_close(&my_socket);
while(tcp_tick(&my_socket)) {
    // you can do other things here while waiting for the socket
    // to be completely closed.
}
```

These status functions can be used to avoid blocking when using `sock_write()` and some of the other I/O functions, as illustrated in the following code.

This block of code checks to make sure that there is enough room in the buffer before adding data with a blocking function. .

```
if(sock_tbleft(&my_socket,size)) {
    sock_write(&my_socket,buffer,size);
}
```

This block of code ensures that there is a string terminated with a new line in the buffer, or that the buffer is full before calling `sock_gets()`:

```
sock_mode(&my_socket, TCP_MODE_ASCII);
if(sock_bytesready(&my_socket) != -1) {
    sock_gets(buffer, MAX_BUFFER);
}
```

### 2.2.5.3 I/O Functions

- `sock_fastread`
- `sock_fastwrite`
- `sock_getc`
- `sock_gets`
- `sock_preread`
- `sock_putc`
- `sock_puts`
- `sock_read`
- `sock_write`

There are two modes of reading and writing to TCP sockets: ASCII and binary. By default, a socket is opened in binary mode, but you can change that with a call to `sock_mode()`.

When a socket is in ASCII mode, `DCR_TCP.LIB` assumes that the data is an ASCII stream with record boundaries on the newline characters for some of the functions. This behavior means `sock_bytesready()` will return  $\geq 0$  only when a complete newline-terminated string is in the buffer or the buffer is full. The `sock_puts()` function will automatically place a newline character at the end of a string, and the `sock_gets()` function will strip the newline character.

When in binary mode, do not use the `sock_scanf` (currently not implemented) or the `sock_gets()` functions.

## 2.3 UDP I/O Interface

The UDP protocol is useful when sending messages where either a lost message does not cause a system failure or is handled by the application. Since UDP is a simple protocol and you have control over the retransmissions, you can decide if you can trade low latency for high reliability.

- `sock_fastread`
- `sock_fastwrite`
- `sock_getc`
- `sock_gets`
- `sock_preread`
- `sock_putc`
- `sock_puts`
- `sock_read`
- `sock_recv`
- `sock_recv_from`
- `sock_recv_init`
- `sock_write`
- `udp_open`

Notice that there are three additional I/O functions that are only available for use with UDP sockets: `sock_recv()`, `sock_recv_from()` and `sock_recv_init()`. The status and con-

control functions that are available for TCP sockets also work for UDP sockets, with the exception of the open functions, `tcp_listen()` and `tcp_open()`.

## Broadcast Packets

UDP can send broadcast packets (i.e., to send a packet to a number of computers on the same network). When done properly, broadcasts can reduce overall network traffic because information does not have to be duplicated when there are multiple destinations.

### 2.3.1 Opening and Closing a UDP Socket

The `udp_open` function takes a remote IP address and a remote port number. If they are set to a specific value, all incoming and outgoing packets are filtered on that value (i.e., you talk only to the one remote address).

If the remote IP address is set to -1, the UDP socket receives packets from any valid remote address, and outgoing packets are broadcast. If the remote IP address is set to 0, no outgoing packets may be sent until a packet has been received. This first packet completes the socket, filling in the remote IP address and port number with the return address of the incoming packet. Multiple sockets can be opened on the same local port, with the remote address set to 0, to accept multiple incoming connections from separate remote hosts. When you are done communicating on a socket that was started with a 0 IP address, you can close it with `sock_close()` and reopen to make it ready for another source.

### 2.3.2 Writing to a UDP Socket

The normal socket functions you used for writing to a TCP socket will work for a UDP socket, but since UDP is a significantly different service, the result could be different. Each atomic write—`sock_putc()`, `sock_puts()`, `sock_write()`, or `sock_fastwrite()`—places its data into a single UDP packet. Since UDP does not guarantee delivery or ordering of packets, the data received may be different either in order or content than the data sent. Packets may also be duplicated if they cross any gateways. A duplicate packet may be received well after the original.

### 2.3.3 Reading From a UDP Socket

There are two ways to read packets using `DCR_TCP.LIB`. The first method uses the same read functions that are used for TCP: `sock_getc()`, `sock_gets()`, `sock_read()`, and `sock_fastread()`. These functions will read the data as it came into the socket, which is not necessarily the data that was written to the socket.

The second mode of operation for reading uses the `sock_recv_init()`, `sock_recv()`, and `sock_recv_from()` functions. The `sock_recv_init()` function installs a large buffer area that gets divided into smaller buffers. Whenever a datagram arrives, `DCR_TCP.LIB` stuffs that datagram into one of these new buffers. The `sock_recv()` and `sock_recv_from()` functions scan these buffers. After calling `sock_recv_init` on the socket, you should not use `sock_getc()`, `sock_read()`, or `sock_fastread()`.

The `sock_recv()` function scans the buffers for any datagrams received by that socket. If there is a datagram, the length is returned and the user buffer is filled, otherwise `sock_recv()` returns zero.

The `sock_recv_from()` function works like `sock_recv()`, but it allows you to record the IP address where the datagram originated. If you want to reply, you can open a new UDP socket with the IP address modified by `sock_recv_from()`.

### 2.3.4 Checksums

There is an optional checksum field inside the UDP header. This field verifies only the header portion of the packet and doesn't cover the data. This feature can be disabled on a reliable network where the application has the ability to detect transmission errors. Disabling the UDP checksum can increase the performance of UDP packets moving through `DCRTCP.LIB`. This feature can be modified by:

```
sock_mode(s, UDP_MODE_CHK);    // enable checksums
sock_mode(s, UDP_MODE_NOCHK);  // disable checksums
```

The first parameter is a pointer to the socket's data structure, either `tcp_Socket` or `udp_Socket`.

## 2.4 Skeleton Program

The following program is a general outline for a Dynamic C TCP/IP program. The first couple of defines set up the default IP configuration information. The "memmap" line causes the program to compile as much code as it can in the extended code window. The "use" line causes the compiler to compile in the Dynamic C TCP/IP code using the configuration data provided above it.

**Pingme.c:**

```
#define MY_IP_ADDRESS "10.10.6.101"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.19"
#memmap xmem
#use dcrtcp.lib
main() {
    sock_init();
    for (;;) {
        tcp_tick(NULL);
    }
}
```

To run this program, start Dynamic C and open the `SAMPLES\TCPIP\ICMP\PINGME.C` file. Edit the `MY_IP_ADDRESS`, `MY_NETMASK`, and `MY_GATEWAY` macros to reflect the appropriate values for your device. Run the program and try to run `ping 10.10.6.101` from a command line on a computer on the same physical network, replacing `10.10.6.101` with your value for `MY_IP_ADDRESS`.

### 2.4.1 TCP/IP Stack Initialization

The `main()` function first initializes the TCP/IP stack with a call to `sock_init()`. This call initializes internal data structures and enables the Ethernet chip, which will take a couple of seconds with the RealTek chip. At this point, `DCRTCP.LIB` is ready to handle incoming packets.

## 2.4.2 Packet Processing

Incoming packets are processed whenever `tcp_tick()` is called. The user-callable functions that call `tcp_tick()` are: `tcp_open`, `udp_open`, `sock_read`, `sock_write`, `sock_close`, and `sock_abort`. Some of the higher-level protocols, e.g. `HTTP.LIB`, will call `tcp_tick()` automatically.

It is a good practice to make sure that `tcp_tick()` is called periodically in your program to insure that the TCP/IP engine has had a chance to process packets. A rule of thumb is to call `tcp_tick()` around 10 times per second, although slower or faster call rates should also work. The Ethernet interface chip has a large buffer memory, and TCP/IP is adaptive to the data rates that both end of the connection can handle; thus the system will generally keep working over a wide variety of tick rates.

## 2.4.3 TCP/IP Daemon Computing Time

The computing time consumed by each call to `tcp_tick()` varies. Rough numbers are less than a millisecond if there is nothing to do, 10s of milliseconds for typical packet processing, and 100s of milliseconds under exceptional circumstances.

## 2.5 State-Based Program Design

An efficient design strategy is to create a state machine within a function and pass the socket's data structure as a function parameter. This method allows you to handle multiple sockets without the services of a multitasking kernel. This is the way the `HTTP.LIB` functions are organized. Many of the common Internet protocols fit well into this state machine model.

The general states are:

- Waiting to be initialized
- Waiting for a connection
- Connected states that perform the real work
- Waiting for the socket to be closed

An example of state-based programming is `SAMPLES\TCPIP\STATE.C`. This program is a basic Web server that should work with most browsers. It allows a single connection at a time, but can be extended to allow multiple connections.

### 2.5.1 Blocking vs. Non-Blocking

There is a choice between blocking and non-blocking functions when doing socket I/O.

#### 2.5.1.1 Non-Blocking Functions

The `sock_fastread()` and `sock_preread()` functions read as much data as is available in the buffers, and return immediately. Similarly, the `sock_fastwrite()` function fills the buff-

ers and returns the number of characters that were written. When using these functions, you must ensure that all of the data were written completely.

```
offset=0;
while(offset<length) {
    bytes_written=sock_fastwrite(&socket,buffer+offset,length-offset);
    if(bytes_written<0) {
        // error handling
    }
    offset+=bytes_written;
}
```

### 2.5.1.2 Blocking Functions

The other functions (`sock_getc()`, `sock_gets()`, `sock_putc()`, `sock_puts()`, `sock_read()` and `sock_write()`) do not return until they have completed or there is an error. If it is important to avoid blocking, you can check the conditions of an operation to insure that it will not block.

```
sock_mode(socket,TCP_MODE_ASCII);
// ...
if (sock_bytesready(&my_socket) != -1){
    sock_gets(buffer,MAX_BUFFER);
}
```

In this case `sock_gets()` will not block because it will be called only when there is a complete new line terminated record to read.

### 2.5.1.3 Blocking Macros

To block at a certain point and wait for a condition, `DCRTCP.LIB` provides the macros `sock_wait_closed`, `sock_wait_established` and `sock_wait_input`, to make this task easier.

In this program fragment, `sock_wait_established` is used to block the program until a connection is established. Notice the timeout (second parameter) value of zero. This tells Dynamic C to never timeout. Associated with these macros is a `sock_err` label to jump to when there is an error. If you supply a pointer to a status integer, it will set the status to an error code. Valid error codes are `-1` for timeout and `1` for a reset connection.

```

tcp_open(&s,0,ip,PORT,NULL);
sock_wait_established(&s,0,NULL,&status);

//...

sock_err:
switch(status) {
    case 1: /* foreign host closed */
        printf("User closed session\n");
        break;
    case -1: /* timeout */
        printf("\nConnection timed out\n");
        break;
}

```

## 2.6 Multitasking and TCP/IP

Dynamic C's TCP/IP implementation is compatible with both  $\mu$ C/OS-II and with the language constructs that implement cooperative multitasking: costatements and cofunctions. Note that TCP/IP is not compatible with the slice statement.

### 2.6.1 $\mu$ C/OS-II

The TCP/IP engine may be used with the  $\mu$ C/OS-II real-time kernel. The line

```
#use ucos2.lib
```

must appear before the line

```
#use dcrtcp.lib
```

### 2.6.2 Cooperative Multitasking

The following program demonstrates the use of multiple TCP sockets with costatements. After compiling and running the program, make the following telnet connections using your own IP address:

```
telnet 10.10.6.11 8888
```

```
telnet 10.10.6.11 8889
```



```

#define MY_IP_ADDRESS "10.10.6.11"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"

#define PORT1 8888
#define PORT2 8889

#define SOCK_BUF_SIZE 2048
#define MAX_SOCKETS 2

#memmap xmem
#use "dcrtcp.lib"

tcp_Socket Socket_1;
tcp_Socket Socket_2;

#define MAX_BUFSIZE 512
char buf1[MAX_BUFSIZE], buf2[MAX_BUFSIZE];

// The function that actually does the TCP work
cofunc int basic_tcp[2](tcp_Socket *tcp_sock, int port, char *buf){
    auto int length, space_avaliable;
    auto sock_type *s;
    s = (sock_type *)tcp_sock;

    tcp_listen(tcp_sock, port, 0, 0, NULL, 0);

    // wait for a connection
    while((-1 == sock_bytesready(s)) && (0 == sock_established(s))) {
        // give other tasks time to do things while we are waiting
        yield;
    }
    while(sock_established(s)) {
        space_avaliable = sock_tbleft(s);
        // limit transfer size to MAX_BUFSIZE, leave room for '\0'
        if(space_avaliable > (MAX_BUFSIZE-1))
            space_avaliable = (MAX_BUFSIZE-1);
        // get some data
        length = sock_fastread(s, buf, space_avaliable);

        if(length > 0) {
            // did we receive any data?
            buf[length] = '\0'; // print it to the stdio window
            printf("%s",buf);
            // send it back out to the user's telnet session
            // sock_fastwrite will work-we verified the space beforehand
            sock_fastwrite(s, buf, length);
        }
        yield; // give other tasks time to run
    }
    sock_close(s);
    return 1;
}

```

```
main() {
    sock_init();
    while (1) {
        costate {
            // Go do the TCP/IP part, on the first socket
            wfd basic_tcp[0](&Socket_1, PORT1, buf1);
        }
        costate {
            // Go do the TCP/IP part, on the second socket
            wfd basic_tcp[1](&Socket_2, PORT2, buf2);
        }
        costate {
            // drive the tcp stack
            tcp_tick(NULL);
        }
        costate {
            // Can insert application code here!
            waitfor(DelayMs(100));
        }
    }
}
```

## 2.7 Function Reference

This section contains descriptions for all user-callable functions in `DCRTCP.LIB`. Descriptions for select user-callable functions in `ARP.LIB`, `ICMP.LIB`, `BSDNAME.LIB` and `XMEM.LIB` are also included. Note that `ARP.LIB`, `ICMP.LIB` and `BSDNAME.LIB` are automatically `#use'd` from `DCRTCP.LIB`.

### `_arp_resolve`

```
int _arp_resolve(longword ina, eth_address *ethap, int nowait);
```

#### DESCRIPTION

Gets the Ethernet address for the given IP address.

#### PARAMETERS

|                     |  |
|---------------------|--|
| <code>ina</code>    | The IP address to resolve to an Ethernet address.  |
| <code>ethap</code>  | The buffer to hold the Ethernet address.   |
| <code>nowait</code> | If 0, return immediately; else if !0 wait up to 5 seconds trying to resolve the address. |

#### RETURN VALUE

1: Success;  
0: Failure.

#### LIBRARY

`ARP.LIB`

## `_chk_ping`

```
longword _chk_ping( longword host_ip, longword
    *sequence_number );
```

### DESCRIPTION

Checks for any outstanding ping replies from host. `_chk_ping` should be called frequently with a host IP address. If an appropriate packet is found from that host IP address, the sequence number is returned through `*sequence_number`. The time difference between our request and their response is returned in milliseconds.

### PARAMETERS

|                              |  |
|------------------------------|--|
| <code>host_ip</code>         | IP address to receive ping reply from. |
| <code>sequence_number</code> | Sequence number of reply.              |

### RETURN VALUE

Time in milliseconds from the ping request to the host's ping reply.  
If `_chk_ping` returns `0xffffffffL`, there were no ping receipts on this current call.

### LIBRARY

`ICMP.LIB`

## getdomainname

```
char * getdomainname( char *name, int length );
```

### DESCRIPTION

Gets the current domain name. The domain name can be changed by the **setdomainname** function.

### PARAMETERS

|               |  |
|---------------|--|
| <b>name</b>   | Buffer to place the name.  |
| <b>length</b> | Max length of the name, or zero to get pointer to internal domain name string. |

### RETURN VALUE

If **length**  $\geq$  1 return pointer to **name**. If **length** is not long enough to hold the domain name, a **NULL** string is written to **name**.

If **length** = 0 return pointer to internal string containing the domain name. Do not modify this string!

### LIBRARY

BSDNAME.LIB

### SEE ALSO

setdomainname, gethostname, sethostname, getpeername, getsockname

### EXAMPLE

```
main() {
    sock_init();
    printf("Using %s for a domain\n", getdomainname(NULL, 0));
}
```

```
longword gethostid( void );
```

**DESCRIPTION**

Return the IP address of the controller in host format.

**RETURN VALUE**

IP address in host format, or zero if not assigned or not valid.

**LIBRARY**

DCRTCP.LIB

**SEE ALSO**

sethostid

**EXAMPLE**

```
char * gethostname( char *name, int length );
```

**DESCRIPTION**

Gets the host portion of our name.

**PARAMETERS**

|               |   |
|---------------|---|
| <b>name</b>   | Buffer to place the name.                     |
| <b>length</b> | Max length of the name, or zero for internal. |

**RETURN VALUE**

If length >=1, return name;  
else if length = 0, return internal host name buffer

## getpeername

```
int getpeername( sock_type * s, void * dest, int * len );
```

### DESCRIPTION

Gets the peer's IP address and port information for the specified socket.

### PARAMETERS

**s** Pointer to the socket.

**dest** Pointer to **sockaddr** to hold the socket information for the remote end of the socket. The data structure is:

```
typedef struct sockaddr {  
    word      s_type;      /* reserved */  
    word      s_port;      /* port number, or zero if not connected */  
    longword  s_ip;        /* IP address, or zero if not connected */  
    byte      s_spares[6]; /* not used for tcp/ip connections */  
};
```

**len** Pointer to the length of **sockaddr**. A **NULL** pointer can be used to represent the **sizeof(struct sockaddr)**.

### RETURN VALUE

0: success;  
-1: failure.

### LIBRARY

BSDNAME.LIB

### SEE ALSO

getsockname

## getsockname

```
int getsockname( sock_type * s, void * dest, int * len );
```

### DESCRIPTION

Gets the controller's IP address and port information for a particular socket.

### PARAMETERS

**s** Pointer to the socket.

**dest** Pointer to **sockaddr** to hold the socket information for the local end of the socket. The data structure is:

```
typedef struct sockaddr {
    word      s_type;      /* reserved */
    word      s_port;      /* port number, or zero if not connected */
    longword  s_ip;        /* IP address, or zero if not connected */
    byte      s_spares[6]; /* not used for tcp/ip connections */
};
```

**len** Pointer to the length of **sockaddr**. A **NULL** pointer can be used to represent the **sizeof(struct sockaddr)**. **BSDNAME.LIB** will assume 14 bytes if a **NULL** pointer is passed.

### RETURN VALUE

0: Success;  
-1: Failure.

### LIBRARY

BSDNAME.LIB

### SEE ALSO

getpeername



## htonl

```
longword htonl( longword value );
```

### DESCRIPTION

This function converts a host-ordered double word to a network-ordered double word. This function is necessary if you are implementing standard internet protocols because the Rabbit does not use the standard for network byte ordering. The network orders bytes with the most significant byte first and the least significant byte last. On the Rabbit, the bytes are in the opposite order.

### PARAMETERS

**value**                    Host-ordered double word.

### RETURN VALUE

Host word in network format, e.g. **htonl(0x44332211)** returns 0x11223344.

### LIBRARY

DCR<sub>T</sub>CP.LIB

### SEE ALSO

htons, ntohl, ntohs

## htons

```
word htons( word value );
```

### DESCRIPTION

Converts host-ordered word to a network-ordered word. This function is necessary if you are implementing standard internet protocols because the Rabbit does not use the standard for network byte ordering. The network orders bytes with the most significant byte first and the least significant byte last. On the Rabbit, the bytes are in the opposite order within each 16-bit section.

### PARAMETERS

**value**                    Host-ordered word.

### RETURN VALUE

Host-ordered word in network-ordered format, e.g. **htons(0x1122)** returns 0x2211.

### LIBRARY

DCR<sub>T</sub>CP.LIB

### SEE ALSO

htonl, ntohl, ntohs

## **inet\_addr**

```
longword inet_addr( char * dotted_ip_string );
```

### **DESCRIPTION**

Converts an IP address from dotted decimal IP format to its binary representation. No check is made as to the validity of the address.

### **PARAMETERS**

**dotted\_ip\_string**    Dotted decimal IP string, e.g. "10.10.6.100".

### **RETURN VALUE**

0: Failure;  
Binary representation of **dotted\_ip\_string**: Success.

### **LIBRARY**

DCRTPC.LIB

### **SEE ALSO**

inet\_ntoa

## inet\_ntoa

```
char *inet_ntoa( char *s, longword ip );
```

### DESCRIPTION

Converts a binary IP address to its dotted decimal format, e.g.

`inet_ntoa(s, 0x0a0a0664)` returns a pointer to "10.10.6.100".

### PARAMETERS

|           |  |
|-----------|--|
| <b>s</b>  | Location to place the dotted decimal string. A sufficient buffer size would be 16 bytes. |
| <b>ip</b> | The IP address to convert.   |

### RETURN VALUE

Pointer to dotted decimal string, i.e. **s**.

### LIBRARY

DCR<sub>T</sub>CP.LIB

### SEE ALSO

inet\_addr

## ip\_timer\_expired

```
word ip_timer_expired( void * s );
```

### DESCRIPTION

Checks the timer field (set by `ip_timer_init()`) inside the socket structure. This function is used in the `sock_wait_...` macros to provide timeouts.

### PARAMETERS

|          |                      |
|----------|----------------------|
| <b>s</b> | Pointer to a socket. |
|----------|----------------------|

### RETURN VALUE

0: Not expired;  
1: Expired.

### LIBRARY

DCR<sub>T</sub>CP.LIB

## EXAMPLE USING IP\_TIMER\_EXPIRED

The following code is from a blocking configuration macro that calls the function `_ip_delay2`.

```
_ip_delay2( void *s, int timeoutseconds, procref fn, int *statusptr) {
    int status;
    ip_timer_init( s , timeoutseconds ); /* set timeout */
    do {
        kbhit(); /* permit ^c */
        if ( !tcp_tick( s )) {
            status = 1; /* fully closed or reset */
            break;
        }
        if ( ip_timer_expired( s )) { /* check for expiration */
            sock_abort( s ); /* give up and use reset */
            status = -1; /* signal an error */
            break;
        }
        if (fn) { /* call optional user function */
            if (status = fn(s))
                break;
        }
        if ( s->tcp.usr_yield )
            (*s->tcp.usr_yield)(); /* call yield */
    } while ( 1 );
    if (statusptr) *statusptr = status;
    return( status );
}
```

## ip\_timer\_init

```
void ip_timer_init( void * s, word seconds );
```

### DESCRIPTION

Sets a timer inside the socket structure.

### PARAMETERS

|                |  |
|----------------|--|
| <b>s</b>       | Pointer to a socket.   |
| <b>seconds</b> | Number of seconds for the time-out, if this value is zero, never time-out. |

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

ip\_timer\_expired

## ntohl

```
longword ntohl( longword value );
```

### DESCRIPTION

Converts network-ordered long word to host-ordered long word. This function is necessary if you are implementing standard internet protocols because the Rabbit does not use the standard for network byte ordering. The network orders bytes with the most significant byte first and the least significant byte last. On the Rabbit, the bytes are in the opposite order.

### PARAMETERS

|              |                            |
|--------------|----------------------------|
| <b>value</b> | Network-ordered long word. |
|--------------|----------------------------|

### RETURN VALUE

Network-ordered long word in host-ordered format,  
e.g. `ntohl(0x44332211)` returns `0x11223344`

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

htons, ntohs, htonl

## ntohs

```
word ntohs( word value );
```

### DESCRIPTION

Converts network-ordered word to host-ordered word. Converts host-ordered word to a network-ordered word. This function is necessary if you are implementing standard internet protocols because the Rabbit does not use the standard for network byte ordering. The network orders bytes with the most significant byte first and the least significant byte last. On the Rabbit, the bytes are in the opposite order.

### PARAMETERS

**value**                Network-ordered word.

### RETURN VALUE

Network-ordered word in host-ordered format,  
e.g. `ntohs(0x2211)` returns `0x1122`

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

htonl, ntohl, htons

## paddr

```
unsigned long paddr(void* pointer);
```

### DESCRIPTION

Converts a logical pointer into its physical address. Use caution when converting address in the E000-FFFF range. This function will return the address based on the XPC on entry.

### PARAMETERS

**pointer**              Pointer to convert.

### RETURN VALUE

Physical address of pointer.

### LIBRARY

XMEM.LIB

## pd\_getaddress

```
void pd_getaddress(int nic, void* buffer);
```

### DESCRIPTION

This function copies the Ethernet address (e.g., MAC address) into the buffer.

### PARAMETERS

|               |  |
|---------------|--|
| <b>nic</b>    | This parameter is reserved for future expandability and for now should be assigned a value of 0. |
| <b>buffer</b> | Place to copy address to. Must be at least 6 bytes.  |

### RETURN VALUE

None

### LIBRARY

PKTDRV.LIB

### EXAMPLE

```
main() {
    char buf[6];
    sock_init();
    pd_getaddress(0,buf);

    printf("Your Link Address is:%02x%02x:%02x%02x:%02x%02x \n",
          buf[0], buf[1], buf[2], buf[3], buf[4], buf[5]);
}
```

## **`_ping`**

```
int _ping( longword host_ip, longword sequence_number );
```

### **DESCRIPTION**

Generates an ICMP request for host. NOTE: this is a macro that calls `_send_ping`.

### **PARAMETERS**

|                              |                               |
|------------------------------|-------------------------------|
| <code>host_ip</code>         | IP address to send ping.      |
| <code>sequence_number</code> | User-defined sequence number. |

### **RETURN VALUE**

0: Success;  
!0: Failure.

### **LIBRARY**

ICMP.LIB

### **SEE ALSO**

`_chk_ping`, `_send_ping`

## **`psocket`**

```
void psocket( void * s );
```

### **DESCRIPTION**

Given an open UDP or TCP socket, the IP address of the remote host is printed out to the Stdio window in dotted IP format followed by a colon and the decimal port number on that machine. This routine can be useful for debugging your programs.

### **PARAMETERS**

|                |                      |
|----------------|----------------------|
| <code>s</code> | Pointer to a socket. |
|----------------|----------------------|

### **RETURN VALUE**

None.

### **LIBRARY**

BSDNAME.LIB



## resolve

```
longword resolve( char *host_string );
```

### DESCRIPTION

Converts a text string, which contains either the dotted IP address or host name, into the longword containing the IP address. In the case of dotted IP, no validity check is made for the address. NOTE: this function blocks. Names are currently limited to 128 characters. If it is necessary to lookup larger names include the following line in the application program:

```
#define MAX_DOMAIN_LENGTH <len in chars>.
```

If `DISABLE_DNS` has been defined this function will not do DNS lookup.

If you are trying to resolve a host name, you must set up at least one name server. You can set the default name server by defining the `MY_NAMESERVER` macro at the top of your program. When you call `resolve()`, it will contact the name server and request the IP address. If there is an error, `resolve()` will return 0L.

To simply convert dotted IP to longword, see `inet_addr()`.

For a sample program, see the Example Using `tcp_open()` listed under `tcp_open()`.

### PARAMETERS

`host_string` Pointer to text string to convert.

### RETURN VALUE

0 if failure,  
!0 is the IP address `*host_string` resolves to.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

`_arp_resolve`, `inet_addr`, `inet_ntoa`

## rip

```
char * rip( char * string );
```

### DESCRIPTION

Strips newline (\n) and/or carriage return (\r) from a string. Only the first \n and \r characters are replaced with \0s. The resulting string beyond the first \0 character is undefined.

### PARAMETERS

**string**            Pointer to a string.

### RETURN VALUE

Pointer to the modified string.

### LIBRARY

DCRTPC.LIB

### EXAMPLE

```
setmode( s, TCP_MODE_ASCII );  
...  
sock_puts( s, rip( questionable_string ) );
```

In ASCII mode **sock\_puts()** adds \n; **rip** is used to make certain the string does not already have a newline character. Remember, **rip** modifies the source string, not a copy!

## `_send_ping`

```
int _send_ping( longword host, longword countnum, byte ttl, byte
               tos, longword *theid )
```

### DESCRIPTION

Generates an ICMP request for host.

### PARAMETERS

|                       |   |
|-----------------------|---|
| <code>host</code>     | IP address to send ping.  |
| <code>countnum</code> | User-defined count number.  |
| <code>ttl</code>      | Time to live for the packets (hop count). 255 is a standard value for this field. |
| <code>tos</code>      | Type of service on the packets.   |
| <code>theid</code>    | The identifier that was sent out.   |

### RETURN VALUE

0: Successful;  
-1: Failure.

### LIBRARY

ICMP.LIB

### See also

`_chk_ping`, `_ping`

## setdomainname

```
char *setdomainname( char *name );
```

### DESCRIPTION

The domain name returned by `getdomainname()` and used for `resolve()` is set to the value in the string pointed to by `name`. Changing the contents of the string after a `setdomainname()` will change the value of the system domain string. It is not recommended. Instead dedicate a static location for holding the domain name.

`setdomainname( NULL )` is an acceptable way to remove any domain name and subsequent `resolve` calls will not attempt to append a domain name.

### PARAMETERS

`name`                    Pointer to string.

### RETURN VALUE

Pointer to string that was passed in.

### LIBRARY

BSDNAME.LIB

### SEE ALSO

`getdomainname`, `sethostname`, `gethostname`, `getpeername`,  
`getsockname`

## sethostid

```
longword sethostid( longword ip );
```

### DESCRIPTION

This function changes the system's default IP address, overriding the macro **MY\_IP\_ADDRESS**. Changing this address will disrupt existing TCP or UDP sessions. You should close all sockets before calling this function.

### PARAMETERS

**ip**                      New IP address.

### RETURN VALUE

New IP address.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

gethostid

## sethostname

```
char * sethostname( char *name);
```

### DESCRIPTION

Sets the host portion of our name.

### PARAMETERS

**name**                    The new host name.

### RETURN VALUE

Pointer to internal hostname buffer on success, or **NULL** on error (if hostname is too long).

### LIBRARY

BSDNAME.LIB

## sock\_abort

```
void sock_abort( void * s );
```

### DESCRIPTION

Close a connection immediately. Under TCP this is done by sending a RST (reset). Under UDP there is no difference between `sock_close()` and `sock_abort()`.

### PARAMETERS

**s**                    Pointer to a socket.

### RETURN VALUE

None.

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

sock\_close

## sock\_bytesready

```
int sock_bytesready( void * s );
```

### DESCRIPTION

If the socket is in binary mode, **sock\_bytesready** returns the number of bytes waiting to be read. If there are no bytes waiting, it returns -1.

In ASCII mode, **sock\_bytesready** returns -1 if there are no bytes waiting to be read or the line that is waiting is incomplete (no line terminating character has been read.) The number of bytes waiting to be read will be returned given one of the following conditions:

- the buffer is full
- the socket has been closed (no line terminating character can be sent,)
- a complete line is waiting

In ASCII mode, a blank line will be read as a complete line with length 0, which will be the value returned. **sock\_bytesready** handles ASCII mode sockets better than **sock\_dataready**, since it can distinguish between an empty line on the socket and an empty buffer.

### PARAMETERS

**s**                      Pointer to a socket.

### RETURN VALUE

- 1: No bytes waiting to be read
- 0: In ASCII mode and a blank line is waiting to be read
- >0: The number of bytes waiting to be read

### LIBRARY

DCRTPC.LIB

### SEE ALSO

sock\_wait\_established, sock\_established, sockstate

## sock\_close

```
void sock_close( void * s );
```

### DESCRIPTION

Attempts to close a socket; no more data may be sent or received through that socket.

In the case of UDP, the socket is closed immediately since UDP is a connectionless protocol. TCP, however, is a connection-oriented protocol so the close must be negotiated with the remote computer. Use **sock\_wait\_closed** or wait for **tcp\_tick()** to return 0 when passed the socket to ensure that a TCP connection is closed.

In emergency cases, it is possible to abort the TCP connection rather than close it. Although not recommended for normal transactions, this service is available and is used by all TCP/IP systems.

### PARAMETERS

**s**                    Pointer to a socket.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

sock\_abort, sock\_tick, sock\_wait\_closed

## sock\_dataready

```
int sock_dataready( void *s );
```

### DESCRIPTION

Gets the number of bytes waiting to be read. If in ASCII mode, it returns zero if a newline character has not been read or the buffer is not full. See **sock\_bytesready()** for a more general function.

### PARAMETERS

**s**                    Pointer to a socket.

### RETURN VALUE

0: No bytes to read;  
!0: Number of bytes ready to read.

### LIBRARY

DCRTCP.LIB



## sockerr

```
char *sockerr( void * s );
```

### DESCRIPTION

Gets the last ASCII error message recorded for a particular socket. If no messages have been recorded, the returned value is **NULL**. The messages are read-only; do not modify them!

### PARAMETERS

**s**                      Pointer to a socket.

### RETURN VALUE

Pointer to last error message, or  
**NULL** pointer if there have been no error messages.

### LIBRARY

DCRTCP.LIB

### EXAMPLE

```
char *p;  
...  
sock_err:  
if ( status == 1)  
    puts("Closed normally");  
else if ( p = sockerr( s ))  
    printf("Socket closed with error '%s'\n\r", p );
```

## sock\_established

```
int sock_established( void *s );
```

### DESCRIPTION

TCP connections require a handshaked open to ensure that both sides recognize a connection. Whether the connection was initiated with `tcp_open()` or `tcp_listen()`, `sock_establish()` will continue to return 0 until the connection is established, at which time it will return 1. It is not enough to spin on this after a listen because it is possible for the socket to be opened, written to and closed between two checks.

`sock_bytesready()` can be called with `sock_established()` to handle this case.

UDP is a connectionless protocol, hence `sock_established()` always returns 1 for UDP sockets.

### PARAMETERS

**s**                    Pointer to a socket.

### RETURN VALUE

0: Not established;

1: Established.

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

`sock_wait_established`, `sock_bytesready`, `sockstate`

## sock\_fastread

```
int sock_fastread( void *s, byte *dp, int len );
```

### DESCRIPTION

**sock\_fastread()** attempts to read data from a socket. If possible, the buffer, **dp**, is filled, otherwise, only the number of bytes read is returned. A return value of -1 indicates a socket error.

This function cannot be used on UDP sockets after **sock\_recv\_init()** is called.

For a sample program, see Example of four input functions listed under **sock\_read()**.

### PARAMETERS

|            |   |
|------------|---|
| <b>s</b>   | Pointer to a socket.                            |
| <b>dp</b>  | Buffer to put bytes that are read.              |
| <b>len</b> | Maximum number of bytes to write to the buffer. |

### RETURN VALUE

Number of bytes read or **-1** if there was an error.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

sock\_read, sock\_fastwrite, sock\_write

## sock\_fastwrite

```
int sock_fastwrite( void *s, byte *dp, int len );
```

### DESCRIPTION

Writes up to **len** bytes from **dp** on socket **s**. This function writes as many bytes as possible to the socket and returns that number of bytes.

For UDP, **sock\_fastwrite()** will send one record if **len**  $\leq$  **ETH\_MTU - 20 - 8**

**ETH\_MTU** is the Ethernet Maximum Transmission Unit; 20 is the IP header size and 8 is the UDP header size. By default, this is 572 bytes. If **len** is greater than this number, then the function does not send the data and returns -1. Otherwise, the UDP datagram would need to be fragmented.

For TCP, the new data is queued for sending and **sock\_fastwrite()** returns the number of bytes that will be sent. The data may be transmitted immediately if enough data is in the buffer, or sufficient time has expired, or the user has explicitly used **sock\_flushnext()** to indicate this data should be flushed immediately. In either case, no guarantee of acceptance at the other end is possible.

### PARAMETERS

|            |   |
|------------|---|
| <b>s</b>   | Pointer to a socket.                            |
| <b>dp</b>  | Buffer to be written.                           |
| <b>len</b> | Maximum number of bytes to write to the socket. |

### RETURN VALUE

Number of bytes written, or  
-1 if there was an error.

### LIBRARY

DCRTPC.LIB

## sock\_flush

```
void sock_flush( void *s );
```

### DESCRIPTION

**sock\_flush()** will flush the unwritten portion of the TCP buffer to the network. No guarantee is given that the data was actually delivered. In the case of a UDP socket, no action is taken.

**sock\_flushnext()** is recommended over **sock\_flush()**.

### PARAMETERS

**s**                      Pointer to a socket.

### RETURN VALUE

None.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

`sock_flushnext`, `sock_fastwrite`, `sock_write`, `sockerr`

## sock\_flushnext

```
void sock_flushnext( void *s );
```

### DESCRIPTION

Writing to TCP sockets does not guarantee that the data are actually transmitted or that the remote computer will pass that data to the other client in a timely fashion. Using a flush function will guarantee that **DCRTPC.LIB** places the data onto the network. No guarantee is made that the remote client will receive that data.

**sock\_flushnext()** is the most efficient of the flush functions. It causes the next function that sends data to the socket to flush, meaning the data will be transmitted immediately.

Several functions imply a flush and do not require an additional flush: **sock\_puts()**, and sometimes **sock\_putc()** (when passed a `\n`).

### PARAMETERS

**s**                    Pointer to a socket.

### RETURN VALUE

None.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

`sock_write`, `sock_fastread`, `sock_read`, `sockerr`,  
`sock_wait_input`, `sock_flush`, `sock_flushnext`

## sock\_getc

```
int sock_getc( void *s );
```

### DESCRIPTION

Gets the next character from the socket. NOTE: This function blocks.

This function cannot be used on UDP sockets after **sock\_recv\_init()** is called.

For a sample program, see Example of four input functions listed under **sock\_read()**.

### PARAMETERS

**s**                      Pointer to a socket.

### RETURN VALUE

Character read or **-1** if error.

### LIBRARY

DCRTCP.LIB

## sock\_gets

```
int sock_gets( void *s, char *text, int len );
```

### DESCRIPTION

Reads a string from a socket and replaces the CR or LF with a '\0'. If the string is longer than **len**, the string is null terminated and the remaining characters in the string are discarded.

To use **sock\_gets()**, you must first set ASCII mode using **sock\_mode()**.

This function cannot be used on UDP sockets after **sock\_recv\_init()** is called.

For a sample program, see Example of four input functions listed under **sock\_read()**.

### PARAMETERS

|             |                           |
|-------------|---------------------------|
| <b>s</b>    | Pointer to a socket       |
| <b>text</b> | Buffer to put the string. |
| <b>len</b>  | Max length of 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.

### LIBRARY

DCR`TCP`.LIB

### SEE ALSO

sock\_puts, sock\_putc, sock\_getc, sock\_read, sock\_write

## sock\_init

```
void sock_init(void);
```

### DESCRIPTION

This function initializes the packet driver and **DCR`TCP`.LIB** using the compiler defaults for configuration. This function must be called before using other **DCR`TCP`.LIB** functions.

### LIBRARY

DCR`TCP`.LIB



## sock\_mode

```
void sock_mode( void *s, word mode );
```

### DESCRIPTION

This function changes some of the basic handling of a socket. The following macros can be passed as the 2nd parameter (OR'd together if necessary):

#### TCP\_MODE\_ASCII | TCP\_MODE\_BINARY

TCP and UDP sockets are usually in binary mode which means an arbitrary stream of bytes is allowed (TCP is treated as a byte stream and UDP is treated as records filled with bytes.) The default is **TCP\_MODE\_BINARY**. By changing the mode to **TCP\_MODE\_ASCII**, some of the **DCRTCP.LIB** functions will see a stream of records terminated with a newline character.

In ASCII mode, **sock\_bytesready()** will return -1 until a newline-terminated string is in the buffer or the buffer is full. **sock\_puts()** will append a newline to any output. **sock\_gets()** (which should only be used in ASCII mode) removes the newline and null terminates the string.

For a sample program, see Example of four input functions listed under **sock\_read()**.

#### TCP\_MODE\_NAGLE | TCP\_MODE\_NONAGLE

The Nagle algorithm may substantially reduce network traffic with little negative effect on a user (In some situations, the Nagle algorithm even improves application performance.) The default is **TCP\_MODE\_NAGLE**. This mode only affects TCP connections. If you are doing X-Windows or real time data collection, you may switch the Nagle algorithm off by selecting the **TCP\_MODE\_NONAGLE** flag.

#### UDP\_MODE\_CHK | UDP\_MODE\_NOCHK

Checksums are required for TCP, but not for UDP. The default is **UDP\_MODE\_CHK**. If you are providing a checksum at a higher level, the low level checksum may be redundant. The checksum for UDP can be disabled by selecting the **UDP\_MODE\_NOCHK** flag. Note that you do not control whether the remote computer will send checksums. If that computer does checksum its outbound data, **DCRTCP.LIB** will check the received packet's checksum.

### PARAMETERS

|             |                                |
|-------------|--------------------------------|
| <b>s</b>    | Pointer to a socket.           |
| <b>mode</b> | New mode for specified socket. |

### LIBRARY

DCRTCP.LIB

## sock\_preread

```
int sock_preread( void *s, byte *dp, int len );
```

### DESCRIPTION

This function reads up to **len** bytes from the socket into the buffer **dp**. The bytes are not removed from the socket's buffer.

### PARAMETERS

|            |                                     |
|------------|-------------------------------------|
| <b>s</b>   | Pointer to a socket.                |
| <b>dp</b>  | Buffer to preread into.             |
| <b>len</b> | Maximum number of bytes to preread. |

### RETURN VALUE

**0**: No data waiting;  
**-1**: Error;  
**>0**: Number of preread bytes.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

sock\_fastread, sock\_fastwrite, sock\_read, sock\_write



## sock\_puts

```
int sock_puts( void *s, byte *dp );
```

### DESCRIPTION

A string is placed on the output buffer and flushed as described under **sock\_flushnext()**. If the socket is in ASCII mode, CR and LF are appended to the string. No other ASCII character expansion is performed. Note that **sock\_puts()** uses **sock\_write()**, and thus may block if the output buffer is full. See **sock\_write()** for more details.

### PARAMETERS

|           |                                 |
|-----------|---------------------------------|
| <b>s</b>  | Pointer to a socket.            |
| <b>dp</b> | Buffer to read the string from. |

### RETURN VALUE

Length of string in buffer.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

sock\_gets, sock\_putc, sock\_getc, sock\_read, sock\_write

## sock\_rleft

```
int sock_rleft( void *s );
```

### DESCRIPTION

Determines the number of bytes available in the receive buffer.

### PARAMETERS

|          |                      |
|----------|----------------------|
| <b>s</b> | Pointer to a socket. |
|----------|----------------------|

### RETURN VALUE

Number of bytes available in the receive buffer.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

sock\_rbsize, sock\_rused, sock\_tbsize, sock\_tused,  
sock\_tleft

## sock\_rbsize

```
int sock_rbsize( void *s );
```

### DESCRIPTION

Determines the size of the receive buffer for the specified socket.

### PARAMETERS

**s**                      Pointer to a socket.

### RETURN VALUE

The size of the receive buffer.

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

sock\_rbleft, sock\_rbused, sock\_tbsize, sock\_tbused,  
sock\_tbleft

## sock\_rbused

```
int sock_rbused( void *s );
```

### DESCRIPTION

Gets the number of bytes in use in the receive buffer for the specified socket.

### PARAMETERS

**s**                      Pointer to a socket.

### RETURN VALUE

Number of bytes in use.

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

sock\_rbleft, sock\_tbsize, sock\_tbused, sock\_tbleft

```
int sock_read( void *s, byte *dp, int len );
```

#### DESCRIPTION

**sock\_read()** will busywait until **len** bytes are read or a socket error exists. If **sock\_yield()** has been called, the user-defined function that is passed to it will be called in a tight loop while **sock\_read()** is busywaiting.

This function cannot be used or

## EXAMPLE OF FOUR INPUT FUNCTIONS

The following program shows how the four main input functions may be used to read a text stream. Note that `sock_fastread()` and `sock_read()` do not necessarily return a complete or single line, they return blocks of bytes. In comparison, `sock_getc()` returns a single byte at a time and yields poor performance.

```
/*
 * This is a sample FINGER program which compares sock_fastread(),
 * sock_read(), sock_gets(), and sock_getc() for handling ASCII
 * data.
 *
 * Note that sock_fastread(), sock_read(), and sock_getc()
 * do NOT return single line strings, they return ordered bytes.
 * sock_getc() looks the simplest, but it has the highest overhead
 * both in terms of DCR_TCP, and especially in terms of the output
 * through putch().
 *
 * FINGER [user]@host mode where mode is 0, 1, 2 or 3 to indicate
 * using sock_fastread(), sock_read(), sock_getc() or sock_gets().
 * All modes returned identical output to the screen.
 */
```

```

#define MY_IP_ADDRESS  "10.10.6.100"
#define MY_NETMASK    "255.255.255.0"
#define MEMMAP        xmem

#include "dcrtcp.lib"

#define FINGER_PORT 79

finger(char* userid, char* host, longword hoststring, int method) {
    tcp_socket fingersock;
    tcp_socket *s;
    char buffer[ 513 ]; /* space for 512 plus zero terminator */
    int status;
    int len;

    s = &fingersock;
    if (!tcp_open( s, 0, host, FINGER_PORT, NULL )) {
        puts("Sorry, unable to connect to that machine right now!");
        return;
    }
    printf("waiting...\r");
    sock_wait_established(s, sock_delay , NULL, &status);

    if (*userid)
        printf("'s' is looking for '%s'...\n\r\n\r\n", hoststring, userid);
    strcpy( buffer, userid );
    rip( buffer ); /* kill all \n and \r's */
    strcat( buffer , "\n");

    sock_puts( s, buffer );

    switch (method) {

```

```

/*****
*          using sock_fastread()          *
*****/

```

```

case 0 :
    while ( 1 ) {
        sock_wait_input( s, 30, NULL, &status );

        len = sock_fastread( s, buffer, 512 );
        buffer[ len ] = 0; /* must terminate it */
        printf( "%s", buffer );
    }
    break;

```



```
/******  
*          using sock_read()          *  
******/
```

```
case 1 :  
    while( 1 ) {  
        sock_wait_input( s, 30, NULL, &status );  
        len = sock_dataready( s );  
        if ( len > sizeof( buffer ))  
            len = sizeof( buffer );  
  
        sock_read( s, buffer, len );  
        buffer[ len ] = 0;  
        printf( "%s", buffer );  
    }  
    break;
```

```
/******  
* using sock_getc()          *  
******/
```

```
case 2 :  
    while ( 1 ) {  
        sock_wait_input( s, 30, NULL, &status );  
        putchar( sock_getc( s ) );  
    }  
    break;
```

```
/*.....  
* using sock_gets() *  
/*...../
```

```
case 3 :  
    sock_mode( s, TCP_MODE_ASCII );  
    while ( 1 ) {  
        sock_wait_input( s, 30, NULL, &status );  
        len = sock_gets( s, buffer, 512 );  
        puts( buffer );  
    }  
    break;  
}  
sock_err:  
    switch (status) {  
        case 1 : /* foreign host closed */  
            break;  
        case -1: /* timeout */  
            printf("\n\rConnection timed out!");  
            break;  
    }  
    sock_close( s );  
    printf("\n\r");  
}
```

```

char *meth[]={ "sock_fastread", "sock_read", "sock_getc",
"sock_gets" };

main() {
    char *user,*server;
    longword host;
    int status;
    word method;

    sock_init();

    strcpy(user,"root");
    strcpy(user,"foo.bar");
    method=0; /* sock_fastread */

    if ( method > 3 ) {
        puts("only values 0 through 3 are valid");
        exit( 2 );
    }
    printf("Using method %s\n\r", meth[ method ]);
    if ( host = resolve( server )) {
        status = finger( user, host, server, method);
    } else {
        printf("Could not resolve host '%s'\n\r", server );
        exit( 3 );
    }
    exit( status );
}

```

## sock\_recv

```
int sock_recv( sock_type *s, char *buffer, int len );
```

### DESCRIPTION

After a UDP socket is initialized with `udp_open()` and `sock_recv_init()`, `sock_recv` scans the buffers for any datagram received by that socket.

### PARAMETERS

|                        |                          |
|------------------------|--------------------------|
| <code>s</code>         | Pointer to a UDP socket. |
| <code>buffer</code>    | Buffer to put datagram.  |
| <code>maxlength</code> | Length of buffer.        |

### RETURN VALUE

Length of datagram;  
0 if no datagram found;  
-1 if receive buffer not initialized with `sock_recv_init()`.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

`sock_recv_from`, `sock_recv_init`

## EXAMPLE USING SOCK\_RECV()

```
#define MY_IP_ADDRESS "10.10.6.100"
#define MY_NETMASK "255.255.255.0"
#include <xmem>

#include <dcrtcp.lib>

#define SAMPLE 401

udp_Socket data;
char bigbuf[ 8192 ];

main() {
    word templen;
    char spare[ 1500 ];

    sock_init();
    if ( !udp_open( &data, SAMPLE, 0xffffffff, SAMPLE, NULL) ) {
        puts("Could not open broadcast socket");
        exit( 3 );
    }

    /* set large buffer mode */
    if ( sock_recv_init( &data, bigbuf, sizeof( bigbuf ) ) ) {
        puts("Could not enable large buffers");
        exit( 3 );
    }

    sock_mode( &data, UDP_MODE_NOCHK );    /* turn off checksums */

    while (1) {
        tcp_tick( NULL );

        if ( templen = sock_recv( &data, spare, sizeof( spare ) ) ) {
            /* something received */
            printf("Got %u byte packet\n", templen );
        }
    }
}
```

## sock\_recv\_from

```
int sock_recv_from( sock_type *s, long *hisip, word *hisport,  
    char *buffer, int len);
```

### DESCRIPTION

After a UDP socket is initialized with `udp_open()` and `sock_recv_init()`, `sock_recv_from()` scans the buffers for any datagram received by that socket and identifies the remote host's address.

### PARAMETERS

|                      |   |
|----------------------|---|
| <code>s</code>       | Pointer to UDP socket.                      |
| <code>hisip</code>   | IP of remote host, according to UDP header. |
| <code>hisport</code> | Port of remote host.                        |
| <code>buffer</code>  | Buffer to put datagram in.                  |
| <code>len</code>     | Length of buffer.                           |

### RETURN VALUE

`>0`: Length of datagram received;  
`0`: No datagram;  
`-1`: Receive buffer was not initialized with `sock_recv_init()`.

### LIBRARY

DCRRTCP.LIB

### SEE ALSO

`sock_recv`, `sock_recv_init`

## sock\_recv\_init

```
int sock_recv_init( sock_type *s, void *space, word len);
```

### DESCRIPTION

The basic socket reading functions (`sock_read()`, `sock_fastread()`, etc.) are not adequate for all your UDP needs. The most basic limitation is their inability to treat UDP as a record service.

A record service must receive distinct datagrams and pass them to the user program as such. You must know the length of the received datagram and the sender (if you opened in broadcast mode). You may also receive the datagrams very quickly, so you must have a mechanism to buffer them.

Once a socket is opened with `udp_open()`, you can use `sock_recv_init()` to initialize that socket for `sock_recv()` and `sock_recv_from()`. Note that `sock_recv()` and related functions are *incompatible* with `sock_read()`, `sock_fastread()`, `sock_gets()` and `sock_getc()`. Once you have used `sock_recv_init()`, you can no longer use the older-style calls.

`sock_recv_init()` installs a large buffer area which gets segmented into smaller buffers. Whenever a UDP datagram arrives, `DCRTCP.LIB` stuffs that datagram into one of these new buffers. The new functions scan those buffers. You must select the size of the buffer you submit to `sock_recv_init()`; make it as large as possible, say 4K, 8K or 16K.

For a sample program, see Example using `sock_recv()` listed under `sock_recv()`.

### PARAMETERS

|              |  |
|--------------|--|
| <b>s</b>     | Pointer to a UDP socket.   |
| <b>space</b> | Buffer of temporary storage space to store newly received packets. |
| <b>len</b>   | Size of the buffer.  |

### RETURN VALUE

0.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

`sock_recv_from`, `sock_recv`

## sockstate

```
char *sockstate( void * s );
```

### DESCRIPTION

Returns a string that gives the current state for a socket.

### PARAMETERS

**s**                    Pointer to a socket.

### RETURN VALUE

An ASCII message which represents the current state of the socket. These strings should not be modified.

"**Listen**" indicates a passively opened socket that is waiting for a connection.

"**SynSent**" and "**SynRcvd**" are connection phase intermediate states.

"**Established**" states that the connection is complete.

"**EstClosing**" "**FinWait1**" "**FinWait2**" "**CloseWait**" "**Closing**"

"**LastAck**" "**TimeWait**" and "**CloseMSL**" are connection termination intermediate states.

"**Closed**" indicates that the connection is completely closed.

"**UDP socket**" is always returned for UDP sockets because they are stateless.

"**Not an active socket**" is a default value used when the socket is not recognized as UDP or TCP.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

sock\_established, sock\_dataready

```
char *p;  
...  
#ifdef DEBUG  
if ( p = sockstate( s ) )  
    printf("Socket state is '%s'\n\r", p );  
#endif DEBUG
```



## sock\_tleft

```
int sock_tleft( void *s );
```

### DESCRIPTION

Gets the number of bytes left in the transmit buffer. If you do not wish to block, you may first query how much space is available for writing by calling this function before generating data that must be transmitted. This removes the need for your application to also buffer data.

### PARAMETERS

**s**                    Pointer to a socket.

### RETURN VALUE

Number of bytes left in the transmit buffer.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

sock\_rbsize, sock\_rbused, sock\_rbleft, sock\_tbsize,  
sock\_tbused

```
if ( sock_tleft( s ) > 10 ) {  
    /* we can send up to 10 bytes without blocking or overflowing */  
    ....  
}
```

## sock\_tbsize

```
int sock_tbsize( void *s );
```

### DESCRIPTION

Determines the size of the transmit buffer for the specified socket.

### PARAMETERS

**s**                    Pointer to a socket.

### RETURN VALUE

The size of the transmit buffer.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

sock\_rbsize, sock\_rbused, sock\_rbleft, sock\_tbleft,  
sock\_tbused

## sock\_tbused

```
int sock_tbused( void *s );
```

### DESCRIPTION

Gets the number of bytes in use in the transmit buffer for the specified socket.

### PARAMETERS

**s**                    Pointer to a socket.

### RETURN VALUE

Number of bytes in use.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

sock\_rbsize, sock\_rbused, sock\_rbleft, sock\_tbsize,  
sock\_tbleft

## sock\_tick

```
sock_tick( void * s, int * optional_status_ptr );
```

### DESCRIPTION

This macro calls `tcp_tick()` to quickly check incoming and outgoing data and to manage all the open sockets. If our particular socket, `s`, is either closed or made inoperative due to an error condition, `sock_tick()` sets the value of `*optional_status_ptr` (if the pointer is not `NULL`) to 1, then jumps to a local, user-supplied label, `sock_err`. If the socket connection is fine and the pointer is not `NULL` `*optional_status_ptr` is set to 0.

### PARAMETERS

|                                  |                         |
|----------------------------------|-------------------------|
| <code>s</code>                   | Pointer to a socket.    |
| <code>optional_status_ptr</code> | Pointer to status word. |

### RETURN VALUE

None.

### LIBRARY

DCRTCP.LIB

## sock\_wait\_closed

```
void sock_wait_closed(void * s, int seconds, int (*fptr)(), int*
    status);
```

### DESCRIPTION

This macro waits until a TCP connection is fully closed. Returns immediately for UDP sockets. On an error, the macro jumps to a local, user-supplied **sock\_err** label. If **fptr** returns !0 the macro returns with the status word set to the value of **fptr**'s return value.

### PARAMETERS

|                |   |
|----------------|---|
| <b>s</b>       | Pointer to a socket.  |
| <b>seconds</b> | Number of seconds to wait before timing out. A value of zero indicates the macro should never time-out. A good value to use is <b>sock_delay</b> , a system variable set on configuration. Typically <b>sock_delay</b> is about 20 seconds, but can be set to something else in <b>main()</b> . |
| <b>fptr</b>    | Function to call repeatedly while waiting. This is a user-supplied function.  |
| <b>status</b>  | Pointer to a status word.   |

### RETURN VALUE

None.

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

sock\_wait\_established, sock\_wait\_input

## sock\_wait\_established

```
void sock_wait_established(void* s, int seconds, int (*fptr)(),
    int* status);
```

### DESCRIPTION

This macro waits until a connection is established for the specified TCP socket, or aborts if a time-out occurs. It returns immediately for UDP sockets. On an error, the macro jumps to the local, user-supplied **sock\_err** label. If **fptr**, a user-supplied function, returns zero the macro returns.

### PARAMETERS

|                |   |
|----------------|---|
| <b>s</b>       | Pointer to a socket.  |
| <b>seconds</b> | Number of seconds to wait before timing out. A value of zero indicates the macro should never time-out. A good value to use is <b>sock_delay</b> , a system variable set on configuration. Typically <b>sock_delay</b> is about 20 seconds, but can be set to something else in <b>main()</b> . |
| <b>fptr</b>    | Function to call repeatedly while waiting. This is a user-supplied function.  |
| <b>status</b>  | Pointer to a status word.   |

### RETURN VALUE

None.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

`sock_wait_input`, `sock_wait_closed`

## sock\_wait\_input

```
void sock_wait_input(void* s, int seconds, int (*fptr)(), int*
    status);
```

### DESCRIPTION

Waits until input exists for functions such as `sock_read()` and `sock_gets()`. As described under `sock_mode()`, if in ASCII mode, `sock_wait_input` only returns when a complete string exists or the buffer is full.

Under some conditions, (e.g., the remote or local host closes or resets the connection) this macro jumps to a local, user-supplied `sock_err` label.

For sample programs, see the examples listed under `tcp_open()`, `tcp_listen()`, and `sock_read()`.

### PARAMETERS

|                |   |
|----------------|---|
| <b>s</b>       | Pointer to a socket.  |
| <b>seconds</b> | Number of seconds to wait before timing out. A value of zero indicates the macro should never time-out. A good value to use is <code>sock_delay</code> , a system variable set on configuration. Typically <code>sock_delay</code> is about 20 seconds, but can be set to something else in <code>main()</code> .   |
| <b>fptr</b>    | Function to call repeatedly while waiting.  |
| <b>status</b>  | A pointer to the status word. If this parameter is <code>NULL</code> , no status number will be available, but the macro will otherwise function normally. Typically the pointer will point to a local signed integer that is used only for status. <code>status</code> may be tested to determine how the socket was ended. A value of 1 means a proper close while a -1 value indicates a reset or abort. |

### RETURN VALUE

None.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

`sock_wait_established`, `sock_wait_closed`, `sock_mode`

## sock\_write

```
int sock_write( void *s, byte *dp, int len );
```

### DESCRIPTION

Writes up to **len** bytes from **dp** on socket **s**. This function busywaits until either the buffer is completely written or a socket error occurs. If **sock\_yield()** has been called, the user-defined function that is passed to it will be called in a tight loop while **sock\_write()** is busywaiting.

For UDP, **sock\_write()** will send one (or more) records. For TCP, the new data may be transmitted if enough data is in the buffer or sufficient time has expired or the user has explicitly used **sock\_flushnext()** to indicate this data should be flushed immediately. In either case, there is no guarantee of acceptance at the other end.

### PARAMETERS

|            |   |
|------------|---|
| <b>s</b>   | Pointer to a socket                             |
| <b>dp</b>  | Pointer to a buffer.                            |
| <b>len</b> | Maximum number of bytes to write to the buffer. |

### RETURN VALUE

Number of bytes written or **-1** on an error.

### LIBRARY

DCRTPC.LIB

### SEE ALSO

`sock_read`, `sock_fastwrite`, `sock_fastread`, `sockerr`,  
`sock_wait_input`, `sock_flush`, `sock_flushnext`

## sock\_yield

```
int sock_yield( tcp_Socket *s, void (*fn)());
```

### DESCRIPTION

This function, if called prior to one of the blocking functions, will cause **fn**, the user-defined function that is passed in as the second parameter, to be called repeatedly while the blocking function is in a busywait state.

### PARAMETERS

|           |                          |
|-----------|--------------------------|
| <b>s</b>  | Pointer to a TCP socket. |
| <b>fn</b> | User-defined function.   |

### RETURN VALUE

0

### LIBRARY

DCR\_TCP.LIB

## tcp\_clearreserve

```
void tcp_clearreserve( word port );
```

### DESCRIPTION

This function causes the **DCR\_TCP.LIB** stack to handle a socket connection to the specified port normally. This undoes the action taken by **tcp\_reserveport()**.

### PARAMETERS

|             |              |
|-------------|--------------|
| <b>port</b> | Port to use. |
|-------------|--------------|

### RETURN VALUE

None.

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

tcp\_reserveport



## tcp\_config

```
void tcp_config(char *name, char *value);
```

### DESCRIPTION

Sets TCP/IP stack parameters at runtime. It should not be called with open sockets.

Additionally, **MY\_IP\_ADDRESS** can be overridden by **sethostid()**, and **MY\_HOSTNAME** can be overridden by **sethostname()**.

### PARAMETERS

|              |   |
|--------------|---|
| <b>name</b>  | Setting to be changed. The possible parameters are:<br><b>MY_IP_ADDRESS</b> : host IP address (use <b>sethostid()</b> instead)<br><b>MY_NETMASK</b><br><b>MY_GATEWAY</b> : host's default gateway<br><b>MY_NAMESERVER</b> : host's default nameserver<br><b>MY_HOSTNAME</b><br><b>MY_DOMAINNAME</b> : host's domain name (use <b>setdomainname()</b> instead)<br><b>MTU</b> : maximum size of packets |
| <b>value</b> | The value to assign to <b>name</b> .  |

### RETURN VALUE

None

### LIBRARY

DCR\_TCP.LIB

## tcp\_keepalive

```
int tcp_keepalive(tcp_Socket *s, long timeout);
```

### DESCRIPTION

Enable or disable TCP keepalives on a specified socket. The socket must already be open. Keepalives will then be sent after "timeout" seconds of inactivity. It is highly recommended to keep timeout as long as possible, to reduce the load on the network. Ideally, the timeout should be no shorter than 2 hours. After the timeout is sent, and **KEEPALIVE\_WAITTIME** seconds pass, another keepalive will be sent, in case the first was lost. This will be retried **KEEPALIVE\_NUMRETRY**s times. Both of these macros can be #defined at the top of your program, overriding the defaults of 60 seconds, and 4 retries.

Using keepalives is not a recommended procedure. Ideally, the application using the socket should send its own keepalives. **tcp\_keepalive()** is provided because telnet and a few other network protocols do not have a method of sending keepalives at the application level.

### PARAMETERS

|                |   |
|----------------|---|
| <b>s</b>       | Pointer to a socket.  |
| <b>timeout</b> | Period of inactivity, in seconds, before sending a keepalive or 0 to turn off keepalives. |

### RETURN VALUE

0: Success;  
1: Error

### LIBRARY

DCR\_TCP.LIB

### SEE ALSO

sock\_fastread, sock\_fastwrite, sock\_write, sockerr, sock\_wait\_input

## tcp\_listen

```
int tcp_listen( tcp_socket *s, word lport, longword ina, word
               port, int (*signal_handler), word timeout );
```

### DESCRIPTION

This function tells **DCRTCP.LIB** that an incoming session for a particular port will be accepted. After a call to **tcp\_listen()**, the function **sock\_established()** (or the macro **sock\_wait\_established**) must be called to poll the connection until a session is fully established.

It is possible for a connection to be opened, written to and closed between two calls to the function **sock\_established()**. To handle this case, call **sock\_bytesready()** or **sock\_dataready()** to determine if there is data to be read from the buffer.

Multiple calls to **tcp\_listen()** to the same local port (**lport**) are acceptable and constitute the **DCRTCP.LIB** mechanism for supporting multiple incoming connections to the same local port. Each time another host attempts to open a session on that particular port, another one of the listens will be consumed until such time as all listens have become established sessions and subsequent remote host attempts will receive a reset.

### PARAMETERS

|                       |  |
|-----------------------|--|
| <b>s</b>              | Pointer to a socket.   |
| <b>lport</b>          | Port to listen on (the local port number).   |
| <b>ina</b>            | IP address of the remote host to accept connections from or 0 for all.   |
| <b>port</b>           | Port to accept connections from or 0 for all.  |
| <b>signal_handler</b> | This function is called if the connection is either closed or reset. The parameter for <b>signal_handler</b> is the pointer to a function which will be called when the socket is either closed or reset. <i>Some details for implementation of this service have not been finalized, and it is recommended the user insert a value of <b>NULL</b> for the present time.</i> |
| <b>timeout</b>        | Number of seconds to wait before timing out in <b>sock_wait_established</b> . Set to zero for no time-out.   |

### RETURN VALUE

0: Error;  
1: Success.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

tcp\_open

## EXAMPLE USING TCP\_LISTEN()

```
#define MY_IP_ADDRESS "10.10.6.100"
#define MY_NETMASK "255.255.255.0"
#memmap xmem
#use "dcrtcp.lib"

#define TELNET_PORT 23

static tcp_Socket *s;
char *userid;

telnets(int port) {
    tcp_Socket telnetsock;
    char buffer[ 512 ];
    int status;
    int len;
    s = &telnetsock;
    tcp_listen( s, port, 0L, 0, NULL, 0);

    sock_wait_established(s, 0, NULL, &status);

    puts("Receiving incoming connection");
    sock_mode( s, TCP_MODE_ASCII );
    sock_puts( s, "Welcome to a sample telnet server.");
    sock_puts( s, "Each line you type will be printed on this\"
        "screen once you hit return.");
    /* other guy closes connection except if we timeout .... */
    while ( 1 ) {
        sock_wait_input( s , 0, NULL, &status);
        sock_gets( s, buffer, 512 );
        puts( buffer );
    }
    sock_err:
    switch (status) {
        case 1 : /* foreign host closed */
            puts("User closed session");
            return;
        case -1: /* timeout */
            printf("\n\rConnection timed out!");
            return;
    }
}

main() {
    sock_init();
    telnets( TELNET_PORT);
    exit( 0 );
}
```

## tcp\_open

```
int tcp_open( void *s, word lport, longword ina, word port,
              int (*signal_handler)());
```

### DESCRIPTION

This function actively creates a session with another machine. After a call to `tcp_open()`, the function `sock_established()` (or the macro `sock_wait_established`) must be called to poll the connection until a session is fully established.

It is possible for a connection to be opened, written to and closed between two calls to the function `sock_established()`. To handle this case, call `sock_bytesready()` or `sock_dataready()` to determine if there is data to be read from the buffer.

### PARAMETERS

|                       |   |
|-----------------------|---|
| <b>s</b>              | Pointer to a socket.  |
| <b>lport</b>          | Our port, zero for the next available 1025-65536. A few applications will require you to use a particular local port number, but most network applications let you use almost any port with a certain set of restrictions. For example, <b>FINGER</b> or <b>TELNET</b> clients can use any local port value, so pass the value of zero for <b>lport</b> and let <b>DCRTCP.LIB</b> pick one for you. |
| <b>ina</b>            | IP address to connect to.   |
| <b>port</b>           | Port to connect to.   |
| <b>signal_handler</b> | This function is called if the connection is either closed or reset. The parameter for <b>signal_handler</b> is the pointer to a function which will be called when the socket is either closed or reset. <i>Some details for implementation of this service have not been finalized, and it is recommended the user insert a value of <b>NULL</b> for the present time.</i>                        |

### RETURN VALUE

0: Unable to resolve the remote computer's hardware address;  
!0 otherwise.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

tcp\_listen

## EXAMPLE USING TCP\_OPEN()

```
#define MY_IP_ADDRESS "10.10.6.100"
#define MY_NETMASK "255.255.255.0"
#define MEMMAP xmem

#include "dcrtcp.lib"

#define ADDRESS "10.10.6.19"
#define PORT "200"

main() {
    word status;
    word port;
    longword host;
    tcp_socket_t tsock;

    sock_init();

    if (!(host = resolve(ADDRESS))) {
        puts("Could not resolve host");
        exit( 3 );
    }
    port = atoi( PORT );
    printf("Attempting to open '%s' on port %u\n\r", ADDRESS, port );
    if ( !tcp_open( &tsock, 0, host, port , NULL )) {
        puts("Unable to open TCP session");
        exit( 3 );
    }

    printf("Waiting a maximum of %u seconds for connection"\
        " to be established\n\r", sock_delay );

    sock_wait_established( &tsock, sock_delay, NULL, &status );
    puts("Socket is established");
    sock_close( &tsock );
    sock_wait_closed( &tsock, sock_delay, NULL, &status );

    sock_err:
    switch ( status ) {
        case 1 :
            puts("Connection closed normally");
            break;
        case 2 :
            puts("Problem occurred...");
            sockerr( &tsock );
            break;
    }
    exit( (status == 1 ) ? 0 : 1 );
}
```

```
/* the following are the results from running 'test sunee 25'  
  
Attempting to open 'sunee' on port 25  
Waiting a maximum of 10 seconds for connection to be established  
Socket is established  
Connection closed normally  
*/
```

## **tcp\_reserveport**

```
void tcp_reserveport( word port );
```

### **DESCRIPTION**

This function allows a connection to be established even if there is not yet a socket available. This is done by setting a parameter in the TCP header during the connection setup phase that indicates 0 bytes of data can be received at the present time. The requesting end of the connection will wait until the TCP header parameter indicates that data will be accepted.

The 2MSL waiting period for closing a socket is avoided by using this function.

The penalty of slower connection times on a controller that is processing a large number of connections is offset by allowing the program to have less sockets and consequently less RAM usage.

### **PARAMETERS**

**port**                   Port to use.

### **RETURN VALUE**

None.

### **LIBRARY**

DCRTCP.LIB

### **SEE ALSO**

tcp\_clearreserve

## tcp\_tick

```
int tcp_tick( void *s );
```

### DESCRIPTION

This function is a single kernel routine designed to quickly process packets and return as soon as possible. `tcp_tick()` performs processing on all sockets upon each invocation: checking for new packets, processing those packets, and performing retransmissions on lost data. On most other computer systems and other kernels, performing these required operations in the background is often done by a task switch. `DCR_TCP.LIB` does not use a tasker for its basic operation, although it can adopt one for the user-level services.

Although you may ignore the returned value of `tcp_tick()`, it is the easiest method to determine the status of the given socket.

### PARAMETERS

**s**                      Pointer to a socket. If **NULL**, the returned value is always **0**.

### RETURN VALUE

**0**: Connection reset or closed by other host or **NULL** was passed in.  
**!0**: Connection is fine.

### LIBRARY

`DCR_TCP.LIB`

### SEE ALSO

`tcp_open`, `sock_close`, `sock_abort`, `sock_tick`,  
`sock_wait_established`



## udp\_open

```
int udp_open( udp_Socket *s, word lport, longword ina, word
              port, int (*datahandler)());
```

### DESCRIPTION

UDP sockets are used for connectionless data transfers. Despite the connectionless nature, which is protocol-dependent, **DCRTCP.LIB** imposes a socket mechanism that requires a destination address. As described under the UDP datagram service, you may elect to use datagram-oriented features.

If the remote host is set to -1, all packets received by this computer and destined for **lport** will return data with the various read statements. Write statements to this socket will cause broadcasts. This mechanism is suitable for broadcast information such as RIP packets. If the remote host is set to 0, the next packet received by **DCRTCP.LIB** destined for this machine's UDP port number, **lport**, will complete the socket.

If the remote host is set to a particular address, either host may initiate traffic. Multiple calls to **udp\_open()** with **ina** set to zero is a useful way of accepting multiple incoming sessions.

**udp\_open()** will return 0 if the socket cannot be opened. A typical reason would be that the host's physical address cannot be resolved using ARP or normal routing mechanisms. When an error occurs, you might try the **PING.EXE** application to test the accessibility of the other computer.

Although multiple calls to **udp\_open()** may normally be made with the same **lport** number, only one **udp\_open()** should be made on a particular **lport** if the **ina** is set to -1. Essentially, the broadcast and nonbroadcast protocols cannot co-exist.

### PARAMETERS

|                     |  |
|---------------------|--|
| <b>s</b>            | Pointer to a UDP socket.                     |
| <b>lport</b>        | Local port                                   |
| <b>ina</b>          | Acceptable remote IP, or -1 for broadcast.   |
| <b>port</b>         | Acceptable remote port, or -1 for broadcast. |
| <b>data_handler</b> | Function to call when data is received.      |

### RETURN VALUE

0 if destination hardware address cannot be resolved; !0 otherwise.

### LIBRARY

DCRTCP.LIB

### SEE ALSO

sock\_recv, sock\_recv\_init, sock\_recv\_from

## EXAMPLE OF USING UDP\_OPEN()

```
#define MY_IP_ADDRESS "10.10.6.100"
#define MY_NETMASK "255.255.255.0"
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/timeb.h>

#define ADDRESS "10.10.6.19"
#define PORT "200"

main() {
    word status, port;
    longword host;
    udp_Socket usock;

    sock_init();
    if (!(host = resolve( ADDRESS ))) {
        puts("Could not resolve host");
        exit( 3 );
    }
    port = atoi( PORT );
    printf("Attempting to open '%s' on port %u\n\r", ADDRESS, port);
    if ( !udp_open( &usock, 0, host, port , NULL ) ) {
        puts("Unable to open UDP session");
        exit( 3 );
    }
    /* udp, no need to wait for connection unless expecting incoming
    session. wait_sock_established would return immediately */

    puts("Socket is established");

    /* note, no data has been sent, no connection established, the
    other guy doesn't even know we are interested */

    sock_close( &usock );

sock_err:
    switch ( status ) {
        case 1 :
            puts("Connection closed normally");
            break;
        case 2 :
            puts("Problem occurred...");
            sockerr( &usock );
            break;
    }
    exit( (status == 1 ) ? 0 : 1 );
}
/* the results of running 'TEST sunee 25' are :
Attempting to open 'sunee' on port 25
Socket is established
Connection closed normally */
```

## 2.8 Macros

### **DISABLE\_DNS**

This macro disables DNS lookup. This prevents a UDP socket for DNS from being allocated, thus saving memory. Users may still call `resolve()` with an IP address.

### **MAX\_SOCKETS**

This macro defines the number of sockets that will be allocated, not including the socket used for DNS lookups. If DNS lookups are used, you must provide a value for `MAX_SOCKETS`.

### **MY\_DOMAIN**

This macro is the initial domain name.

### **MY\_GATEWAY**

### **MY\_IP\_ADDRESS**

### **MY\_NAMESERVER**

### **MY\_NETMASK**

## **SOCK\_BUF\_SIZE**

This macro determines the size of the socket buffers. A TCP socket will have two buffers of size **SOCK\_BUF\_SIZE**/2 for send and receive. A UDP socket will have a single socket of size **SOCK\_BUF\_SIZE**. Both types of sockets take the same total amount of buffer space.

## **tcp\_MaxBufSize**

This use of this macro is deprecated in Dynamic C version 6.57 and higher; it has been replaced by **SOCK\_BUF\_SIZE**. It will work slightly differently in these later versions: the buffer for the UDP socket will be **tcp\_MaxBufSize \* 2**, which is twice as large as before. This macro is being kept for backwards compatibility.

In Dynamic C versions 6.56 and earlier, this macro determines the size of the input and output buffers for TCP/IP sockets. The **sizeof(tcp\_socket)** will be about 200 bytes more than double this value. The optimum value for local Ethernet connections is greater than the MSS (1460). You may want to lower this value to reduce RAM usage.

```
#define tcp_MaxBufSize 600  
#use "dcrtcp.lib"
```

# Server Utility Library 3

The server utility library, **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.

HTML form functionality is included in **ZSERVER.LIB**.

## 3.1 Data Structures for Zserver.lib

There are several data structures in this library of interest to developers of HTTP or FTP servers.

### 3.1.1 ServerSpec Structure

A file transfer server has access to a list of objects: files, functions and variables. This list is defined as a global array in **ZSERVER.LIB**.

```
ServerSpec server_spec[SSPEC_MAXSPEC];
```

Throughout this manual, this array will be called the TCP/IP servers' object list.

### 3.1.2 ServerAuth Structure

**ZSERVER.LIB** also defines a global array that is a list of user name/password pairs.

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

Throughout this manual, this array will be called the TCP/IP users list.

### 3.1.3 FormVar Structure

An array of **FormVars** 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 TCP/IP servers' object 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 she wants the variable to be 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 (e.g., the integrity-checking properties) for these **FormVar** entries 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.

## 3.2 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 3.4, some as function parameters and some as return values.

### 3.2.1 ServerSpec Type Field

This field describes the objects in the TCP/IP servers' object list.

```

SSPEC_ERROR      // Error condition
SSPEC_FILE       // Data resides in a file
SSPEC_FSFILE     // The data resides in a file system file
SSPEC_FORM       // Set of modifiable variables
SSPEC_FUNCTION   // Data is a function
SSPEC_ROOTFILE  // Data resides in root memory
SSPEC_UNUSED
SSPEC_VARIABLE   // Data is a variable (for HTTP)
SSPEC_XMEMFILE   // Data resides in extended memory
SSPEC_ROOTVAR    // Data is a variable in root memory
SSPEC_XMEMVAR    // Data is a variable in xmem

```

### 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 Servermask field

The type of server (HTTP and/or FTP) that has access to a particular data structure is determined by the servermask field. Both **ServerSpec** and **ServerAuth** have this field. It must be set when adding the structure to its array. The default is that no server has access. **servermask** can be one of the following, or any bitwise inclusive OR of these values:

```

SERVER_FTP
SERVER_HTTP
SERVER_USER      // for use with the flash file system

```

### 3.2.4 Configurable Constants

These constants define system limits on various data lengths and array sizes.

**SSPEC\_MAXNAME**

Maximum length of strings in a **ServerSpec** structure entry. Default is 20.

**SSPEC\_MAXSPEC**

Sets the maximum number of entries in the global array, **server\_spec**. **HTTP\_MAXRAMSPEC** (from **HTTP.LIB**) should override **SSPEC\_MAXSPEC**. If you attempt to use both you may not get the desired results, therefore, the use of **HTTP\_MAXRAMSPEC** should be deprecated. If both **HTTP\_MAXRAMSPEC** and **SSPEC\_MAXSPEC** are not defined, **SSPEC\_MAXSPEC** defaults to 10.

**SSPEC\_XMEMVARLEN**

Defines the size of the stack-allocated buffer used by **sspec\_readvariable()** when reading a variable in xmem. It defaults to 20.

**SAUTH\_MAXNAME**

Maximum length of strings in **ServerAuth** structure. Default is 20.

**SAUTH\_MAXUSERS**

Maximum number of users for a TCP/IP users list. Default is 10.

### 3.3 HTML Forms

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 4.3.4 on page 152 for an example program.

## 3.4 Functions

### sauth\_adduser

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

#### DESCRIPTION

Adds a user to the TCP/IP users list.

#### PARAMETERS

|                   |  |
|-------------------|--|
| <b>username</b>   | Name of the user.  |
| <b>password</b>   | Password of the user.  |
| <b>servermask</b> | Bitmask representing valid servers (e.g. <b>SERVER_HTTP</b> , <b>SERVER_FTP</b> ). |

#### RETURN VALUE

-1: Failure;  
>=0: Success; index in TCP/IP users list (id passed to **sauth\_getusername()**).

#### LIBRARY

ZSERVER.LIB

#### SEE ALSO

sauth\_authenticate, sauth\_getwriteaccess,  
sauth\_setwriteaccess



## sauth\_authenticate

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

### DESCRIPTION

Authenticate a user.

### PARAMETERS

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

### RETURN VALUE

-1: Failure, user not valid.  
>=0: Success, array index of the **ServerAuth** structure for authenticated user.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sauth\_adduser

## sauth\_getusername

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

### DESCRIPTION

Gets a pointer to **username** from the **ServerAuth** structure.

### PARAMETERS

**uid**                    The user's id, i.e., the array index in the TCP/IP users list.

### RETURN VALUE

**NULL**: Failure;

**!NULL**: Success, pointer to the **username** string on success.

### LIBRARY

ZSERVER.LIB

### See also

sspec\_getusername

## sauth\_getwriteaccess

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

### DESCRIPTION

Checks whether or not a user has write access.

### PARAMETERS

**sauth**                    Index of the user in the TCP/IP users list.

### RETURN VALUE

**0**: User does not have write access;

**1**: User has write access

**-1**: Failure.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sauth\_setwriteaccess

## **sauth\_setwriteaccess**

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

### **DESCRIPTION**

Sets the write accessibility of a user.

### **PARAMETERS**

|                    |  |
|--------------------|--|
| <b>sauth</b>       | Index of the user in the TCP/IP users list.                          |
| <b>writeaccess</b> | Set to <b>1</b> to give write access, <b>0</b> to deny write access. |

### **RETURN VALUE**

**0**: Success  
**-1**: Failure

### **LIBRARY**

ZSERVER.LIB

### **SEE ALSO**

sauth\_getwriteaccess

## **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. This function is currently only useful for the HTTP server.

### **PARAMETERS**

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

### **RETURN VALUE**

**>=0**: Success; location of form in TCP/IP servers' object 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 file system to the TCP/IP servers' object list.

### **PARAMETERS**

|                   |  |
|-------------------|--|
| <b>name</b>       | Name of the new file                   |
| <b>filenum</b>    | Number of the file in the file system. |
| <b>servermask</b> | Bitmask representing valid servers.    |

### **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 list of objects recognized by the server. This function is currently only useful for HTTP servers.

### PARAMETERS

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

### 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\_addxmemfile, sspec\_aliasspec

## sspec\_addfv

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

### DESCRIPTION

Adds a variable to a form.

### PARAMETERS

|             |   |
|-------------|---|
| <b>form</b> | Index of the form in the TCP/IP servers' object list.     |
| <b>var</b>  | Index of the variable in the TCP/IP servers' object list. |

### RETURN VALUE

-1: Failure;  
>=0: Success; next available index into the **FormVar** array.

### LIBRARY

ZSERVER.LIB

## 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 TCP/IP servers' object list.

### PARAMETERS

|                   |   |
|-------------------|---|
| <b>name</b>       | Name of the new file.   |
| <b>fileloc</b>    | Pointer to the beginning of the file.   |
| <b>len</b>        | Length of the file in bytes.  |
| <b>servermask</b> | Bitmask representing servers for which this entry will be valid (e.g. <b>SERVER_HTTP</b> , <b>SERVER_FTP</b> ). |

### RETURN VALUE

-1: Failure;  
>=0: Success, location of the file in the TCP/IP servers' object list.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sspec\_adddfsfile, sspec\_addxmemfile, sspec\_addvariable,  
sspec\_addfunction sspec\_addform, sspec\_aliasspec

## sspec\_addvariable

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

### DESCRIPTION

Adds a variable to the TCP/IP servers' object list. This function is currently only useful for the HTTP server.

### PARAMETERS

|                   |  |
|-------------------|--|
| <b>name</b>       | Name of the new variable.  |
| <b>variable</b>   | Address of actual variable.  |
| <b>type</b>       | Type of the variable (e.g., <b>INT8</b> , <b>INT16</b> , <b>PTR16</b> , etc.).                                       |
| <b>format</b>     | Output format of the variable.   |
| <b>servermask</b> | Bitmask representing servers for which this function will be valid (currently only useful with <b>SERVER_HTTP</b> ). |

### RETURN VALUE

-1: Failure;  
>=0: Success, the location of the variable in the TCP/IP servers' object list.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sspec\_addfsfile, sspec\_addrootfile, sspec\_addxmemfile,  
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 TCP/IP servers' object list.

### PARAMETERS

|                   |  |
|-------------------|--|
| <b>name</b>       | Name of the new file.  |
| <b>fileloc</b>    | Location of the beginning of the file. The first 4 bytes of the file must represent the length of the file ( <b>#ximport</b> does this automatically). |
| <b>servermask</b> | Bitmask representing servers for which this entry will be valid (e.g. <b>SERVER_HTTP</b> , <b>SERVER_FTP</b> ).  |

### RETURN VALUE

-1: Failure;  
>=0: Success, the location of the file in the TCP/IP servers' object list.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sspec\_adddfsfile, 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 TCP/IP servers' object list. Currently, this function is useful only for the HTTP server.

### PARAMETERS

|                   |  |
|-------------------|--|
| <b>name</b>       | Name of the new variable.  |
| <b>variable</b>   | Address of the variable in extended memory.  |
| <b>type</b>       | Variable type (e.g., <b>INT8</b> , <b>INT16</b> , <b>PTR16</b> , etc.).              |
| <b>format</b>     | Output format of the variable.   |
| <b>servermask</b> | Bitmask representing valid servers (currently only useful with <b>SERVER_HTTP</b> ). |

### RETURN VALUE

-1: Failure;  
>=0: Success, the location of the variable in the TCP/IP servers' object 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 object in the TCP/IP servers' object list. Please note, this is NOT a deep copy. That is, any file, variable, or form that the alias references will be the same copy of the file, variable, or form that already exists in the TCP/IP servers' object list. This should be called only when the original entry has been completely set up.

### **PARAMETERS**

|              |   |
|--------------|---|
| <b>sspec</b> | Location of the object in the TCP/IP servers' object list that will be aliased. |
| <b>name</b>  | Name field of the <b>ServerSpec</b> structure that will be aliased.             |

### **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\_checkaccess**

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

### **DESCRIPTION**

This function checks whether or not the specified user has permission to access the specified object in the TCP/IP servers' object list.

### **PARAMETERS**

**sspec**                Location of object in TCP/IP servers' object list.  
**uid**                 Location of the user in the TCP/IP users list.

### **RETURN VALUE**

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

### **LIBRARY**

ZSERVER.LIB

### **SEE ALSO**

sspec\_needsauthentication

## **sspec\_findfv**

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

### **DESCRIPTION**

Finds the index in the array of type **FormVar** of a form variable in a given form.

### **PARAMETERS**

**form**                Location of the form in the TCP/IP servers' object list.  
**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

## **sspec\_findname**

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

### **DESCRIPTION**

Finds the location of the object associated with **name** and returns the location (index into the **server\_spec** array) of the object if the server is allowed access to it. (Access is determined by the **servermask** field in the **ServerSpec** structure for the object.)

### **PARAMETERS**

|               |   |
|---------------|---|
| <b>name</b>   | Name to search for in the TCP/IP servers' object list.    |
| <b>server</b> | The server making the request (e.g. <b>SERVER_HTTP</b> ). |

### **RETURN VALUE**

-1: Failure;  
>=0: Success, location of the object in the TCP/IP servers' object list.

### **LIBRARY**

ZSERVER.LIB

### **SEE ALSO**

sspec\_findnextfile

## sspec\_findnextfile

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

### DESCRIPTION

Finds the first **ServerSpec** structure in the array, at or following the structure indexed by **start**, that is associated with a file and that is accessible by the server.

### PARAMETERS

**start**                   The array index at which to begin the search.

**server**                  The server making the request (e.g. **SERVER\_HTTP**).

### RETURN VALUE

-1: Failure;  
>=0: Success, index of requested **ServerSpec** structure.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sspec\_findname

## 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. Note that the location of the file is returned as a long; the return value should be cast to the appropriate type (**char\*** for a root file, **FileNum** for the file system) by the user. **sspec\_getfiletype()** can be used to find the file type.

### PARAMETERS

**sspec**                   Index into the array of **ServerSpec** structures.

### 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

Gets the type of a file represented by a **ServerSpec** structure.

### PARAMETERS

**sspec**                    Index into the array of **ServerSpec** structures.

### RETURN VALUE

**SSPEC\_ERROR**: Failure;  
**!=SSPEC\_ERROR**: Success, the type of file.

### 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** on failure;  
**!NULL** on success, title string.

### LIBRARY

ZSERVER.LIB

## **sspec\_getfunction**

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

### **DESCRIPTION**

Accesses the array of **ServerSpec** structures to get a pointer to the requested function.

### **PARAMETERS**

**sspec**                    Index into the array of **ServerSpec** structures.

### **RETURN VALUE**

**NULL** on failure;  
**!NULL** on 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**

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable. |

### **RETURN VALUE**

**NULL** on failure;  
**!NULL** on success, description string.

### **LIBRARY**

ZSERVER.LIB

## 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

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> 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

## 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

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable. |

### RETURN VALUE

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

### LIBRARY

ZSERVER.LIB

## **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**

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

### **RETURN VALUE**

**NULL** on failure;  
**!NULL**, name of the form variable.

### **LIBRARY**

ZSERVER.LIB

## **sspec\_getfvnum**

```
int sspec_getfvnum(int form);
```

### **DESCRIPTION**

Gets the number of variables in a form.

### **PARAMETERS**

|             |                                       |
|-------------|---------------------------------------|
| <b>form</b> | <b>server_spec</b> index of the form. |
|-------------|---------------------------------------|

### **RETURN VALUE**

**-1**: Failure;  
**>=0**: Success, number of form variables.

### **LIBRARY**

ZSERVER.LIB

## sspec\_getfvopt

```
char* sspec_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

|               |  |
|---------------|--|
| <b>form</b>   | <b>server_spec</b> index of the form.                              |
| <b>var</b>    | Index into the array of <b>FormVar</b> structures of the variable. |
| <b>option</b> | Index of the form variable option.                                 |

### RETURN VALUE

**NULL** on failure;  
**!NULL** on success, form variable option.

### LIBRARY

ZSERVER.LIB

## 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

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable. |

### RETURN VALUE

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

### LIBRARY

ZSERVER.LIB

## **sspec\_getfvreadonly**

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

### DESCRIPTION

Checks if a form variable is read-only.

### PARAMETERS

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable. |

### RETURN VALUE

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

### LIBRARY

ZSERVER.LIB

## **sspec\_getfvspec**

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

### DESCRIPTION

Gets the **server\_spec** index of a variable in a form.

### PARAMETERS

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

### RETURN VALUE

-1: Failure;  
>=0: Success, location of the form variable in the TCP/IP servers' object list.

### LIBRARY

ZSERVER.LIB

## **sspec\_getlength**

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

### **DESCRIPTION**

Gets the length of the file associated with the specified **ServerSpec** structure.

### **PARAMETERS**

**sspec**                    Location of file in TCP/IP servers' object list.

### **RETURN VALUE**

-1: Failure;  
>=0: Success, length of the file in bytes.

### **LIBRARY**

ZSERVER.LIB

### **SEE ALSO**

sspec\_readfile, sspec\_getfileloc

## **sspec\_getname**

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

### **DESCRIPTION**

Accesses the array of **ServerSpec** structures and returns a pointer to the object's name.

### **PARAMETERS**

**sspec**                    Location of object in TCP/IP servers' object list.

### **RETURN VALUE**

**NULL**: Failure;  
**!NULL**: Success, pointer to name string.

### **LIBRARY**

ZSERVER.LIB

## sspec\_getrealm

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

### DESCRIPTION

Returns the realm for the object.

### PARAMETERS

**sspec**                    Location of the object in the TCP/IP servers' object list.

### RETURN VALUE

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

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sspec\_setrealm

## sspec\_gettype

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

### DESCRIPTION

Gets the type field of a **ServerSpec** structure.

### PARAMETERS

**sspec**                    Location of the object in the TCP/IP servers' object list.

### RETURN VALUE

**SSPEC\_ERROR**: Failure;  
type field: Success (See "Constants Used in Zserver.lib" on page 88). For files and variables, it returns the generic type **SSPEC\_FILE** or **SSPEC\_VARIABLE**, respectively.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sspec\_getfiletype, sspec\_getvartype

## sspec\_getusername

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

### DESCRIPTION

Gets the username field of a **ServerAuth** structure.

### PARAMETERS

**sspec**                    Location of user in TCP/IP users list.

### RETURN VALUE

**NULL**: Failure;  
**!NULL**: Success, pointer to **username**.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sauth\_adduser, sspec\_setuser

## sspec\_getvaraddr

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

### DESCRIPTION

Returns a pointer to the requested variable in the TCP/IP servers' object list.

### PARAMETERS

**sspec**                    Location of the variable in the TCP/IP servers' object list.

### RETURN VALUE

**NULL** on failure;  
**!NULL** on success, pointer to variable.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sspec\_readvariable



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

**DESCRIPTION** Returns the kind of variable represented by **sspec** (**INT8**, **INT16**, **INT32**, **FLOAT32**, or **PTR16**).

**PARAMETERS**

**sspec** Location of the variable in the TCP/IP servers' object list.

**RETURN VALUE**

the kind of variable;  
0: Failure.

**LIBRARY**

ZSERVER.LIB

**SEE**

## sspec\_needsauthentication

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

### DESCRIPTION

Checks if an object in the TCP/IP servers' object list needs user authentication to permit access. There is a field in the **ServerSpec** structure that is an index into the array of **ServerAuth** structures (list of valid users). If this field has a value, access to the object is limited to the one user specified.

### PARAMETERS

**sspec**                    Index into the array of **ServerSpec** structures.

### RETURN VALUE

0: Does not need authentication;  
1: Does need authentication;  
-1: Failure.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

sspec\_getrealm

## 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.

### PARAMETERS

|               |   |
|---------------|---|
| <b>sspec</b>  | Index into the array of <b>ServerSpec</b> structures.                           |
| <b>buffer</b> | The buffer to put the file contents into.                                       |
| <b>offset</b> | The offset from the start of the file, in bytes, at which copying should begin. |
| <b>len</b>    | 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**                Index into the array of **ServerSpec** structures.

**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 an object from the TCP/IP servers' object list.

### PARAMETERS

**sspec**                Index into the array of **ServerSpec** structures.

### RETURN VALUE

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

### LIBRARY

ZSERVER.LIB

## 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: Success.  
-1: Failure.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

`sspec_save`, `sspec_setsavedata`

## 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: Success.  
-1: Failure.

### LIBRARY

ZSERVER.LIB

### SEE ALSO

`sspec_restore`, `sspec_setsavedata`

## **sspec\_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\_redirectto** 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**

|                 |   |
|-----------------|---|
| <b>form</b>     | Index into the array of <b>ServerSpec</b> structures.   |
| <b>function</b> | Index into the array of <b>ServerSpec</b> structures. This is the return value of the function <b>sspec_addfunction()</b> . |

### **RETURN VALUE**

0 : Success.  
-1 : Failure.

### **LIBRARY**

ZSERVER.LIB

### **SEE ALSO**

sspec\_addfunction

## **sspec\_setformfunction**

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

### **DESCRIPTION**

Sets the function that will generate the form.

### **PARAMETERS**

|             |   |
|-------------|---|
| <b>form</b> | <b>server_spec</b> index of the form.                             |
| <b>fptr</b> | Form generation function ( <b>NULL</b> 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**

|                 |  |
|-----------------|--|
| <b>form</b>     | Index into the array of <b>ServerSpec</b> structures.  |
| <b>function</b> | Index into the array of <b>ServerSpec</b> structures. This is the return value of <b>sspec_addfunction()</b> . |

### **RETURN VALUE**

0: Success.  
-1: Failure.

### **LIBRARY**

ZSERVER.LIB

### **SEE ALSO**

sspec\_addfunction



## **sspec\_setformtitle**

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

### **DESCRIPTION**

Sets the title for an automatically generated form.

### **PARAMETERS**

|              |                                       |
|--------------|---------------------------------------|
| <b>form</b>  | <b>server_spec</b> index of the form. |
| <b>title</b> | Title of the HTML page.               |

### **RETURN VALUE**

0: Success  
-1: Failure;

### **LIBRARY**

ZSERVER.LIB

## 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

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

### RETURN VALUE

>0: Success  
-1: Failure

### LIBRARY

ZSERVER.LIB

## 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

|             |   |
|-------------|---|
| <b>form</b> | <b>server_spec</b> index of the form.                                 |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable.                |
| <b>desc</b> | Description of the variable. This text will display on the html page. |

### RETURN VALUE

0: Success  
-1: Failure

### LIBRARY

ZSERVER.LIB

## 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

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

### RETURN VALUE

0: Success  
-1: Failure

### LIBRARY

ZSERVER.LIB

## sspec\_setfvfloatrange

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

### DESCRIPTION

Sets the range of a float.

### PARAMETERS

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable. |
| <b>low</b>  | Minimum value of the variable.                         |
| <b>high</b> | Maximum value of the variable.                         |

### RETURN VALUE

0: Success  
-1: Failure

### LIBRARY

ZSERVER.LIB

## 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).

### PARAMETERS

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable. |
| <b>len</b>  | Length of the variable.                                |

### RETURN VALUE

0: Success  
-1: Failure

### LIBRARY

ZSERVER.LIB

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

#### DESCRIPTION

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

#### PARAMETERS

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable. |
| <b>name</b> | Display name of the variable.                          |

#### RETURN VALUE

0: Success  
-1: Failure

#### LIBRARY

ZSERVER.LIB

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

#### DESCRIPTION

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

#### PARAMETERS

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

#### RETURN VALUE

0: Success  
-1: Failure

#### LIBRARY

## **sspec\_setfvrange**

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

### DESCRIPTION

Sets the range of an integer.

### PARAMETERS

|             |  |
|-------------|--|
| <b>form</b> | <b>server_spec</b> index of the form.                  |
| <b>var</b>  | Index (into the <b>FormVar</b> array) of the variable. |
| <b>low</b>  | Minimum value of the variable.                         |
| <b>high</b> | Maximum value of the variable.                         |

### RETURN VALUE

0: Success  
-1: Failure

### LIBRARY

ZSERVER.LIB

## **sspec\_setfvreadonly**

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

### DESCRIPTION

Sets the form variable to be read-only.

### PARAMETERS

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

### RETURN VALUE

0: Success  
-1: Failure

### LIBRARY

ZSERVER.LIB

## **sspec\_setrealm**

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

### **DESCRIPTION**

Sets the **realm** field of a **ServerSpec** structure for HTTP authentication purposes.

### **PARAMETERS**

|              |   |
|--------------|---|
| <b>sspec</b> | Index into the array of <b>ServerSpec</b> structures. |
| <b>realm</b> | 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**

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

### **RETURN VALUE**

0: Success  
-1: Failure

### **LIBRARY**

ZSERVER.LIB

### **SEE ALSO**

**sspec\_save**, **sspec\_restore**



## **sspec\_setuser**

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

### **DESCRIPTION**

Sets the user (owner) of a **ServerSpec** structure.

### **PARAMETERS**

|              |   |
|--------------|---|
| <b>sspec</b> | Index into the array of <b>ServerSpec</b> structures.                   |
| <b>uid</b>   | Index into the array of <b>ServerAuth</b> structures (identifies user). |

### **RETURN VALUE**

0: Success  
-1: Failure

### **LIBRARY**

ZSERVER.LIB

### **SEE ALSO**

sauth\_adduser, sspec\_getusername



# HTTP Server 4

An HTTP (Hypertext Transfer Protocol) server makes HTML (Hypertext Markup Language) documents and other documents available to clients, i.e., web browsers. HTTP is implemented by `HTTP.LIB`.

## 4.1 HTTP Server Data Structures

There are four data structures in `HTTP.LIB` of interest to developers of HTTP servers.

### 4.1.1 HttpSpec

The data structure `HttpSpec` contains all the files, variables, and functions the Web server has access to. The structure `ServerSpec` from `ZSERVER.LIB` may be instead.

```
typedef struct {
    word type;
    char name[HTTP_MAXNAME];
    long data;
    void* addr;
    word vartype;
    char* format;
    HttpRealm* realm;
} HttpSpec;
```

#### 4.1.1.1 HttpSpec fields

|                |  |
|----------------|--|
| <b>type</b>    | This field tells the server if the entry is a file, variable or function ( <code>HTTPSPEC_FILE</code> , <code>HTTPSPEC_VARIABLE</code> or <code>HTTPSPEC_FUNCTION</code> , respectively).  |
| <b>name</b>    | This field specifies a unique name for referring to the entry. The Web server recognizes “/index.html” as the entity that matches “http://someurl.com/index.html”, and delivers the entry’s content based on the value of <b>type</b> (the first field). |
| <b>data</b>    | The third field is the physical address of the entity.   |
| <b>addr</b>    | The fourth field is a short pointer to the entity. Either the third field or the fourth field is valid, not both. All files must use the physical address, variables and functions use the short pointer.  |
| <b>vartype</b> | This field describes the type of variable. Supported types are : <code>INT8</code> <code>INT16</code> , <code>PTR16</code> , <code>INT32</code> , and <code>FLOAT</code> .   |
| <b>format</b>  | The format field describes the <code>printf</code> format specifier used to display the variable.  |
| <b>realm</b>   | This field is the name and password required to access the entity.   |

### 4.1.2 `HttpType`

The structure `HttpType` associates a file extension with a MIME type (Multipurpose Internet Mail Extension) and a function which handles the MIME type. 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[20];
    int (*fptr)(/* HttpState* */);
} HttpType;
```

### 4.1.3 `HttpRealm`

The structure `HttpRealm` holds user-ID and password pairs for partitions called realms. These realms allow the protected resources on a server to be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database.

```
typedef struct {
    char username[HTTP_MAXNAME];
    char password[HTTP_MAXNAME];
    char realm[HTTP_MAXNAME];
} HttpRealm;
```

HTTP/1.0 Basic authentication is used. 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.

In the `HttpSpec` structure, there is a pointer to a structure of type `HttpRealm`. To password-protect the entity, add the name, password, and realm desired. If you do not want to password-protect the entity, leave the realm pointer in the `HttpSpec` structure `NULL`.

#### 4.1.4 HttpState

Use of this structure is necessary for CGI functions. Some of the fields are off-limits to developers.

```
typedef struct {
    tcp_socket s;

    /* State information */
    int state, substate, subsubstate, nextstate, laststate;

    /* File referenced */
    HttpSpecAll spec, subspec;
    HttpType *type;
    int (*handler)(), (*exec)();

    /* rx/tx state variables */
    long offset;
    long length;
    long filelength, subfilelength;
    long pos, subpos;
    long timeout, long main_timeout;
    char buffer[HTTP_MAXBUFFER];
    char *p;

    /* http request and header info */
    char method;
    char url[HTTP_MAXURL];
    char version;
    char connection;
    char content_type[40];
    long content_length;
    char has_form;
    char finish_form;
    char username[HTTP_MAXNAME];
    char password[HTTP_MAXNAME];
    char cookie[HTTP_MAXNAME];

    /* other - don't touch */
    int headerlen;
    int headeroff;
    char tag[HTTP_MAXNAME];
    char value[HTTP_MAXNAME];
} HttpState;
```

#### 4.1.4.1 HttpState Fields

The fields discussed here are available for developers to use in their application programs.

|                                       |   |
|---------------------------------------|---|
| <b>s</b>                              | This is the socket associated with the given HTTP server. A developer can use this in a CGI function to output dynamic data. Any of the TCP functions can be used.  |
| <b>substate</b><br><b>subsubstate</b> | These are intended to be used to hold 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 <b>http_handler()</b> 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.  |
| <b>timeout</b>                        | This value can be used by the CGI function to implement an internal timeout.  |
| <b>main_timeout</b>                   | This value holds the timeout that is used by the web server. The web server checks against this timeout on every call of <b>http_handler()</b> . When the web server changes states, it resets <b>main_timeout</b> . 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:<br><pre>state-&gt;main_timeout = set_timeout(HTTP_TIMEOUT);</pre> <b>HTTP_TIMEOUT</b> is the number of seconds until the web server will time out. It is 16 seconds by default. |
| <b>buffer[]</b>                       | A buffer that the developer can use to put data to be transmitted over the socket. It is of size <b>HTTP_MAXBUFFER</b> .  |
| <b>p</b>                              | Pointer into the buffer given above.  |
| <b>method</b>                         | This should be treated as read-only. It holds the method by which the web request was submitted. The value is either <b>HTTP_METHOD_GET</b> or <b>HTTP_METHOD_POST</b> , for the GET and POST request methods, respectively.  |
| <b>url[]</b>                          | This should be treated as read-only. It holds the URL by which the current web request was submitted.   |
| <b>version;</b>                       | This should be treated as read-only. This holds the version of the HTTP request that was made. It can be <b>HTTP_VER_09</b> , <b>HTTP_VER_10</b> , or <b>HTTP_VER_11</b> for 0.9, 1.0, or 1.1 requests, respectively.   |

|                                      |  |
|--------------------------------------|--|
| <b>content_type[ ]</b>               | This should be treated as read-only. This buffer holds the value from the Content-Type header sent by the client.  |
| <b>content_length;</b>               | 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.   |
| <b>username[ ]</b>                   | Read-only buffer has username of the user making the request, if authentication took place.  |
| <b>password[ ]</b>                   | Read-only buffer has password of the user making the request, if authentication took place.  |
| <b>cookie[ ]</b>                     | Read-only buffer contains the value of the cookie "DCRABBIT" (see <code>http_setcookie()</code> for more information).   |
| <b>headerlen</b><br><b>headeroff</b> | These variables can be used in conjunction to cause the web server to flush data from the <code>buffer[ ]</code> array in the <code>HttpState</code> structure. <b>headerlen</b> should be set to the amount of data in <code>buffer[ ]</code> , and <b>headeroff</b> should be set to 0 (to indicate the offset into the array). When the CGI function is called the next time, the data in <code>buffer[ ]</code> will be flushed to the socket. |

## 4.2 Configuration Constants

The following macros are available in `HTTP.LIB`:

### `HTTP_MAXNAME`

This is the maximum length for a name in the `HttpSpec` structure. This defaults to 20 characters. Without overriding this value, the maximum length of any name is 19 characters because one character is used for the `NULL` termination.

### `HTTP_MAXRAMSPEC`

This is the maximum number of `HttpSpec` entries that can be added at runtime. This macro overrides `SSPEC_MAXSPEC`.

### `HTTP_MAXSERVERS`

This is the maximum number of HTTP servers listening on port 80. The default is two. 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 four: 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.

## **TIMEZONE**

This macro specifies the distance in hours you are from Greenwich Mean Time (GMT), which is 5 hours ahead of Eastern Standard Time (EST). The default **TIMEZONE** is -8, which represents Pacific Standard Time. You can use the **tm\_wr** function to set the clock to the correct value. If you lose power and don't have the battery-backup option, the time will need to be reset.

## **4.3 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. Unless you are using BOOTP/DHCP, the macros in this block need to be changed to reflect your network settings. For most HTTP programs, you will be concerned with **TIMEZONE** and the IP address macros: **MY\_IPADDRESS**, **MY\_NETMASK**, **MY\_GATEWAY**.

### **4.3.1 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.



```

// Static.c
#define MY_IP_ADDRESS    "10.10.6.100"
#define MY_NETMASK      "255.255.255.0"
#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

const HttpType http_types[] =
{
    { ".html", "text/html", NULL},
    { ".gif", "image/gif", NULL}
};
const HttpSpec http_flashspec[] =
{
    {HTTPSPEC_FILE, "/", index_html, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/index.html", index_html, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/rabbit1.gif", rabbit1_gif, NULL, 0, NULL, NULL},
};
main()
{
    sock_init(); // Initializes the TCP/IP stack
    http_init(); // Initializes the web server

    tcp_reserveport(80);
    while (1) {
        http_handler();
    }
}

```

This 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, modify this line in `Static.c`:

```
#ximport "samples/tcpip/http/pages/static.html" index_html
```

#### 4.3.1.1 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`:

```
HttpSpec http_flashspec[] =
{
    {HTTPSPEC_FILE, "/", index_html, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/index.html", index_html, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/newfile.html", index_html, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/rabbit1.gif", rabbit1_gif, NULL, 0, NULL, NULL},
};
```

Insert the name of your new file, preceded by “/”, into this structure, using the same format as the other lines. Compile and run the program. Open up your browser to the new page (e.g. “http://10.10.6.100/newfile.html”), and your new page will be displayed by the browser.

#### 4.3.1.2 Adding Files with Different Extensions

If you are adding a file with an extension that is not html or gif, you will need to make an entry in the `HttpType` 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.

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

#### 4.3.1.3 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 `http_types[]`.

### 4.3.2 Dynamic Web Pages Without HTML Forms

Serving a dynamic web page without the use of HTML forms is done by sample program `Ssi.c`, shown below and located in `/Samples/Tcpip/Http`. This program displays four 'lights' and four buttons to toggle them. Users can browse to the device and change the status of the lights.

```
#define MY_GATEWAY "10.10.6.19"
#define MY_IP_ADDRESS "10.10.6.100"
#define MY_NETMASK "255.255.255.0"

#define SOCK_BUF_SIZE 2048
#define HTTP_MAXSERVERS 1
#define MAX_SOCKETS 1

#define REDIRECTHOST MY_IP_ADDRESS
#define REDIRECTTO "http: //" REDIRECTHOST "/index.shtml"

#mmap xmem
#use "dcrtcp.lib"
#use "http.lib"

/*
 * The source code for this program is ximported. This allows
 * us to put the line <!--#include file="ssi.c" --> in the
 * file Samples/Tcpip/Http/Pages/Showsrc.shtml.
 */

#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

/*
 * In this case the extension .shtml is the first type in
 * the type table. This causes the default (no extension)
 * to assume the shtml_handler.
 */

const HttpType http_types[] = {
    { ".shtml", "text/html", shtml_handler}, // ssi
    { ".html", "text/html", NULL},          // html
    { ".cgi", "", NULL},                    // cgi
    { ".gif", "image/gif", NULL}
};
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)
{
    if (strcmp(led2,"ledon.gif")==0)
        strcpy(led2,"ledoff.gif");
    else
        strcpy(led2,"ledon.gif");
    cgi_redirectto(state,REDIRECTTO);
    return 0;
}
int led3toggle(HttpState* state)
{
    if (strcmp(led3,"ledon.gif")==0)
        strcpy(led3,"ledoff.gif");
    else
        strcpy(led3,"ledon.gif");
    cgi_redirectto(state,REDIRECTTO);
    return 0;
}
int led4toggle(HttpState* state)
{
    if (strcmp(led4,"ledon.gif")==0)
        strcpy(led4,"ledoff.gif");
    else
        strcpy(led4,"ledon.gif");
    cgi_redirectto(state,REDIRECTTO);
    return 0;
}
```

```

const HttpSpec http_flashspec[] = {
    {HTTPSPEC_FILE, "/", index_html, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/index.shtml", index_html, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/showsrc.shtml", showsrc_shtml, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/rabbit1.gif", rabbit1_gif, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/ledon.gif", ledon_gif, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/ledoff.gif", ledoff_gif, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "/button.gif", button_gif, NULL, 0, NULL, NULL},
    {HTTPSPEC_FILE, "ssi.c", ssi_c, NULL, 0, NULL, NULL},
    {HTTPSPEC_VARIABLE, "led1", 0, led1, PTR16, "%s", NULL},
    {HTTPSPEC_VARIABLE, "led2", 0, led2, PTR16, "%s", NULL},
    {HTTPSPEC_VARIABLE, "led3", 0, led3, PTR16, "%s", NULL},
    {HTTPSPEC_VARIABLE, "led4", 0, led4, PTR16, "%s", NULL},
    {HTTPSPEC_FUNCTION, "/led1tog.cgi", 0, led1toggle, 0, NULL, NULL},
    {HTTPSPEC_FUNCTION, "/led2tog.cgi", 0, led2toggle, 0, NULL, NULL},
    {HTTPSPEC_FUNCTION, "/led3tog.cgi", 0, led3toggle, 0, NULL, NULL},
    {HTTPSPEC_FUNCTION, "/led4tog.cgi", 0, led4toggle, 0, NULL, NULL},
};

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 Common Gateway Interface (CGI).

### 4.3.2.1 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 recognized by the HTTP server by the file extension `shtml`.

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 and replaces the SSI command with the output.

### Dynamically changing a variable on a web page

The `Ssi.shtml` file, located in the `/Samples/Tcpip/Http/Pages` folder, 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 `http_flashspec` structure.

```
HttpSpec http_flashspec[] =
{
    //...
    { HTTPSPEC_VARIABLE, "led1", 0, led1, PTR16, "%s", NULL}
    //...
};
```

`shtml_handler` 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">
```

or

```
<img SRC="ledoff.gif">
```

This causes the browser to load the appropriate image file.

#### 4.3.2.2 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 an URL corresponding to a CGI program, the server will execute the CGI program in real-time.

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.

#### 4.3.3 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 4.3.3.1, "Sample HTML Page," on page 146, 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.

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 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**".

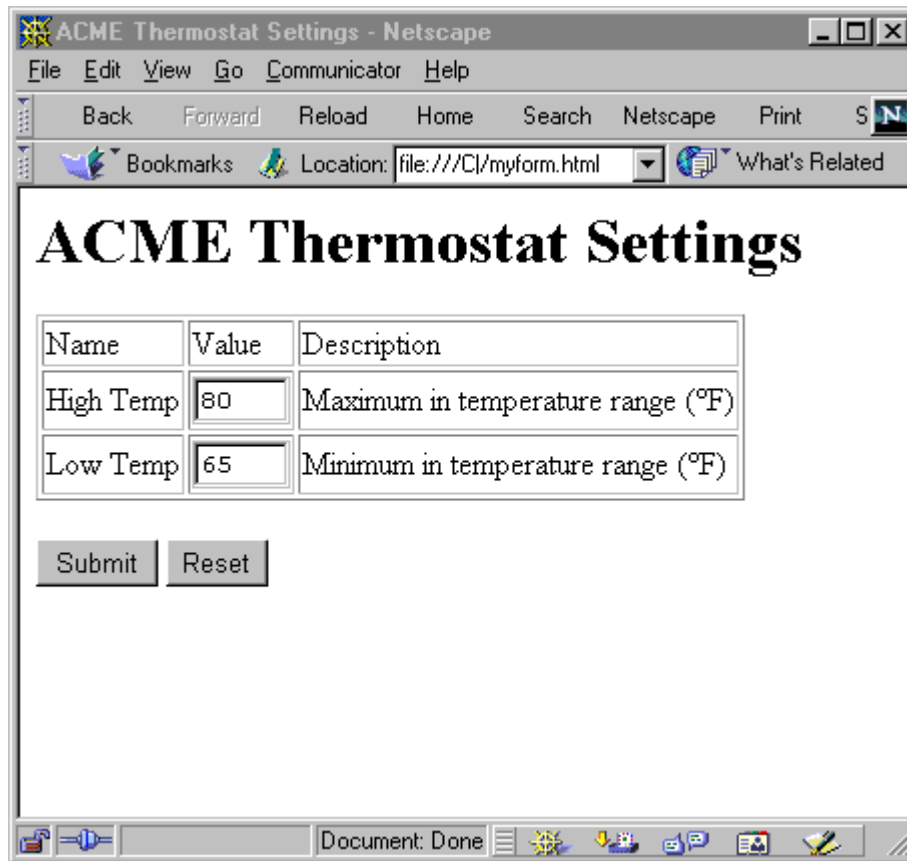
### 4.3.3.1 Sample HTML Page

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">
  </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 check the validity of the new values, and then assign them to the appropriate C variable if they are valid.

### 4.3.3.2 POST-style form submission

If an HTML file specifies a POST-style form submission (i.e., `METHOD="POST"`), the form will still be waiting on the socket when the 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 known 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.

### 4.3.3.3 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_FORMSIZE64
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()` reads URL-encoded data off the network, and calls `parse_token()` to store the data in `FORMSpec[]`.

```
// Parse one token 'foo=bar', matching 'foo' to the name field in
// the struct, and store 'bar' into the value

void parse_token(HttpState* state) {
    int i, len;
    for(i=0; i<HTTP_MAXBUFFER; i++) {
        if(state->buffer[i] == '=')
            state->buffer[i] = '\0';
    }
    state->p = state->buffer + strlen(state->buffer) + 1;
    for(i=0; i<(sizeof(FORMSpec)/sizeof(FORMType)); i++) {
        if(!strcmp(FORMSpec[i].name, state->buffer)) {
            len = (strlen(state->p) > MAX_FORMSIZE) ? MAX_FORMSIZE - 1 :
                strlen(state->p);
            strncpy(FORMSpec[i].value, state->p, 1+len);
            FORMSpec[i].value[MAX_FORMSIZE - 1] = '\0';
        }
    }
}
```

```
// Read URL-encoded data and call parsing function to store data
int parse_post(HttpState* state) {
    int ret;
    while(1) {
        ret = sock_fastread((sock_type *)&state->s, state->p, 1);
        if(0 == ret) {
            *state->p = '\0';
            parse_token(state);
            return 1;
        }
        if((*state->p=='&' || (*state->p=='\r') || (*state->p=='\n'))
        { /* found one token */
            *state->p = '\0';
            parse_token(state);
            state->p = state->buffer;
        } else {
            state->p++;
        }
    }
    if((state->p - state->buffer) > HTTP_MAXBUFFER) {
        /* input too long */
        return 1;
    }
}
```

#### 4.3.3.4 Sample of a CGI Handler

This next function is the CGI handler. 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.

```
/*
 * Sample submit.cgi function
 */
int submit(HttpState* state) {
    int i;
    if(state->length) {

        /* buffer to write out */

        if(state->offset < state->length) {
            state->offset += sock_fastwrite((sock_type *)&state->s,
            state->buffer + (int)state->offset, (int)state->length-
            (int)state->offset);
        }
        else
        {
            state->offset = 0;
            state->length = 0;
        }
    }
}
```

```

/*
 * Sample submit.cgi function continued
 */
} else {
    switch(state->substate) {
        case 0:
            strcpy(state->buffer, "HTTP/1.0 200 OK\r\n");
            break;
        case 1:
            /* init the FORMSpec data */
            FORMSpec[0].value[0] = '\0';
            FORMSpec[1].value[0] = '\0';
            state->p = state->buffer;
            parse_post(state);
            state->substate++;
            return 0;
        case 2:
            http_setcookie(state->buffer, FORMSpec[0].value);
            break;
        case 3:
            strcpy(state->buffer, "\r\n\r\n<html><head>
                <title>Results</title></head><body>\r\n");
            break;
        case 4:
            sprintf(state->buffer, "<p>Username:
                %s<p>\r\n<p>Email:
                %s<p>\r\n",
                FORMSpec[0].value, FORMSpec[1].value);
            break;
        case 5:
            strcpy(state->buffer, "<p>Go <a href=\"/\>home</a>
                </body></html>\r\n");
            break;
        default:
            state->substate = 0;
            return 1;
    }
    state->length = strlen(state->buffer);
    state->offset = 0;
    state->substate++;
}
return 0;
}

```

### 4.3.4 HTML Forms Using Zserver.lib

In this section, we will step through a complete example program that uses HTML forms. Through this step-by-step explanation, the method of using the functions in **ZSERVER.LIB** will become clearer.

These lines are part of the standard TCP/IP configuration. You must change them to whatever your local IP address and netmask are. Contact your network administrator for these numbers.

```
#define MY_IP_ADDRESS    "10.10.6.112"  
#define MY_NETMASK      "255.255.255.0"
```

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 which 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 **http\_flashspec** array, then we can define the following macro, which removes some code for handling this array from the web server.

```
#define HTTP_NO_FLASHSPEC
```

These lines are part of the standard TCP/IP configuration.

```
#memmap xmem  
#use "dcrtcp.lib"  
#use "http.lib"  
const HttpType http_types[] =  
{  
    { ".html", "text/html", NULL}  
};
```

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 pull-down 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 TCP/IP servers' object list. The first parameter gives the name of the form. Hence, 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 TCP/IP servers' object list.

```
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 TCP/IP servers' object list. It must be added to the TCP/IP servers' object list 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 index of the variable in the TCP/IP servers' object list.

```
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 TCP/IP servers' object list ("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.

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.

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.

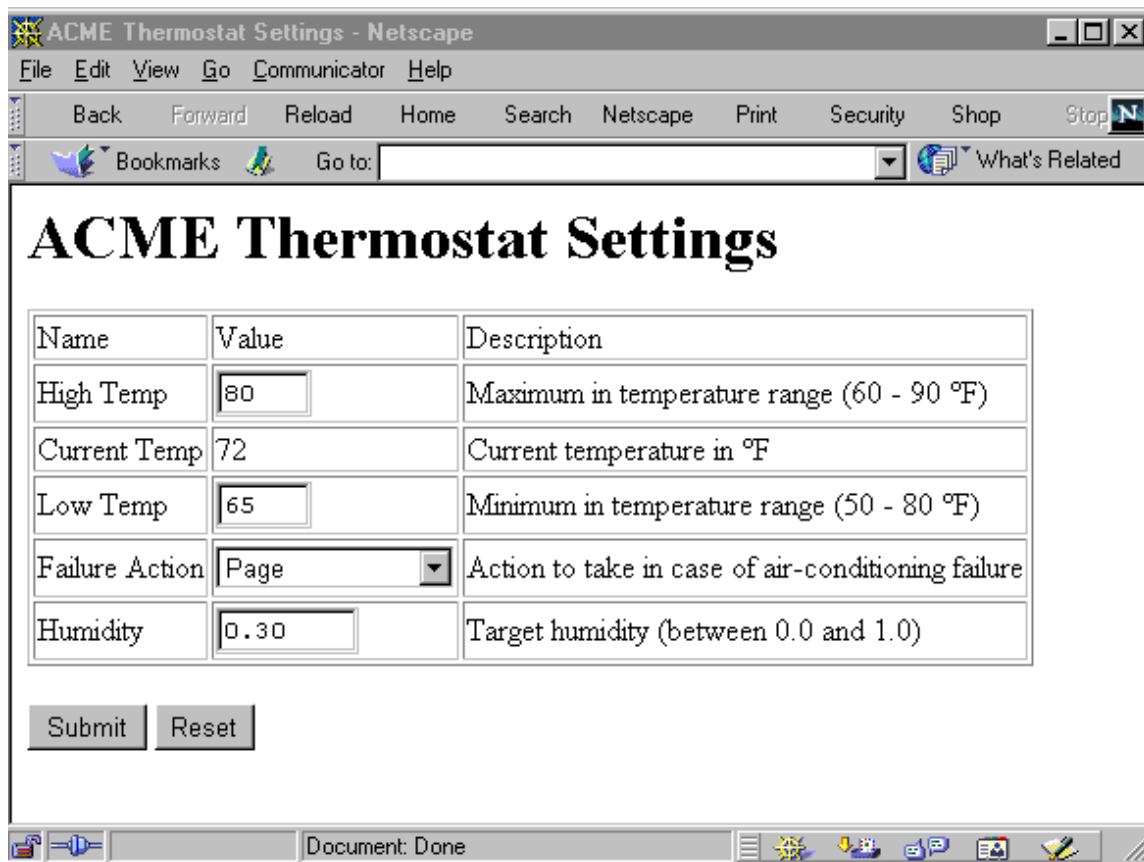
Finally, this code sets up the last variable. Note that it is a float, so **FLOAT32** is given in the **sspec\_addvariable()** call. The last function call is **sspec\_setfvfloatrange()** instead of **sspec\_setfvrange()**, since this is a floating point variable.

These calls create aliases in the TCP/IP servers' object list for

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 address bar is empty, and the page content displays the following form:

## ACME Thermostat Settings

| 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)              |

Below the table are two buttons:  and .

## 4.4 Functions

### `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.

#### PARAMETERS

|                    |   |
|--------------------|---|
| <code>state</code> | Current server struct, as received by the CGI function. |
| <code>url</code>   | 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

|              |   |
|--------------|---|
| <b>state</b> | Current server struct, as received by the CGI function. |
| <b>str</b>   | 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\_addfile

```
int http_addfile(char* name, long location);
```

### DESCRIPTION

Adds a file to the TCP/IP servers list.

### PARAMETERS

|                 |   |
|-----------------|---|
| <b>name</b>     | Name of the file (e.g., “/index.html”).                 |
| <b>location</b> | Address of the file data. (from <code>#ximport</code> ) |

### 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 `&#64;` (decimal) for at-sign) where appropriate. Source string is **NULL** byte terminated. Destination buffer is bounded by a max string length. This function is reentrant.

### PARAMETERS

|             |  |
|-------------|--|
| <b>dest</b> | Buffer where encoded string is stored.       |
| <b>src</b>  | Buffer holding original string (not changed) |
| <b>len</b>  | 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\_delfile

```
int http_delfile(char* name);
```

### DESCRIPTION

Deletes a file from TCP/IP servers' object list.

### 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\_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

|              |   |
|--------------|---|
| <b>start</b> | Pointer to the variable in the buffer for which we want to get information.               |
| <b>name</b>  | Return location for the name of the variable.   |
| <b>value</b> | Return location for the value of the variable.  |
| <b>error</b> | Return location for whether or not the variable is in error (0 if it is not, 1 if it is). |
| <b>next</b>  | Return location for a pointer to the variable after this one.                             |

### LIBRARY

HTTP.LIB

## http\_handler

```
void http_handler();
```

### 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\_init

```
int http_init(void);
```

### DESCRIPTION

Initializes the HTTP daemon.

### RETURN VALUE

0: Success.

### LIBRARY

HTTP.LIB

### SEE ALSO

http\_handler

## 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

|              |  |
|--------------|--|
| <b>form</b>  | <b>server_spec</b> index of the form (i.e., location in TCP/IP servers' object list) |
| <b>state</b> | The HTTP server with which to parse the POSTed data.                                 |

### RETURN VALUE

0 if there is more processing to do;  
1 form processing has been completed.

### LIBRARY

HTTP.LIB

## 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**. This will not be written out to the client, and it is still the responsibility of the client to write out. Also, this utility will generate an HTTP header line that must be written along with any other headers that are written before the HTML file itself is written out. 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 **state->cookie[0]** will equal **'\0'** if no cookie was available.

### PARAMETERS

|              |  |
|--------------|--|
| <b>buf</b>   | Buffer to store cookie-generation header.  |
| <b>value</b> | Text to store in cookie-generation header. |

### LIBRARY

HTTP.LIB

## http\_urldecode

```
char *http_urldecode(char *dest, const char *src, int len);
```

### DESCRIPTION

Converts a string with HTTP transfer-coding ``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

|             |  |
|-------------|--|
| <b>dest</b> | Buffer where decoded string is stored.                                   |
| <b>src</b>  | Buffer holding original string (not changed).                            |
| <b>len</b>  | Maximum size of string ( <b>NULL</b> terminated strings can be shorter). |

### RETURN VALUE

**dest**: If all conversion was good.  
**NULL**: If some conversion had troubles.

### LIBRARY

HTTP.LIB

### SEE ALSO

http\_contentencode

## shtml\_addfunction

```
int shtml_addfunction(char* name, void (*fptr()));
```

### DESCRIPTION

Adds a CGI/SSI-exec function for making dynamic web pages to the TCP/IP servers' object list.

### PARAMETERS

|             |   |
|-------------|---|
| <b>name</b> | Name of the function (e.g., <code>"/foo.cgi"</code> ).  |
| <b>fptr</b> | 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 the `shtml_handler`.

### PARAMETERS

|                 |  |
|-----------------|--|
| <b>name</b>     | Name of the variable.  |
| <b>variable</b> | Pointer to the variable.   |
| <b>type</b>     | Type of variable. The following types are supported: <b>INT8</b> , <b>INT16</b> , <b>INT32</b> , <b>PTR16</b> , <b>FLOAT32</b> |
| <b>format</b>   | 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 TCP/IP servers' object list.

### **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 TCP/IP servers' object list.

### **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

# FTP CLIENT 5

The library **FTP\_CLIENT.LIB** implements the File Transfer Protocol (FTP) for the client side of the connection.

## 5.1 Configuration Macros

### **DTP\_PORT**

The port to listen on for data connections. The low byte of the port number must be 0, as we use the next 256 ports above the one supplied. The default is 0xA00.

### **FTP\_MODE\_DOWNLOAD**

Specifies downloading a file.

### **FTP\_MODE\_UPLOAD**

Specifies uploading a file.

### **MAX\_NAMELEN**

Maximum length for all usernames, passwords, and filenames. The default is 64. Note that this must contain the **NULL** byte, so if it is set to 64, the maximum filename length is 63 characters.

## 5.2 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.

#### PARAMETERS

|                       |  |
|-----------------------|--|
| <code>host</code>     | Host IP address of FTP server.   |
| <code>port</code>     | Port of FTP server, 0 for default.   |
| <code>username</code> | Username of account on FTP server.   |
| <code>password</code> | Password of account on FTP server.   |
| <code>mode</code>     | Mode of transfer ( <code>FTP_MODE_UPLOAD</code> or <code>FTP_MODE_DOWNLOAD</code> ). |
| <code>filename</code> | Filename to get/put.   |
| <code>dir</code>      | Directory file is in, <code>NULL</code> for default directory.                       |
| <code>buffer</code>   | Buffer to get/put the file from/to.  |
| <code>length</code>   | On upload, length of file; on download size of buffer.                               |

#### RETURN VALUE

0: Success;  
1: Failure.

#### LIBRARY

`FTP_CLIENT.LIB`



## **ftp\_client\_tick**

```
int ftp_client_tick(void);
```

### **DESCRIPTION**

Tick function to run the FTP daemon. Must be called periodically.

### **RETURN VALUE**

- 0: Still pending, call again;
- 1: Success (file transfer complete);
- 2: Failure (general);
- 3: Failure (Couldn't connect to remote host);
- 4: Failure (File not found).

### **LIBRARY**

FTP\_CLIENT.LIB

## **ftp\_client\_filesize**

```
int ftp_client_filesize(void);
```

### **DESCRIPTION**

If a file was downloaded (mode == **FTP\_MODE\_DOWNLOAD**), when **ftp\_client\_tick()** returns 1, this function will return the size of the fetched file. This number will be clobbered if **ftp\_client\_setup()** is called again, so it should be copied out and stored quickly!

### **RETURN VALUE**

Size, in bytes.

### **LIBRARY**

FTP\_CLIENT.LIB

## 5.3 Sample FTP Transfer

```
#define MY_IP_ADDRESS "10.10.6.105"
#define MY_NETMASK "255.255.255.0"

#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 download
    of the file "bar" is selected to 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 the size of the transfer,
        senses we requested a download.*/

        buf[ftp_client_filesize()] = '\0';
        printf("Data => '%s'\n", buf);
    } else {
        printf("FTP failed: status == %d\n",ret);
    }
}
```

# FTP Server 6

The library `FTP_SERVER.LIB` implements the File Transfer Protocol for the server side of the 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. Anonymous login is supported.

## 6.1 Configuration Constants

### **FTP\_MAXSERVERS**

This is the number of simultaneous connections the FTP server can support. It is recommended that this be set to one (the default), as each subsequent server requires a significant amount of RAM (2500 bytes by default; this can change through `SOCK_BUF_SIZE` or `tcp_MaxBufSize` (deprecated)).

### **FTP\_MAXNAME**

The maximum length of filenames, usernames, and passwords. (It must include a null character so, with its default value of 20, filenames can be 19 characters long.)

### **FTP\_MAXLINE**

The size of the working buffer in each server. Also, this is the maximum size of each network read/write. It needs to be a minimum of about 256 bytes for the server to function properly. You probably don't need to change its default of 1024 bytes.

### **FTP\_TIMEOUT**

The length of time to wait for data from the remote host, before terminating the connection. If you have a high-latency network condition, this might need to be increased from its default of 16 seconds to avoid premature closures.

### 6.1.1 File Options

```
#define O_UNUSED 0
#define O_RDONLY 1
#define O_WRONLY 2
#define O_RDWR 3
```

## 6.2 File Handlers

The data structure **FTPHandlers** can be passed to **ftp\_init** to redefine how files are read and written to. It contains function pointers to all of the individual functions. The default functions are listed below.

```
typedef struct {
    int (*open)();
    int (*read)();
    int (*write)();
    int (*close)();
    int (*getfilesize)();
} FTPHandlers;
```

### open

```
int open(char *name, int options, int uid);
```

#### DESCRIPTION

Opens a file.

#### PARAMETERS

|                |   |
|----------------|---|
| <b>name</b>    | The file to open,   |
| <b>options</b> | For a read-only file the value is <b>O_RDONLY</b> ; for a write-only file, the value is <b>O_WRONLY</b> |
| <b>uid</b>     | The userid of the currently logged in user.   |

#### RETURN VALUE

A file descriptor should be returned, or **-1** on error.

## getfilesize

```
int getfilesize(int fd);
```

### DESCRIPTION

If a file was opened for reading (`O_RDONLY`), this should return the size of the file.

### PARAMETERS

**fd**                    The file descriptor that was returned when the file was opened.

### RETURN VALUE

The size of the file in bytes.

## read

```
int read(int fd, char *buf, int len);
```

### DESCRIPTION

Reads a buffer of length **len** from **fd** into **buf**.

### PARAMETERS

**fd**                    The file descriptor returned from `open( )`.

**buf**                   The location to read the file into.

**len**                   The number of bytes to read.

### RETURN VALUE

The number of bytes read.

## write

```
int write(int fd, char *buf, int len);
```

### DESCRIPTION

Writes a buffer of length **len** from **buf** to **fd**. This is not currently supported.

### PARAMETERS

|            |   |
|------------|---|
| <b>fd</b>  | The file descriptor returned from <b>open ( )</b> . This is destination the data will be written to |
| <b>buf</b> | The source location of the data to be written   |
| <b>len</b> | The number of bytes to write.   |

### RETURN VALUE

Number of bytes written.

## close

```
int close(int fd);
```

### DESCRIPTION

Closes the file, and invalidates the file descriptor.

### PARAMETERS

|           |  |
|-----------|--|
| <b>fd</b> | The file descriptor (returned from <b>open ( )</b> ) of the file to close. |
|-----------|--|

### RETURN VALUE

0

**Please note that if you redefine any of these file handler functions, all must be replaced.**

## 6.3 Functions

### `ftp_init`

```
void ftp_init(FTPHandlers *handlers);
```

#### DESCRIPTION

Initializes the FTP daemon.

#### PARAMETERS

**handlers**      **NULL** means use default internal file handlers;  
                  **!NULL** means to supply a struct of pointers to the various custom  
                  file handlers (open, read, write, close, getfilesize).

#### RETURN VALUE

None.

#### LIBRARY

FTP\_SERVER.LIB

### `ftp_tick`

```
void ftp_tick(void);
```

#### DESCRIPTION

Once **ftp\_init** has been called, **ftp\_tick** must be called periodically to run the daemon. This function is non-blocking.

#### LIBRARY

FTP\_SERVER.LIB

## 6.4 Sample FTP Server

This code demonstrates a simple FTP server. The user "anonymous" may download the file "rabbitA.gif", but not "rabbitF.gif". The user "foo" (with password "bar") may download "rabbitF.gif", but not "rabbitA.gif".

```
/* ftp_server.c */

#define MY_IP_ADDRESS "10.10.6.105"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.19"

#include <xmem>
#include "dcrtcp.lib"
#include "ftp_server.lib"

#include "samples/tcpip/http/pages/rabbit1.gif" rabbit1_gif

main() {
    int file;
    int user;
    // Set up the first file and user
    file = sspec_addxmemfile("rabbitA.gif", rabbit1_gif, SERVER_FTP);
    user = sauth_adduser("anonymous", "", SERVER_FTP);
    sspec_setuser(file, 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);

    sock_init();
    ftp_init(NULL); /* use default handlers */
    tcp_reserveport(21);
    while(1) {
        ftp_tick();
    }
}
```

The program `SSTATIC2.C` in `SAMPLES\TCPIP\HTML` provides a more advanced example than the one shown here.



# TFTP Client 7

**TFTP.LIB** implements the Trivial File Transfer Protocol (TFTP). This standard protocol (internet RFC783) is a lightweight protocol typically used

## 7.0.2 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;          // misc 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.
};
```

### Macros 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)
```

## 7.0.3 Function Reference

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 `xmem` (see `xalloc()`).

### 7.0.3.1 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

```
int tftp_tickx(long ts_addr);
```

**DESCRIPTION**

This function is a

## 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

|                 |  |
|-----------------|--|
| <b>put</b>      | 0: get file from remote host; 1: put file to host.   |
| <b>buf_addr</b> | Physical address of data buffer.   |
| <b>len</b>      | 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). |
| <b>mode</b>     | Data representation: 0=NETASCII, 1=OCTET (binary), 2=MAIL.   |
| <b>host</b>     | Remote host name, or <b>NULL</b> to use default BOOTP host.  |
| <b>hostfile</b> | Name of file on remote host, or e-mail address for mail.   |
| <b>sock</b>     | UDP socket to use, or <b>NULL</b> 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





# SMTP Mail Client 8

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.

Sending mail with the `SMTP.LIB` client library is a four-step process. First, build your e-mail message, then call `smtp_sendmail()`. Next, repetitively call `smtp_maintick()` while it is returning `SMTP_PENDING`. Finally, call `smtp_status()` to determine if the mail was sent successfully. There is a sample program in Section 8.4 that outlines how to send a simple mail message.

## 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 Configuration

The SMTP client is configured by using compiler macros.

### **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\_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
```

## 8.3 Functions

### `smtp_sendmail`

```
void smtp_sendmail(char* to, char* from, char* subject, char*
    message);
```

#### DESCRIPTION

This function initializes the internal data structures with strings for the to e-mail address, the from e-mail address, the subject, and the body of the message. You should not modify these strings until `smtp_maintick` no longer returns `SMTP_PENDING`.

#### PARAMETERS

|                      |   |
|----------------------|---|
| <code>to</code>      | String containing the e-mail address of the destination.  |
| <code>from</code>    | String containing the e-mail address of the source.   |
| <code>subject</code> | String containing the subject of the message.   |
| <code>message</code> | String containing the message. (This string 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.) |

#### RETURN VALUE

None.

#### LIBRARY

`SMTP.LIB`

## smtp\_sendmailxmem

```
void smtp_sendmailxmem(char* to, char* from, char* subject,  
    long message, int messagelen);
```

### DESCRIPTION

This function initializes the internal data structures with strings for the to e-mail address, the from e-mail address, the subject, and the body of the message. You should not modify these strings until `smtp_maintick` no longer returns `SMTP_PENDING`

### PARAMETERS

|                         |   |
|-------------------------|---|
| <code>to</code>         | String containing the e-mail address of the destination.  |
| <code>from</code>       | String containing the e-mail address of the source.   |
| <code>subject</code>    | String containing the subject of the message.   |
| <code>message</code>    | 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.) |
| <code>messagelen</code> | Length of the message in xmem.  |

### RETURN VALUE

None.

### LIBRARY

SMTP.LIB

## smtp\_maintick

```
int smtp_maintick(void);
```

### DESCRIPTION

Repetitively call this function until e-mail is completely sent. For a small message, this function will need to be called about 20 times to send the message. The number of times will vary depending on the latency of your connection to the mail server and the size of your message.

### 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

## 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

## 8.4 Sample Sending of an E-mail

This program, `smtp.c`, uses the SMTP library to send an e-mail.

```
/* Change these macros to the appropriate values or change
 * the smtp_sendmail(...) call in main() to reference your values.
 */

#define FROM      "myaddress@mydomain.com"
#define TO        "myaddress@mydomain.com"
#define SUBJECT   "test mail"
#define BODY      "You've got mail!"

/* Change these values to your network settings */
#define MY_IP_ADDRESS "10.10.6.100"
#define MY_NETMASK    "255.255.255.0"
#define MY_GATEWAY    "10.10.6.19"

/* SMTP_SERVER tells DCRTCP where your mail server is. This
 * value can be the name or the IP address. */

#define SMTP_SERVER "mymailserver.mydomain.com"

//#define SMTP_DOMAIN "mycontroller.mydomain.com"

//#define SMTP_DEBUG

#memmap xmem
#use dcrtcp.lib
#use smtp.lib

main() {
    sock_init();

    smtp_sendmail(FROM, TO, SUBJECT, BODY);

    while(smtp_maintick()==SMTP_PENDING)
        continue;

    if(smtp_status()==SMTP_SUCCESS)
        printf("Message sent\n");
    else
        printf("Error sending message\n");
}
```

# POP3 Client 9

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 Three 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 several times to give you the e-mail.

## 9.3 Call-Back Function

There are two types of call-back functions, depending on if `POP_PARSE_EXTRA` is defined and will be handled separately.

### 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);
```

**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.

**buf** 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**.

See `pop.c` in the Dynamic C **Sample** folder for 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 only be used 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 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

|                 |   |
|-----------------|---|
| <b>username</b> | The <b>username</b> of the account to access.                                   |
| <b>password</b> | The <b>password</b> of the account to access.                                   |
| <b>server</b>   | The IP address of the server to connect to, as returned from <b>resolve()</b> . |

### 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

`parse_extra.c` connects to a POP3 server and downloads e-mail form it.

```
#define MY_IP_ADDRESS "10.10.6.105" // change these configuration macros
#define MY_NETMASK    "255.255.255.0" // to match your host.
#define MY_GATEWAY    "10.10.6.1"
#define MY_NAMESERVER "10.10.6.254"

#define POP_HOST mail.domain.com //enter the name of your POP3 server

#define POP_USER "myname" //enter username for POP3 account
#define POP_PASS "secret" //enter 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 function

    printf("Resolving name...\n");
    address = resolve(POP_HOST);
    printf("Calling pop3_getmail()...\n");
    pop3_getmail(POP_USER, POP_PASS, address); // POP3 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 the server's message, and lines starting with 'C:' are the client's messages.

```
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.

# Telnet 10

The library `vserial.lib` implements the telecommunications network interface, known as telnet. This implementation is a telnet to serial and serial to telnet gateway.

## 10.1 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 (useful if you are actually Telnetting to the device from another box; should probably NOT be defined if you are using two devices as a transparent bridge over the Ethernet).

## 10.2 Functions

### `telnet_init`

```
int telnet_init(int which, longword addy, int port);
```

#### DESCRIPTION

Initializes the connection.

#### PARAMETERS

|              |  |
|--------------|--|
| <b>which</b> | Is one of the following:<br><b>TELNET_LISTEN</b> —Listens on a port for incoming connections.<br><b>TELNET_RECONNECT</b> —Connects to a remote host, and reconnects if the connection die.<br><b>TELNET_CONNECT</b> —Connects to a remote host, and terminates if the connection dies. |
| <b>addy</b>  | IP address of the remote host, or 0 if we are listening.   |
| <b>port</b>  | 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

## 10.3 An Example Telnet Server

```
/*
 * Telnet Server: Listens on a telnet port for a connection, and
 * transparently passes data on to the serial port
 */

// Initialize the IP address/etc as usual
#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

/*
 * We want RAW data, leaving in any telnet/etc control codes.
 * (this is a raw data port). #define this to cook the input.
 */
#undef TELNET_COOKED

#include <xmem.h>
#include <dcrtcp.h>
#include <vserial.h>

/*
 * TCP Port to listen on. 0 defaults to normal telnet port
 */
#define SERVER_PORT 0

main() {
    sock_init(); // Init TCP/IP
    telnet_init(TELNET_LISTEN,0,SERVER_PORT); //Init Vserial server

    // Loop on telenet_tick() to run server; this is non-blocking
    while(!telnet_tick())
        continue;

    // Error happened, close telnet connection (shouldn't happen)
    telnet_close();
}
```

### 10.3.1 A Sample Client To Connect to the Server

```
// Client.c Connects to above server, at given IP address and port

#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"

// Set the speed of the serial port
#define SERIAL_PORT_SPEED 115200
#undef TELNET_COOKED
#mmap xmem
#use "dcrtcp.lib"
#use "vserial.lib"

// TCP Port to connect to. 0 defaults to normal telnet port
#define SERVER_PORT 0

// Remote IP to connect to.
#define REMOTE_HOST "10.10.6.19"

main() {
    sock_init();
    /*
     * Init the VSerial server to connect, and reconnect if the
     * connection is lost
     */
    telnet_init(TELNET_RECONNECT,resolve(REMOTE_HOST),SERVER_PORT);

    // Loop on telenet_tick() to run it; this is non-blocking
    while(!telnet_tick())
        continue;

    // Error happened, we get here - close it (shouldn't happen)
    telnet_close();
}
```



# General Purpose Console 11

## 11.1 Introduction

The library, `zconsole.lib`, implements a serial-based console that can:

- Configure a board.
- Upload and download web pages.
- Change web page variables without re-uploading the page.
- Send e-mail.

## 11.2 Console 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. The Console includes:

- A fail-safe backup system for configuration data.

## 11.3 Console Commands and Messages

The Console 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. The Console 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 the Console whether the command completed successfully or not.

### 11.3.1 Console 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 the Console.

```
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 the Console are listed in Section 11.3.3.1 on page 206.

#### **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 the Console. 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

### 11.3.2 Console 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.

### 11.3.3 Console Commands

The following is a list of the commands provided by `Zconsole.lib`. 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 }
};
```

### 11.3.3.1 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 11.3.3.2 on page 209. 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\_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. The 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. The 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. The command that uses this function takes one parameter: the name of the variable.

### **con\_help()**

This function implements the help system for the Console. The command that uses this function takes one parameter: the name of another command. The Console outputs the associated help text for the requested command. The help text is the text file referenced in the third field of the **ConsoleCommand** structure.

### **con\_list\_files()**

This function lists the files in the filesystem and their file sizes. The command that uses this function takes no parameters.

### **con\_list\_variables()**

This function displays the names and types of all variables. The command that uses this function takes no parameters.

### **con\_mail()**

This function sends mail. If the command that uses this function 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. The 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\_gateway()**

This function changes the gateway of the board. The command that uses this function takes one parameter: the new gateway in dotted quad notation, e.g., 192.168.1.1.

### **con\_set\_ip()**

This function changes the IP address of the board. The 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. The 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. The command that uses this function takes one parameter: the return address.

### **con\_set\_mail\_server**

This functions identifies the SMTP server to use. The command that uses this function takes one parameter: The IP address of the SMTP server.

### **con\_set\_netmask()**

This function changes the netmask of the board. The command that uses this function takes one parameter: the new netmask in dotted quad notation, e.g., 255.255.255.0.

### **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`. The command that uses this function takes no parameters.

### 11.3.3.2 Custom Console Commands

Developers are not limited to the default commands. A custom command is easy to add to the Console; 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; // Read-only: check # of parms in command.
    char cmddata[CON_CMD_DATA_SIZE];
    FileNumber filenum; // Use for file processing.
    File file;          // Use for file processing.
    int spec;           // Use for working with Zserver entities
    long timeout;      // Use for extending the timeout.
} ConsoleState;
```

To accomplish its tasks, the function should use `state->substate` for its state machine (which will be 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.

**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.

The function 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.

### 11.3.4 Console Error Messages

The Console library provides a list of default error messages for the default Console 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
}
```

#### 11.3.4.1 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_SMTPELORR 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_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_SMPTEPERROR, "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." }

```

### 11.3.4.2 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.

## 11.4 Console 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) ();
    int (*close)();
    int (*tick) ();
    int (*puts) ();
    int (*rdUsed) ();
    int (*wrUsed) ();
    int (*wrFree) ();
    int (*read) ();
    int (*write) ();
} ConsoleIO;
```

### 11.4.1 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[]`.

### 11.4.2 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, serAwrUsed, serAwrFree, serAread, serAwrite}
```

#### 11.4.2.1 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.

#### 11.4.2.2 Telnet

The Console 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.

### 11.4.2.3 Slave Port

The Rabbit slave port is an 8-bit bidirectional data port. The Console runs on the slave processor. Two drivers are needed.

#### 11.4.2.3.1 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 Console 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 Console commands for the slave is pro-

### 11.5.1 File System Initialization

The Console depends on the file system that is included with Dynamic C. Besides including the library and defining the macro that directs the file system to EEPROM memory:

```
#define FS_FLASH
#define "FileSystem.lib"
```

the application program must initialize the file system with a call to `fs_init()`.

### 11.5.2 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.

### 11.5.3 Using TCP/IP

To use the TCP/IP functionality of the Console 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 `smtp.lib`.

## 11.5.4 Required Console Functions

To run the Console, the following two functions are required.

### `console_init`

```
int console_init(void);
```

#### DESCRIPTION

This function will initialize the Console.

#### RETURN VALUE

0: Success;

1: Error.

### `console_tick`

```
void console_tick(void);
```

#### DESCRIPTION

This function needs to be called periodically in an application program to allow the Console time for processing.

## 11.5.5 Console Execution Choices

The Console can be used interactively with a terminal emulator or programatically by sending commands from a program running on a device connected to the controller that is running the Console.

### 11.5.5.1 Terminal Emulator

To manually enter Console commands from a keyboard and view results in the stdio window you must:

1. Run Dynamic C, version 7.5 or later.
2. Open a terminal emulator. Windows HyperTerminal comes with Windows. It does not work with binary files, only ASCII. Tera Term, available for free download at

<http://hp.vector.co.jp/authors/VA002416/teraterm.html>

can handle both ASCII and binary. 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 Console 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.

## 11.6 Backup System

The Console can save configuration parameters to the file system so that they are available across power cycles. The backup process is done by `con_backup()`. Unlike the other console 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 console 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.

### 11.6.1 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**

#### **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).

## 11.6.2 Array Definition for Backup System

```
const ConsoleBackup console_backup[] = {
    CONSOLE_BASIC_BACKUP, // echo state, baud-rate/port number
    CONSOLE_TCPIP_BACKUP,
    CONSOLE_HTTP_BACKUP,
    CONSOLE_SMTP_BACKUP
    { 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 netmask, gateway and name server.

**CONSOLE\_HTTP\_BACKUP** causes backup of the files and variables visible to the HTTP server.

**CONSOLE\_SMTP\_BACKUP** causes backup of the mail configuration.

## 11.7 Console Macros

**zconsole.lib** offers many macros that change the behavior of the Console.

### **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\_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\_DATA\_SIZE**

Adjusts the size of the user data area within the state structure so that user commands can preserve arbitrary information across calls. It is allocated in root data space. Defaults to 16 bytes.

### **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\_INIT\_MESSAGE**

Defines the message that is displayed on all Console I/O methods upon startup. Defaults to “Console Ready\r\n”.

### **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\_BACKUP\_FILE1**

The file number used for the first backup file. 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.

### **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. This number must be in the range 128-143. Defaults to 129.

### **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\_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.

## **11.8 Sample Program**

```
/*
*****
tcpipconsole.c
Z-World, 2001
This sample program demonstrates many of the features of ZCONSOLE.LIB.

Among the features this sample program 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.
*****/

#define MY_IP_ADDRESS "10.10.6.112"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"
#define MY_NAMESERVER "10.10.6.1"
#define SMTP_SERVER "10.10.6.1"

/*
 * 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 (see above), the mail client uses one
 * socket, and the telnet interface uses 1 socket.
 */
#define MAX_SOCKETS 4

/*
 * All web server content is dynamic, so we do not need
 * http_flashspec[].
 */
#define HTTP_NO_FLASHSPEC

/*
 * The filesystem that the console uses should be located in flash.
 */
#define FS_FLASH

/*
 * Console 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

```

```

/*
 * These ximport directives include the help texts for the
 * consolecommands. Having the help text in xmem helps save
 * root code space.
 */
#ximport "samples\zconsole\tcpipconsole_help\help.txt" help_txt
#ximport "samples\zconsole\tcpipconsole_help\help_help.txt"
    help_help_txt
#ximport "samples\zconsole\tcpipconsole_help\help_echo.txt"
    help_echo_txt
#ximport "samples\zconsole\tcpipconsole_help\help_set.txt"
    help_set_txt
#ximport "samples\zconsole\tcpipconsole_help\help_set_param.txt"
    help_set_param_txt
#ximport "samples\zconsole\tcpipconsole_help\help_set_mail.txt"
    help_set_mail_txt
#ximport
    "samples\zconsole\tcpipconsole_help\help_set_mail_server.txt"
    help_set_mail_server_txt
#ximport "samples\zconsole\tcpipconsole_help\help_set_mail_from.txt"
    help_set_mail_from_txt
#ximport "samples\zconsole\tcpipconsole_help\help_show.txt"
    help_show_txt
#ximport "samples\zconsole\tcpipconsole_help\help_put.txt"
    help_put_txt
#ximport "samples\zconsole\tcpipconsole_help\help_get.txt"
    help_get_txt
#ximport "samples\zconsole\tcpipconsole_help\help_delete.txt"
    help_delete_txt
#ximport "samples\zconsole\tcpipconsole_help\help_list.txt"
    help_list_txt
#ximport "samples\zconsole\tcpipconsole_help\help_createv.txt"
    help_createv_txt
#ximport "samples\zconsole\tcpipconsole_help\help_putv.txt"
    help_putv_txt
#ximport "samples\zconsole\tcpipconsole_help\help_getv.txt"
    help_getv_txt
#ximport "samples\zconsole\tcpipconsole_help\help_mail.txt"
    help_mail_txt
#ximport "samples\zconsole\tcpipconsole_help\help_reset.txt"
    help_reset_txt
#ximport "samples\zconsole\tcpipconsole_help\help_reset_files.txt"
    help_reset_files_txt
#ximport
    "samples\zconsole\tcpipconsole_help\help_reset_variables.txt"
    help_reset_variables_txt

```

```

#mmap xmem

#use "FileSystem.lib"
#use "dcrtcp.lib"
#use "http.lib"
#use "smtp.lib"

/*
 * Note that all libraries that zconsole.lib needs must be #use'd
 * before #use'ing zconsole.lib .
 */
#use "zconsole.lib"

/*
 * This function prototype is for a custom command, so it must be
 * declared before the console_command[] array.
 */
int hello_world(ConsoleState* state);

/*
 * This array defines which I/O streams for which the console will
 * be available. The streams included below are defined through
 * macros. Available macros are CONSOLE_IO_SERA, CONSOLE_IO_SERB,
 * CONSOLE_IO_SERC, CONSOLE_IO_SERD, CONSOLE_IO_TELNET, and
 * CONSOLE_IO_SP (for the slave port). The parameter for the macro
 * represents the initial baud rate for serial ports, the port
 * number for telnet, or the channel number for the slave port.
 * It is possible for the user to define her own I/O handlers and
 * include them in a ConsoleIO structure in the console_io array.
 * Remember that if you change the number of I/O streams here, you
 * should also change the NUM_CONSOLES macro 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--that is, the means by which 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 },
    { "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", NULL, help_reset_txt },
    { "RESET FILES", con_reset_files, help_reset_files_txt },
    { "RESET VARIABLES", con_reset_variables,
      help_reset_variables_txt }
};

```

```

/*
 * This array sets up the error messages that can be generated.
 * CON_STANDARD_ERRORS is a macro that expands to the standard
 * errors that the built-in commands in zconsole.lib uses. 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 filesystem. Note that if, for example, the
 * HTTP or SMTP related commands are include 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 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}
};
/*
 * This is a custom command. C5Ea ",c2(his)-614lra(H)12(ma)12(cro)12(ake(his)-61

```

```

void main(void)
{
    /*
     * All initialization of TCP/IP, clients, servers, and I/O
     * must be done by the user prior to using any console functions.
     */
    sock_init();
    tcp_reserveport(80);    // Enable SYN-queueing and disable the
                          // 2MSL wait for the web server (results
                          // in performance improvements).

    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);
        /*
         * Anytime after the file system has been initialized or
         * formatted (after console_init() has been executed),
         * con_backup_reserve() must be called to reserve space 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();
    }
}

```

# Index

## Numerics

2MSL .....81

## A

### Application Protocols

FTP Client .....169  
FTP Server .....173  
HTTP .....133  
POP3 Client .....193  
SMTP Client .....187  
Telnet .....199  
TFTP .....179

## B

Buffer sizes .....10

## C

Checksums .....51  
Console .....203  
  Backup System .....216  
  circular buffers .....214  
  Commands .....204  
    action taken .....204  
    command array .....205  
    custom commands .....209  
    data structure .....204  
    default commands .....205  
    default functions .....206  
    help overview .....204  
    help text for command .....204  
    name of command .....204  
  configuration macros .....217  
  Console Execution .....213  
    slave port .....213  
    Telnet .....212  
    terminal emulator .....215  
  Daemon .....215  
  Error Messages .....210  
    custom error messages .....211  
    default error messages .....210  
  file system initialization .....214  
  I/O Interface .....212  
    custom I/O methods .....213  
    including an I/O method ....  
    212  
    multiple I/O streams .....213  
    predefined I/O methods .....212  
  Initialization .....215  
  physical connection .....213  
  required functions .....215  
  sample program .....218

using TCP/IP .....214

## D

### Daemons

ftp\_client\_tick .....171  
ftp\_tick .....177  
http\_handler .....162  
pop3\_tick .....196  
tcp\_tick .....82  
telnet\_tick .....200  
tftp\_tick .....183

## E

### E-mail

POP3 Client  
  call-back function .....194  
  configuration .....193  
  receiving e-mail .....193  
  sample conversation .....198  
  sample program .....197  
SMTP Client  
  configuration .....188  
  debug .....188  
  define server .....188  
  HELO command .....188  
  sample conversation .....187  
  sample program .....192  
  sending e-mail .....187  
  timeout value .....188  
Ethernet Transmission Unit ...46

## F

FTP Client .....169  
  download file .....169  
  FTP daemon .....171  
  port number .....169  
  set up file transfer .....170  
  size of downloaded file ....171  
  upload files .....169  
FTP Server .....173  
  anonymous login .....173  
  Configuration Constants ..173  
    buffer size .....173  
    connection timeout .....173  
    simultaneous connections ..  
    173  
    string lengths .....173  
  file handlers .....174  
  sample program .....178  
Function Reference  
  Addressing  
    arp\_resolve .....21  
    getdomainname .....23

gethostid .....24  
gethostname .....24  
getpeername .....25  
getsockname .....26  
pd\_getaddress .....33  
psocket .....34  
resolve .....35  
setdomainname .....38  
sethostid .....39  
sethostname .....39  
CGI  
  cgi\_redirectto .....158  
  cgi\_sendstring .....159  
Configuration  
  tcp\_config .....75  
Console  
  console\_init .....215  
  console\_tick .....215  
Cookie  
  http\_setcookie .....164  
Data Conversion  
  htonl .....27  
  htons .....27  
  http\_contentencode .....160  
  http\_urldecode .....165  
  inet\_addr .....28  
  inet\_ntoa .....29  
  ntohl .....31  
  ntohs .....32  
  paddr .....32  
  rip .....36  
E-mail  
  pop3\_getmail .....196  
  pop3\_init .....195  
  pop3\_tick .....196  
  smtp\_mailltick .....191  
  smtp\_sendmail .....189  
  smtp\_sendmailxmem ...190  
  smtp\_status .....191  
FTP Client  
  ftp\_client\_filesize .....171  
  ftp\_client\_setup .....170  
  ftp\_client\_tick .....171  
FTP Server  
  ftp\_init .....177  
  ftp\_tick .....177  
HTML Forms  
  http\_finderrbuf .....161  
  http\_nextfverr .....162  
  http\_parseform .....163  
  sspec\_addfv .....96  
  sspec\_findfv .....102  
  sspec\_getformtitle .....105  
  sspec\_getfvdesc .....107

|                           |     |                             |     |                              |          |
|---------------------------|-----|-----------------------------|-----|------------------------------|----------|
| sspec_getfventrytype ...  | 108 | sock_tbsize .....           | 68  | sauth_authenticate .....     | 91       |
| sspec_getfvlen .....      | 108 | sock_tbusd .....            | 68  | sauth_getusername .....      | 92       |
| sspec_getfvname .....     | 109 | Socket Status               |     | sauth_getwriteaccess .....   | 92       |
| sspec_getfvnum .....      | 109 | sock_bytesready .....       | 41  | sauth_setwriteaccess .....   | 93       |
| sspec_getfvopt .....      | 110 | sock_dataready .....        | 42  | TFTP Client                  |          |
| sspec_getfvoptlistlen ... | 110 | sockerr .....               | 43  | tftp_exec .....              | 185      |
| sspec_getfvreadonly ....  | 111 | sockstate .....             | 66  | tftp_init .....              | 181      |
| sspec_setformepilog ....  | 120 | tcp_tick .....              | 82  | tftp_initx .....             | 182      |
| sspec_setformfunction .   | 121 | TCP/IP Engine               |     | tftp_tick .....              | 183      |
| sspec_setformprolog ....  | 122 | sock_init .....             | 50  | tftp_tickx .....             | 184      |
| sspec_setformtitle .....  | 123 | tcp_tick .....              | 82  | Timers                       |          |
| sspec_setfvcheck .....    | 124 | TCP/IP servers' list        |     | ip_timer_expired .....       | 29       |
| sspec_setfvdesc .....     | 125 | http_delfile .....          | 161 | ip_timer_init .....          | 31       |
| sspec_setfventrytype ...  | 125 | TCP/IP servers' object list |     | UDP Socket I/O               |          |
| sspec_setfvfloatrange ..  | 126 | http_addfile .....          | 159 | sock_recv .....              | 62       |
| sspec_setfvlen .....      | 126 | shtml_addfunction .....     | 166 | sock_recv_from .....         | 64       |
| sspec_setfvname .....     | 127 | shtml_addvariable .....     | 167 | sock_recv_init .....         | 65       |
| sspec_setfvoptlist .....  | 127 | shtml_delfunction .....     | 168 | udp_open .....               | 83       |
| sspec_setfvrange .....    | 128 | shtml_delvariable .....     | 168 |                              |          |
| sspec_setfvreadonly ....  | 128 | sspec_addform .....         | 94  | <b>H</b>                     |          |
| HTTP server               |     | sspec_adddfsfile .....      | 95  | HTML Forms .....             | 145      |
| http_handler .....        | 162 | sspec_addfunction .....     | 96  | buffer allocation .....      | 152      |
| http_init .....           | 163 | sspec_addrootfile .....     | 97  | FORM tag .....               | 145      |
| Ping                      |     | sspec_addvariable .....     | 98  | ACTION option .....          | 145      |
| _chk_ping .....           | 22  | sspec_addxmemfile .....     | 99  | METHOD option .....          | 145      |
| _ping .....               | 34  | sspec_addxmemvar ....       | 100 | INPUT tag .....              | 145      |
| _send_ping .....          | 37  | sspec_aliasspec .....       | 101 | NAME parameter .....         | 145      |
| Socket Configuration      |     | sspec_checkaccess .....     | 102 | SIZE parameter .....         | 145      |
| sock_mode .....           | 51  | sspec_findname .....        | 103 | TYPE parameter .....         | 145      |
| tcp_clearreserve .....    | 74  | sspec_findnextfile .....    | 104 | VALUE parameter .....        | 145      |
| tcp_reserveport .....     | 81  | sspec_getfileloc .....      | 104 | option list .....            | 156      |
| Socket Connection         |     | sspec_getfiletype .....     | 105 | POST-style submission ...    | 148      |
| sock_abort .....          | 40  | sspec_getfunction .....     | 106 | pulldown menu .....          | 153      |
| sock_close .....          | 42  | sspec_getfvspec .....       | 111 | sample page .....            | 146      |
| sock_established .....    | 44  | sspec_getlength .....       | 112 | Zserver.lib functionality .. | 152      |
| Socket I/O                |     | sspec_getname .....         | 112 | HTTP Servers .....           | 133      |
| sock_fastread .....       | 45  | sspec_getrealm .....        | 113 | authentication .....         | 134      |
| sock_fastwrite .....      | 46  | sspec_gettype .....         | 113 | CGI .....                    | 145      |
| sock_flush .....          | 47  | sspec_getusername .....     | 114 | sample handler .....         | 150      |
| sock_flushnex .....       | 48  | sspec_getvaraddr .....      | 114 | configurable constants ....  | 137      |
| sock_getc .....           | 49  | sspec_getvarkind .....      | 115 | Data Structures .....        | 133      |
| sock_gets .....           | 50  | sspec_getvartype .....      | 115 | HttpRealm .....              | 134      |
| sock_preread .....        | 52  | sspec_needsauthentication . | 116 | HttpSpec .....               | 133      |
| sock_putc .....           | 53  | sspec_readfile .....        | 117 | HttpState .....              | 135      |
| sock_puts .....           | 54  | sspec_readvariable .....    | 118 | HttpType .....               | 134      |
| sock_read .....           | 56  | sspec_remove .....          | 118 | dynamic web pages .....      | 141      |
| sock_write .....          | 73  | sspec_restore .....         | 119 | file extensions .....        | 134, 140 |
| tcp_listen .....          | 77  | sspec_save .....            | 119 | HTML Forms .....             | 145      |
| tcp_open .....            | 79  | sspec_setrealm .....        | 129 | MIME type .....              | 134      |
| Socket I/O Buffer         |     | sspec_setrealm .....        | 129 | number of servers .....      | 137      |
| sock_rbleft .....         | 54  | sspec_setsavedata .....     | 130 | POST command .....           | 148      |
| sock_rbsize .....         | 55  | sspec_setuser .....         | 131 | protection spaces .....      | 134      |
| sock_rbusd .....          | 55  | TCP/IP users list           |     | SSI .....                    | 144      |
| sock_tbleft .....         | 67  | sauth_adduser .....         | 90  | static web pages .....       | 138      |



|                               |          |  |  |
|-------------------------------|----------|--|--|
| URL-encoded Data .....        | 148      |  |  |
| Reading & Storing .....       | 149      |  |  |
| <b>I</b>                      |          |  |  |
| IP Addresses                  |          |  |  |
| lease .....                   | 4, 5     |  |  |
| Set Dynamically .....         | 3        |  |  |
| Set Manually .....            | 3        |  |  |
| <b>M</b>                      |          |  |  |
| Maximum Segment Size .....    | 10       |  |  |
| memmap .....                  | 15       |  |  |
| MIME types .....              | 140      |  |  |
| <b>N</b>                      |          |  |  |
| Nagle algorithm .....         | 51       |  |  |
| <b>P</b>                      |          |  |  |
| Packet Processing .....       | 16       |  |  |
| POP3 Client                   |          |  |  |
| Configuration .....           | 193      |  |  |
| debug option .....            | 193      |  |  |
| receiving e-mail .....        | 193      |  |  |
| <b>R</b>                      |          |  |  |
| Reset clock .....             | 138      |  |  |
| <b>S</b>                      |          |  |  |
| Server Utility Library .....  | 87       |  |  |
| configurable constants .....  | 88       |  |  |
| Data Structures .....         | 87       |  |  |
| access .....                  | 88       |  |  |
| TCP/IP servers' object list . |          |  |  |
| 87                            |          |  |  |
| TCP/IP users list .....       | 87       |  |  |
| number of objects .....       | 89       |  |  |
| number of users .....         | 89       |  |  |
| object types .....            | 88       |  |  |
| variable types .....          | 88       |  |  |
| SMTP Client                   |          |  |  |
| Configuration .....           | 188      |  |  |
| debug option .....            | 188      |  |  |
| define mail server .....      | 188      |  |  |
| HELO command .....            | 188      |  |  |
| timeout value .....           | 188      |  |  |
| sending e-mail .....          | 187      |  |  |
| Socket                        |          |  |  |
| data structure .....          | 10       |  |  |
| default mode .....            | 13       |  |  |
| definition .....              | 10       |  |  |
| empty line vs empty buffer    | 41       |  |  |
| <b>T</b>                      |          |  |  |
| TCP Socket .....              | 10       |  |  |
| Active Open .....             | 11       |  |  |
| Blocking Macros .....         | 17       |  |  |
| Control Functions .....       | 11       |  |  |
| Delay a Connection .....      | 11       |  |  |
| I/O Functions .....           | 13       |  |  |
| Blocking .....                | 17       |  |  |
| Non-Blocking .....            | 16       |  |  |
| Passive Open .....            | 10       |  |  |
| TCP/IP .....                  | 3        |  |  |
| Configuration .....           | 3        |  |  |
| BOOTP/DHCP .....              | 3        |  |  |
| I/O Buffers .....             | 10       |  |  |
| IP Addresses .....            | 3        |  |  |
| MAC address .....             | 3        |  |  |
| Skeleton Program .....        | 15       |  |  |
| Initialization .....          | 15       |  |  |
| Multitasking .....            | 18       |  |  |
| TFTP Client .....             | 179      |  |  |
| Data Structure .....          | 180      |  |  |
| DHCP/BOOTP .....              | 179      |  |  |
| stack space .....             | 180      |  |  |
| Tick rates .....              | 16       |  |  |
| <b>U</b>                      |          |  |  |
| UDP                           |          |  |  |
| Broadcast Packets .....       | 14       |  |  |
| Performance .....             | 15       |  |  |
| UDP Socket                    |          |  |  |
| Checksum .....                | 15       |  |  |
| Functions .....               | 13       |  |  |
| Open and Close .....          | 14       |  |  |
| Read .....                    | 14       |  |  |
| record service .....          | 65       |  |  |
| Write .....                   | 14       |  |  |
| URL-encoded Data .....        | 148, 149 |  |  |
| <b>W</b>                      |          |  |  |
| Well-known Ports              |          |  |  |
| FTP server .....              | 173      |  |  |
| HTTP server .....             | 137      |  |  |
| POP3 .....                    | 193      |  |  |
| SMTP server .....             | 187      |  |  |