Dynamic C

TCP/IP User's Manual

019-0100 • 020131-D

Dynamic C TCP/IP User's Manual

Part Number 019-0100 • 020131–D • Printed in U.S.A.

Notice to Users

Z-WORLD PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE-SUPPORT DEVICES OR SYSTEMS UNLESS A SPECIFIC WRITTEN AGREEMENT REGARDING SUCH INTENDED USE IS ENTERED INTO BETWEEN THE CUSTOMER AND Z-WORLD PRIOR TO USE. Life-support devices or systems are devices or systems intended for surgical implantation into the body or to sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling and user's manual, can be reasonably expected to result in significant injury.

No complex software or hardware system is perfect. Bugs are always present in a system of any size. In order to prevent danger to life or property, it is the responsibility of the system designer to incorporate redundant protective mechanisms appropriate to the risk involved.

The TCP/IP software used in the Rabbit 2000 TCP/IP Development Kit is designed for use only with Rabbit Semiconductor chips.

Trademarks

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

Windows® is a registered trademark of Microsoft Corporation

Z-World, Inc.

2900 Spafford Street
Davis, California 95616-6800
USA

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

Table of Contents

1. Introduction

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**. As of Dynamic C 7.05, this library is a light wrapper around **DNS.LIB**, **IP.LIB**, **NET.LIB**, **TCP.LIB** and **UDP.LIB**. These libraries implement DNS (Domain Name Server), IP, TCP, and UDP (User Datagram Protocol). This, along with the libraries **ARP.LIB** and **ICMP.LIB**, are 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.

2. TCP/IP Engine

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**.

> **NOTE:** Starting with Dynamic C version 7.05, **DCRTCP.LIB** is a light wrapper around **DNS.LIB**, **IP.LIB**, **NET.LIB**, **TCP.LIB** and **UDP.LIB**. No changes are required to existing code.

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**. 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. Note that initialization may take longer when using DHCP as opposed to static configuration, but this depends on your server.

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 names.
- list of name servers.

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.

2.1.2.1 BOOTP/DHCP Control Macros

Various macros control the use of DHCP. Apart from setting these macros before '#use dcrtcp.lib', there is typically very little additional work that needs to be done to use DHCP/BOOTP services. Most of the work is done automatically when you call **sock_init()** to initialize TCP/IP. There are more control macros available than what are listed here. Please look at the beginning of the file **/lib/tcpip/bootp.lib** for more information.

USE_DHCP

If this macro is defined, the target uses BOOTP and/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.

DHCP_CLIENT_ID clientid_char_ptr
DHCP_CLIENT_ID_LEN clientid_length

Define a client identifier string. Since the client ID can contain binary data, the length of this string must be specified as well. This string MUST be unique amongst all clients in an administrative domain, thus in practice the client ID must be individually set for each client e.g. via front-panel configuration. It is NOT recommended to program a hard-coded string (as for class ID). Note that RFC2132 recommends that the first byte of the string should be zero if the client ID is not actually the hardware type and address of the client (see next).

DHCP_CLIENT_ID_MAC

If defined, this overrides **DHCP_CLIENT_ID**, and automatically sets the client ID string to be the hardware type (1 for ethernet) and MAC address, as suggested by RFC2132.

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**: If BOOTP/DHCP fails, then a runtime error occurs. This is the default.
- **1**: If BOOTP fails, then use the values in **MY_IP_ADDRESS** etc. If those macros are not defined, a runtime error occurs.

_dhcphost

IP address of last-used DHCP server (~0UL if none). If **_survivebootp** is true, then this variable should be checked to see if DHCP/BOOTP was actually used to obtain the lease. If **_dhcphost** is ~0UL, then the fallback parameters (**MY_IP_ADDRESS** etc.) were used since no DHCP server responded.

_bootphost

IP address of the last-used BOOTP/TFTP server (~0UL if none). Usually obtained from the **siaddr** field of the DHCP OFFER/ACK message. This is the default host used if **NULL** is given for the hostname in the call to **tftp_exec()**. This is the host that provides the boot file.

_dhcplife, _dhcpt1, _dhcpt2

These variables contain various absolute time values (referenced against **SEC_TIMER**) at which certain aspects of the DHCP protocol get activated. **_dhcplife** is when the current lease expires. If **_dhcplife** is ~0UL (i.e. 0xFFFFFFFF) then the lease is permanent and the other variables are not used. Otherwise, **_dhcpt1** is when the current lease must be renewed by the current DHCP server. **_dhcpt2** is when the lease must be re-bound to a possibly different server, if the current server does not respond. In general, **_dhcpt1** < **_dhcpt2** < **_dhcplife**. To work out the number of seconds remaining until the current lease expires, use code similar to:

```
if (_dhcplife == ~0UL)
   printf("Lease is permanent\r\n");
else if (_dhcplife > SEC_TIMER)
   printf("Remaining lease %lu seconds\r\n",
    _dhcplife - SEC_TIMER);
else
   printf("Lease is expired\r\n");
```

_bootptimeout
Number of seconds to wait for a BOOTP or DHCP offer. If there is no response within this time (default 30 sec), then BOOTP is assumed to have failed, and the action specified by **_survivebootp** will be taken. You can set this variable to a different value before calling **sock_init()**.

_bootpdone
Is set to a non-zero value when TFTP download of the boot file is complete. This variable only exists if **DHCP_USE_TFTP** is defined. It is set to one of the following values:

0: Download not complete, or boot file not yet known.
1: Boot file download completed (check **_bootperror** for status).
2: No boot file was specified by the server.

_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 DHCP Functions

There are two user-callable functions regarding IP address leases. To obtain a lease, call **dhcp_acquire()**. To relinquish it, call **dhcp_release()**.

2.1.2.4 DHCP Sample Program

The following sample code is a 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 configuration items. A more extensive sample program is in **Samples\tcpip\dhcp.c**. It demonstrates other DHCP features, such as releasing and re-acquiring IP addresses and downloading a configuration file.

Program Name: samples/tcpip/dhcp.c

```
// 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 (_dhcpstate == 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 use fallback addresses
   _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

Starting with Dynamic C version 7.05, TCP and UDP I/O buffers are sized separately using:

- **TCP_BUF_SIZE** determines the TCP buffer size and defaults to 4096 bytes.
- **UDP_BUF_SIZE** determines the UDP buffer size and defaults to 4096 bytes.

Compatibility is maintained with earlier versions of Dynamic C. If **SOCK_BUF_SIZE** is defined, **TCP_BUF_SIZE** and **UDP_BUF_SIZE** will be assigned the value of **SOCK_BUF_SIZE**. If **SOCK_BUF_SIZE** is not defined, but **tcp_MaxBufSize** is, then **TCP_BUF_SIZE** and **UDP_BUF_SIZE** will be assigned the value of **tcp_MaxBufSize** * 2.

2.1.3.1 User-supplied Buffers

Starting with Dynamic C version 7.05, a user can associate their own buffer with a TCP or UDP socket. The memory for the buffer must be allocated by the user. This can be done with **xalloc()**, which returns a pointer to the buffer. This buffer will be tied to a socket by a call to an extended open function: **tcp_extlisten()**, **tcp_extopen()** or **udp_extopen()**. Each function requires a long pointer to the buffer and its length be passed as parameters.

2.2 TCP Socket Interface

Throughout this manual, the term socket refers to four numbers: the IP addresses and port numbers for both sides of a connection.

With Dynamic C version 6.57, each socket must have an associated **tcp_Socket** structure of 145 bytes or a **udp_Socket** structure of 62 bytes. The I/O buffers are in extended memory. For Dynamic C 7.05 these sizes are 132 bytes and 48 bytes, respectively.

For earlier versions of Dynamic C, each socket must have a **tcp_Socket** data structure that holds the socket state and I/O buffers. These structures are, by default, around 4200 bytes each. The majority of this space is used by the input and output buffers.

2.2.1 Number of Sockets

Starting with Dynamic C version 7.05, there are two macros that define the number of sockets available:

- **MAX_TCP_SOCKET_BUFFERS** determines the maximum number of TCP sockets with preallocated buffers. The default is 4. A buffer is tied to a socket with the first call to **tcp_open()** or **tcp_listen()**.
- **MAX_UDP_SOCKET_BUFFERS** determines the maximum number of UDP sockets with preallocated buffers. The default is 0. A buffer is tied to a socket with the first call to **udp_open()**.

Note that DNS does not need a UDP socket buffer since it manages its own buffer. DHCP and **TFTP.LIB**, however, each need one UDP socket buffer.

Prior to Dynamic C version 7.05, **MAX_SOCKETS** defined the number of sockets that could be allocated, not including the socket for DNS lookups. If you use libraries such as **HTTP.LIB** or **FTP_SERVER.LIB**, you must provide enough sockets in **MAX_SOCKETS** for them also.

In Dynamic C 7.05 (and later), if **MAX_SOCKETS** is defined in an application program, **MAX_TCP_SOCKET_BUFFERS** will be assigned the value of **MAX_SOCKETS**.

2.2.2 Passive Open

There are two ways to open a TCP socket, passive and active. To passively open a socket, call **`tcp_listen()`**; then wait for someone to contact your device. This type of open is commonly used for Internet servers that listen on a well-known port, like 80 for HTTP (Hypertext Transfer Protocol) servers. You supply **`tcp_listen()`** with a pointer to a **`tcp_Socket`** data structure, the local port number others will be contacting on your device, and the IP address and port number that are valid for the device. If you want to be able to accept connections from any IP address or any port number, set one or both to zero.

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 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, causing an unused local port between 1024 and 65535 to be selected.

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 for TCP Sockets

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 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 for TCP Sockets

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 estab-

lished 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 for TCP Sockets

- **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 the mode with a call to **sock_mode()**.

When a socket is in ASCII mode, it is assumed 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 ≥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.

Do not use **sock_gets()**in binary mode.

2.3 UDP Socket 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.

Broadcast Packets

UDP can send broadcast packets (i.e., to send a packet to a number of computers on the same network). This is accomplished by setting the remote IP address to -1, in either a call to **udp_open()** or a call to **udp_sendto()**. When used properly, broadcasts can reduce overall network traffic because information does not have to be duplicated when there are multiple destinations.

Checksums

There is an optional checksum field inside the UDP header. This field verifies the header and 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 TCP/IP engine. 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**.

In Dynamic C version 7.20, some convenient macros offer a safer, faster alternative to using **sock_mode()**. They are **udp_set_chk(s)** and **udp_set_nochk(s)**.

Improved Interface

With Dynamic C version 7.05 there is a redesigned UDP API. The new interface is incompatible with the previous one. Section 2.3.1 covers the new interface and Section 2.3.2 covers the previous one.

2.3.1 Dynamic C 7.05 (and later)

This UDP interface is a record service. It receives distinct datagrams and passes them as such to the user program. The socket I/O functions available for TCP sockets will not work for UDP sockets.

See Section 2.3.3 for information on porting a program to the new UDP interface.

2.3.1.1 Control Functions for UDP Sockets

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

- **sock_flush**
- **sock_flushnext**
- **udp_close**
- **udp_open**

2.3.1.2 I/O Functions for UDP Sockets

These functions handle datagram-at-a-time I/O:

- **udp_recv**
- **udp_recvfrom**
- **udp_send**
- **udp_sendto**

The write function, **`udp_sendto()`**, allows the remote IP address and port number to be specified. The read function, **`udp_recvfrom()`**, identifies the IP address and port number of the host that sent the datagram. There is no longer a UDP read function that blocks until data is ready.

2.3.1.3 Status Function for UDP Sockets

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**

For a udp socket, **`sock_bytesready()`** returns the number of bytes in the next datagram in the socket buffer, or -1 if no datagrams are waiting. Note that a return of 0 is valid, since a datagram can have 0 bytes of data.

2.3.2 UDP Interface Prior to Dynamic C 7.05

This interface is basically the TCP socket interface with some additional functions for simulating a record service. Some of the TCP socket functions work differently for UDP because of its connectionless state. The descriptions for the applicable functions details these differences.

2.3.2.1 I/O Functions for UDP Sockets

Prior to Dynamic C 7.05, the functions that handle UDP socket I/O are mostly the same functions that handle TCP socket I/O.

- **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_close**
- **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 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()`**.

2.3.2.2 Opening and Closing a UDP Socket

udp_open() 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.

2.3.2.3 Writing to a UDP Socket

Prior to Dynamic C 7.05, the normal socket functions 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.

2.3.2.4 Reading From a UDP Socket

There are two ways to read UDP packets prior to Dynamic C 7.05. 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, it is stuffed 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()**.

2.3.3 Porting Programs from the older UDP API to the new UDP API

To update applications written with the older-style UDP API, use the mapping information in the following table.

UDP API prior to Dynamic C 7.05	UDP API starting with Dynamic C 7.05
MAX_SOCKETS	**MAX_UDP_SOCKET_BUFFERS** and **MAX_TCP_SOCKET_BUFFERS**
SOCK_BUF_SIZE	**UDP_BUF_SIZE** and **TCP_BUF_SIZE**
udp_open()	**udp_open()**
sock_write(), sock_fastwrite()	**udp_send()** or **udp_sendto()**
sock_read()(blocking function)	**udp_recv()** or **udp_recvfrom()** (nonblocking functions)
sock_fastread()	**udp_recv()** or **udp_recvfrom()**
sock_recv_init()	**udp_extopen()** (to specify your own buffer)
sock_recv()	**udp_recv()**
sock_recv_from()	**udp_recvfrom()**
sock_close()	**sock_close()** or **udp_close()**
sock_bytesready()	**sock_bytesready()**
sock_dataready()	**sock_dataready()**

2.4 DNS Lookups

Starting with Dynamic C 7.05, non-blocking DNS lookups are supported. Prior to DC 7.05, there was only the blocking function, **resolve()**. Compatibility has been preserved for **resolve()**, **MAX_DOMAIN_LENGTH** , and **DISABLE_DNS**.

The application program has to do two things to resolve a host name:

1. Call **resolve_name_start()** to start the process.
2. Call **resolve_name_check()** to check for a response.

Call **resolve_cancel()**to cancel a pending lookup.

2.4.1 Configuration Macros for DNS Lookups

DISABLE_DNS

If this macro is defined, DNS lookups will not be done. The DNS subsystem will not be compiled in, saving some code space and memory.

DNS_MAX_RESOLVES

4 by default. This is the maximum number of concurrent DNS queries. It specifies the size of an internal table that is allocated in xmem.

DNS_MAX_NAME

64 by default. Specifies the maximum size in bytes of a host name that can be resolved. This number includes any appended default domain and the **NULL**-terminator. Backwards compatibility exists for the **MAX_DOMAIN_LENGTH** macro. Its value will be overridden with the value **DNS_MAX_NAME** if it is defined.

For temporary storage, a variable of this size must be placed on the stack in DNS processing. Normally, this is not a problem. However, for µC/OS-II with a small stack and a large value for **DNS_MAX_NAME**, this could be an issue.

DNS_MAX_DATAGRAM_SIZE

512 by default. Specifies the maximum length in bytes of a DNS datagram that can be sent or received. A root data buffer of this size is allocated for DNS support.

DNS_RETRY_TIMEOUT

2000 by default. Specifies the number of milliseconds to wait before retrying a DNS request. If a request to a nameserver times out, then the next nameserver is tried. If that times out, then the next one is tried, in order, until it wraps around to the first nameserver again (or runs out of retries).

DNS_NUMBER_RETRIES

2 by default. Specifies the number of times a request will be retried after an error or a timeout. The first attempt does not constitute a retry. A retry only occurs when a request has timed out, or when a nameserver returns an unintelligible response. That is, if a host name is looked up and the nameserver reports that it does not exist and then the DNS resolver tries the same host name with or without the default domain, that does not constitute a retry.

DNS_MIN_KEEP_COMPLETED

10000 by default. Specifies the number of milliseconds a completed request is guaranteed to be valid for **resolve_name_check()**. After this time, the entry in the internal table corresponding to this request can be reused for a subsequent request.

DNS_SOCK_BUF_SIZE

1024 by default. Specifies the size in bytes of an xmem buffer for the DNS socket. Note that this means that the DNS socket does not use a buffer from the socket buffer pool.

2.5 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.

Program Name: /samples/tcpip/icmp/pingme.c

```
/*
 * You must change the following values to whatever
 * your local IP address, netmask, and gateway are.
 * Contact your network administrator for these numbers.
 */

#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**.

2.5.1 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, the TCP/IP engine is ready to handle incoming packets.

2.5.2 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.

2.5.3 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, tens of milliseconds for typical packet processing, and hundreds of milliseconds under exceptional circumstances.

2.6 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.

2.6.1 Blocking vs. Non-Blocking

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

2.6.1.1 Non-Blocking Functions

The **sock_fastread()** and **sock_preread()** functions read all available data in the buffers, and return immediately. Similarly, the **sock_fastwrite()** function fills the buffers 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;
}
```

2.6.1.2 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.

2.7 Multitasking and TCP/IP

Dynamic C's TCP/IP implementation is compatible with both µC/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.

2.7.1 µC/OS-II

The TCP/IP engine may be used with the µC/OS-II real-time kernel. The line

```
#use ucos2.lib
```

must appear before the line

```
#use dcrtcp.lib
```

in the application program. Also be sure to call **OSInit()** before calling **sock_init()**.

Dynamic C version 7.05 and later requires the macro **MAX_SOCKET_LOCKS** for µC/OS-II support. If it is not defined, it will default to **MAX_TCP_SOCKET_BUFFERS** + **TOTAL_UDP_SOCKET_BUFFERS** (which is **MAX_UDP_SOCKET_BUFFERS** + 1 if there are DNS lookups).

If buffers have been **xalloc**'d for socket I/O, they should be accounted for in **MAX_SOCKET_LOCKS**.

2.7.2 Cooperative Multitasking

The following program demonstrates the use of multiple TCP sockets with costatements.

Program Name: costate_tcp.c

```
#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 *s, int port, char *buf){
   auto int length, space_avaliable;

   tcp_listen(s, 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;
}
```

Program Name: costate_tcp.c (continued)

```
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 socet
         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));
      }
   }
}
```

2.8 Function Reference

This section contains descriptions for all user-callable functions in **DCRTCP.LIB**. Starting with Dynamic C 7.05, **DCRTCP.LIB** is a light wrapper around **DNS.LIB**, **IP.LIB**, **NET.LIB**, **TCP.LIB** and **UDP.LIB**. This update requires no changes to existing code.

Descriptions for select user-callable functions in **ARP.LIB**, **ICMP.LIB**, **BSDNAME.LIB** and **XMEM.LIB** are also included here. Note that **ARP.LIB**, **ICMP.LIB** and **BSDNAME.LIB** are automatically **#use**'d from **DCRTCP.LIB**.

arpcache_create

```
ATHandle arpcache_create(longword ipaddr)
```

DESCRIPTION

Create a new entry in the ARP cache table for the specified IP address. If a matching entry for that address already exists, then that entry is returned. Otherwise, a new entry is initialized and returned. If a new entry is created, then an old entry may need to be purged. If this is not possible, then **ATH_NOENTRIES** is returned.

PARAMETER

ipaddr IP address of entry.

RETURN VALUE

Positive value: success.
ATH_NOENTRIES: no space is available in the table, and none of the entries could be purged because they were all marked as permanent or router entries.

LIBRARY

ARP.LIB

arpcache_flush

```
ATHandle arpcache_flush(ATHandle ath);
```

DESCRIPTION

Mark an ARP cache table entry for flushing. This means that the given table entry will be the first entry to be re-used for a different IP address, if necessary. Any entry (including permanent and router entries) may be flushed except for the broadcast entry.

PARAMETER

ath	ARP table handle obtained from e.g. **arpcache_search()**.

RETURN VALUE

Positive value: success.
ATH_UNUSED: the table entry was unused.
ATH_INVALID: the **ath** parameter was not a valid handle.
ATH_OBSOLETE: the given handle was valid, but obsoleted by a more recent entry. No change made.

LIBRARY

ARP.LIB

arpcache_hwa

```
ATHandle arpcache_hwa(ATHandle ath, byte * hwa);
```

DESCRIPTION

Copy the ethernet (hardware) address from the given ARP cache table entry into the specified area.

PARAMETERS

ath	ARP cache table entry.
hwa	Address of where to store the hardware address (6 bytes).

RETURN VALUE

Positive value: handle to the entry.
ATH_UNUSED: the table entry was unused.
ATH_INVALID: the ath parameter was not a valid handle.
ATH_OBSOLETE: the given handle was valid, but obsoleted by a more recent entry. No change made.

LIBRARY

ARP.LIB

arpcache_load

```
ATHandle arpcache_load(ATHandle ath, byte * hwa, byte iface,
   word flags, byte router_used);
```

DESCRIPTION

Load an entry in the ARP cache table. The entry must have been created using **arpcache_create()**, or be an existing valid entry located via **arpcache_search()**.

This function is primarily intended for internal use by the ARP library, although advanced applications could also use it. Most applications should not need to call this function directly.

PARAMETERS

ath — Handle for the entry.

hwa — Hardware (ethernet) address, or **NULL**. Pass **NULL** if the current hardware address is not to be changed.

iface — Interface to use (**IF_DEFAULT** to use default, or not change current setting).

flags — Flags for entry: one or more of the following values, OR'd together:

- **ATE_PERMANENT**: permanent entry
- **ATE_RESOLVING**: initiate network resolve for this entry (**hwa** is ignored if this flag is set)
- **ATE_RESOLVED**: this entry now resolved
- **ATE_ROUTER_ENT**: this is a router entry
- **ATE_FLUSH**: mark this entry for flush
- **ATE_VOLATILE**: set short timeout for this entry
- **ATE_ROUTER_HOP**: this entry uses the specified router as the first hop. **hwa** ignored.
- **ATE_REDIRECTED**: this entry redirected by ICMP.

Only one of **ATE_ROUTER_ENT** or **ATE_ROUTER_HOP** should be set. For either of these, the next parameter indicates the router table entry to use.

Only one of **ATE_RESOLVING** or **ATE_RESOLVED** should be set.

router_used — Router table entry. Only used if one of **ATE_ROUTER_ENT** or **ATE_ROUTER_HOP** is set in the flags parameter.

arpcache_load (continued)

RETURN VALUE

Positive value: success.

ATH_NOROUTER: the specified router entry number is invalid. This can be because the router_used parameter is bad, or because the router entry has a mismatching ATH.

ATH_INVALID: invalid table handle passed (or unused entry).

ATH_OBSOLETE: the given handle was valid, but obsoleted by a more recent entry. No change made.

LIBRARY

ARP.LIB

arpcache_search

```
ATHandle arpcache_search(longword ipaddr, int virt);
```

DESCRIPTION

Return handle which refers to the ARP cache table entry for the given IP address. This does not do any resolving. It only consults the existing cache entries. The returned handle is guaranteed to be valid at least until the next call to **tcp_tick()**. Usually the handle will be valid for considerably longer, however it is possible for the handle to become obsolete if the cache entry is re-used for a different address. The caller should be able to deal with this possibility. The entry returned for the broadcast address is guaranteed to be permanent.

PARAMETERS

ipaddr	IP address to locate in the cache. This may be -1L to locate the broadcast entry or our own IP address to return the "loopback" entry.
virt	0: do not return the broadcast or loopback entries. 1: allow the broadcast or loopback entries.

Return Value

Positive value: handle to the entry.
ATH_NOTFOUND: no entry exists for the given IP address.

LIBRARY

ARP.LIB

_arp_resolve

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

DESCRIPTION

Gets the Ethernet address for the given IP address. This function is deprecated starting in Dynamic C 7.20.

PARAMETERS

ina	The IP address to resolve to an Ethernet address.
ethap	The buffer to hold the Ethernet address.
nowait	If 0, return within 750 ms; else if !0 wait up to 5 seconds trying to resolve the address.

RETURN VALUE

1: Success.
0: Failure.

LIBRARY

ARP.LIB

arpresolve_check

```
ATHandle arpresolve_check(ATHandle ath, longword ipaddr)
```

DESCRIPTION

Check up on status of resolve process initiated by **arpresolve_start()**. This function should be called regularly to ensure that an ARP table handle is pointing to the correct entry, and that the entry is still current.

This caller must call **tcp_tick()** if spinning on this function.

PARAMETERS

ath	ARP Table Handle obtained from **arpresolve_start()**.
ipaddr	IP address specified to **arpresolve_start()**. If this is zero, no check is performed. Otherwise, the ARP table entry is checked to see that it is the correct entry for the specified IP address.

RETURN VALUE

Positive value: completed successfully. The return value will be the same as the ath parameter.
ATH_AGAIN: not yet completed, try again later.
ATH_FAILED: completed in error. Address cannot be resolved because of a network configuration problem.
ATH_TIMEDOUT: resolve timed out. No response from addressee within the configured time limit.
ATH_INVALID: the **ath** parameter was not a valid handle|.
ATH_OBSOLETE: the given handle was valid, but obsoleted by a more recent entry. Restart using **arpresolve_start()**.
ATH_MISMATCH: the **ipaddr** parameter was not zero, and the IP address does not match the table entry.

LIBRARY

ARP.LIB

arpresolve_ipaddr

```
longword arpresolve_ipaddr(ATHandle ath)
```

DESCRIPTION

Given an ARP table handle, return the IP address of the corresponding table entry.

PARAMETER

ath ARP Table Handle obtained from e.g. **router_for()**.

RETURN VALUE

0: An error occurred, such as an invalid or obsolete handle.
0xFFFFFFFF: The handle refers to either the broadcast address, or to a point-to-point entry whose IP address is not defined.
Else: An IP address. This may be 127.0.0.1 for the loopback entry.

LIBRARY

ARP.LIB

arpresolve_start

```
ATHandle arpresolve_start(longword ipaddr);
```

DESCRIPTION

Start resolve process for the given IP address. This may return immediately if the IP address is in the ARP cache table and still valid. Otherwise, if the IP address is on the local subnet then an ARP resolve request is issued through the appropriate interface. If the address is not on the local subnet, then a router table entry is used and no network activity is necessary (unless the router itself is not resolved, in which case its resolution commences).

PARAMETER

ipaddr IP address of host whose hardware address is to be resolved.

RETURN VALUE

Positive value: success. The value is actually the ATH of the ARP cache table entry which is (or will be) used. This value should be passed to subsequent calls to **arpresolve_check()**.

ATH_NOENTRIES: no space is available in the table, and none of the entries could be purged, because they were all marked as permanent or router entries.

ATH_NOROUTER: no router ("gateway") is configured for the specified address, which is not on the local subnet.

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

host_ip	IP address to receive ping reply from.
sequence_number	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

SEE ALSO

_ping, _send_ping

dhcp_acquire

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

DESCRIPTION

This function acquires a DHCP lease that 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 reassign the same IP address previously used by the client, however this is sometimes not possible.

LIBRARY

BOOTP.LIB

dhcp_get_timezone

```
int dhcp_get_timezone( long * seconds );
```

DESCRIPTION

This function returns the time zone offset provided by the DHCP server, if any, or uses the fallback time zone defined by the **TIMEZONE** macro. Note that **TIMEZONE** is expressed in hours, whereas the return result is in seconds.

PARAMETERS

seconds Pointer to result longword. If the return value is 0 (OK), then this will be set to the number of seconds offset from Coordinated Universal Time (UTC). The value will be negative for west; positive for east of Greenwich. If the return value is -1, then the result will be set using the hard-coded value from the macro **TIMEZONE** (converted to seconds by multiplying by 3600), or zero if this macro is not defined.

Return Value

0: Time zone obtained from DHCP.
-1: Time zone not valid, or not yet obtained, or not using DHCP.

LIBRARY

BOOTP.LIB

dhcp_release

```
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 that 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

getdomainname

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

DESCRIPTION

Gets the current domain name. For example, if the controller's internet address is "test.mynetwork.com" then "mynetwork" is the domain portion of the name.

The domain name can be changed by the **setdomainname()** function.

PARAMETERS

name	Buffer to place the name.
length	Maximum length of the name, or zero to get a pointer to the internal domain name string. Do not modify this string!

RETURN VALUE

If **length** ≥1: Pointer to **name**. If **length** is not long enough to hold the domain name, a **NULL** string is written to **name**.

If **length** = 0: 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));
}
```

gethostid

```
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

IP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sethostid

EXAMPLE

```
main() {
   char buffer[ 512 ];
   sock_init();
   printf("My IP address is %s\n", inet_ntoa( buffer, gethostid()));
}
```

gethostname

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

DESCRIPTION

Gets the host portion of our name. For example if the controller's internet address is "test.mynetwork.com" the host portion of the name would be "test."

The host name can be changed by the **sethostname()** function.

PARAMETERS

name	Buffer to place the name.
length	Maximum length of the name, or zero for the internal host name buffer. Do not modify this buffer.

RETURN VALUE

length ≥1: return **name**;
length = 0: return internal host name buffer (do not modify!)

LIBRARY

BSDNAME.LIB

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

NET.LIB (Prior to DC 7.05, this was DCRTCP.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

NET.LIB (Prior to DC 7.05, this was DCRTCP.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

IP.LIB (Prior to DC 7.05, this was DCRTCP.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

s	Location to place the dotted decimal string. A sufficient buffer size would be 16 bytes.
ip	The IP address to convert.

RETURN VALUE

Pointer to the dotted decimal string pointed to by **s**.

LIBRARY

IP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

inet_addr

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

value	Network-ordered long word.

RETURN VALUE

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

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

htons, ntohs, htonl

ntohs

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

DESCRIPTION

Converts network-ordered word to host-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

NET.LIB (Prior to DC 7.05, this was DCRTCP.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

nic	This parameter is reserved for future expandability and for now should be assigned a value of 0.
buffer	Place to copy address to. Must be at least 6 byes.

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

host_ip IP address to send ping.

sequence_number User-defined sequence number.

RETURN VALUE

0: Success.
1: Failure: unable to resolve hardware address.
-1: Failure: unable to transmit ICMP request.

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

s 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 64 characters. If it is necessary to lookup larger names include the following line in the application program:

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

If **DISABLE_DNS** has been defined, **resolve()** 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: Failure.
!0: The IP address ***host_string** resolves to.

LIBRARY

DNS.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

_arp_resolve, inet_addr, inet_ntoa

resolve_cancel

```
int resolve_cancel(int handle);
```

DESCRIPTION

Cancels the resolve request represented by the given handle. If the handle is 0, then this function cancels all outstanding resolve requests.

PARAMETERS

handle Handle that represents a DNS lookup process, or 0 to cancel all outstanding resolve requests.

RETURN VALUE

RESOLVE_SUCCESS: The resolve request corresponding to the given handle has been cancelled. The given handle will no longer be valid after this value is returned.

RESOLVE_HANDLENOTVALID: There is no request for the given handle.

RESOLVE_NONAMESERVER: No nameserver has been defined.

LIBRARY

DNS.LIB

SEE ALSO

resolve_name_start, resolve_name_check, resolve

resolve_name_check

```
int resolve_name_check(int handle, longword* resolved_ip);
```

DESCRIPTION

Checks if the DNS lookup represented by the given handle has completed. On success, it fills in the resolved IP address in the space pointed to by **resolved_ip**.

PARAMETERS

handle Handle that represents a DNS lookup process.

resolved_ip A pointer to a user-supplied **longword** where the resolved IP address will be placed.

RETURN VALUE

RESOLVE_SUCCESS: The address was successfully resolved. The given handle will no longer be valid after this value is returned.

RESOLVE_AGAIN The resolve process has not completed, call this function again.

RESOLVE_FAILED The DNS server responded that the given host name does not exist. The given handle will no longer be valid if **RESOLVE_FAILED** is returned.

RESOLVE_TIMEDOUT The request has been cancelled because a response from the DNS server was not received before the last timeout expired. The given handle will no longer be valid after this value is returned.

RESOLVE_HANDLENOTVALID There is no DNS lookup occurring for the given handle.

RESOLVE_NONAMESERVER: No nameserver has been defined.

LIBRARY

DNS.LIB

SEE ALSO

resolve_name_start, resolve_cancel, resolve

resolve_name_start

```
int resolve_name_start(char* hostname);
```

DESCRIPTION

Starts the process of resolving a host name into an IP address. The given host name is limited to **DNS_MAX_NAME** characters, which is 64 by default (63 characters + the **NULL** terminator). If a default domain is to be added, then the two strings together are limited to **DNS_MAX_NAME**.

If **hostname** does not contain a '.' then the default domain (**MY_DOMAIN**) , if provided, is appended to **hostname**. If **hostname** with the appended default domain does not exist, **hostname** is tried by itself. If that also fails, the lookup fails.

If **hostname** does contain a '.' then **hostname** is looked up by itself. If it does not exist, the default domain is appended, and that combination is tried. If that also fails, the lookup fails.

If **hostname** ends with a '.', then the default domain is not appended. The host name is considered "fully qualified." The lookup is attempted without the ending '.' and if that fails no other combinations are attempted.

This function returns a handle that must be used in the subsequent **resolve_name_check()** and **resolve_cancel()** functions.

PARAMETERS

hostname Host name to convert to an IP address

RETURN VALUE

>**0**: Handle for calls to **resolve_name_check()** and **resolve_cancel()**.
RESOLVE_NOENTRIES: Could not start the resolve process because there were no resolve entries free.
RESOLVE_LONGHOSTNAME: The given hostname was too large.
RESOLVE_NONAMESERVER: No nameserver has been defined.

LIBRARY

DNS.LIB

SEE ALSO

resolve_name_check, resolve_cancel, resolve

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

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

EXAMPLE

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

NOTE: 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!

router_add

```
ATHandle router_add(longword ipaddr, byte iface, longword
   subnet, longword mask, word flags);
```

DESCRIPTION

Add a router to the router table. The same router can be added multiple times, with different subnet and mask. Normally, only one entry is needed in order to access non-local subnets: this entry should be specified with a zero mask. The hardware address of the router is not immediately resolved, however this can be done explicitly by calling **arpresolve_start()** with the same IP address. Otherwise, the router will be resolved only when it first becomes necessary.

PARAMETERS

ipaddr	IP address of the router. This address should be on the local subnet, since non-local routers are not supported.
iface	Interface to use to access this router, or **IF_DEFAULT**.
subnet	Subnet accessible through this entry.
mask	Subnet mask for this entry.
flags	Flags word: set to zero (non-zero reserved for internal use).

RETURN VALUE

Positive value: completed successfully. The return value is the ARP cache table entry for this router.

ATH_NOENTRIES: insufficient space in either the router or ARP cache tables.

LIBRARY

ARP.LIB

router_del_all

```
void router_del_all(void);
```

DESCRIPTION

Delete all router table entries. This will make any host that is not on the local subnet inaccessible. This function is usually called in preparation for adding a new router entry.

LIBRARY

ARP.LIB

router_delete

```
ATHandle router_delete(longword ipaddr);
```

DESCRIPTION

Delete a router from the router table. All instances of the router's IP address are deleted, and the ARP cache table entry is flushed.

PARAMETER

ipaddr	IP address of the router. This address should be on the local subnet, since non-local routers are not supported.

RETURN VALUE

Positive value: completed successfully.
ATH_NOTFOUND: specified entry did not exist.

LIBRARY

ARP.LIB

router_for

```
ATHandle router_for(longword ipaddr, byte * router_used, byte *
   r_iface);
```

DESCRIPTION

Return the ARP cache table entry corresponding to the router that handles the given IP address. If there is a pre configured router for the given address, it is selected. Otherwise, routers discovered via DHCP or ICMP router discovery are searched, with the highest preference being selected. Failing this, if there is a point-to-point interface, this is selected as the default.

An alternative mode of calling this function is invoked if **ipaddr** is zero. In this case, the default router for the specified interface (***r_iface**) is returned. If **r_iface** is **NULL**, then the default interface is assumed:**IF_DEFAULT**, the only interface supported at present. **IF_DEFAULT** may refer to the primary ethernet NIC or a PPP connection that uses a serial port or the primary ethernet NIC.

PARAMETERS

ipaddr	IP address of the host which is not on the local subnet.
router_used	If not **NULL**, the byte at this location is set to the index of the router in the router table.
r_iface	If not **NULL**, the byte at this location is set to the interface number that can access the router.

RETURN VALUE

Positive value: completed successfully.
ATH_NOROUTER: no suitable router found. Either no router is configured, or the given IP address is on the local subnet.

LIBRARY

ARP.LIB

router_print

```
int router_print(byte r)
```

DESCRIPTION

Print a router table entry, indexed by '**r**.' This is for debugging only, since the results are printed to the Dynamic C stdio window. '**r**' may be obtained from the **router_for()** function, by passing **&r** as the **router_used** parameter to that function.

If the specified router entry is not in use, nothing is printed and the return value is non-zero. Otherwise, the information is printed and zero returned.

See **router_printall()** for a description of the output fields printed.

PARAMETER

r	Router table index. A number from 0 through (**ARP_ROUTER_TABLE_SIZE**-1).

RETURN VALUE

0: Success, information printed to stdio window.
!0: Entry is not in use.

LIBRARY

ARP.LIB

SEE ALSO

router_printall

router_printall

```
int router_printall(void)
```

DESCRIPTION

Print all router table entries. This is for debugging only, since the results are printed to the Dynamic C stdio window. If no routers exist in the table, nothing is printed and the return value is non-zero.

There are 6 fields for each router entry:

Router Table Entry Field	Description of Field
#	The entry number.
Flags	A list of the following characters: **P** = this entry pre configured **T** = transient entry **D** = added by DHCP/BOOTP **R** = added by ICMP redirect **?** = router not reachable **H** = router's hardware address resolved
Address	Either the router's IP address or an indication that the entry is a point-to-point link.
i/f	Interface number.
Net/preference	For pre configured entries, indicates the network(s) which are served by this entry (the Mask indicates which bits of the IP address are used to match with the network address). For non-pre configured entries, this is the "preference value" assigned.
Mask/exp(sec)	For pre configured entries, the bitmask to apply to IP addresses when matching against the above network. Otherwise, is the expiry time from the present, in seconds, of a transient entry.

RETURN VALUE

0: Success, information printed to stdio window.

!0: No routers in the table.

LIBRARY

ARP.LIB

_send_ping

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

DESCRIPTION

Generates an ICMP request for host.

PARAMETERS

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

RETURN VALUE

0: Success.
1: Failure: unable to resolve hardware address.
-1: Failure: unable to transmit ICMP request.

LIBRARY

ICMP.LIB

SEE ALSO

_chk_ping, _ping, sock_set_ttl, sock_set_tos

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 current IP address. Changing this address will disrupt existing TCP or UDP sessions. You should close all sockets before calling this function.

Normally there is no need to call this function. The macro **MY_IP_ADDRESS** defines an initial IP address for this host, or you can define **USE_DHCP** to obtain a dynamically assigned address. In either case, it is not recommended to use this function to change the address.

PARAMETERS

ip New IP address.

RETURN VALUE

New IP address.

LIBRARY

IP.LIB (Prior to DC 7.05, this was 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

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_close,

sock_bytesready

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

DESCRIPTION

For TCP sockets:

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.

For UDP sockets:

Returns the number of bytes in the next datagram to be read. If it is a datagram with no data (an empty datagram), then it will return 0. If there are no datagrams waiting, then it returns -1.

PARAMETERS

s	Pointer to a socket.

RETURN VALUE

-1: No bytes waiting to be read.
0: If in ASCII mode and a blank line is waiting to be read;
for DC 7.05 and later, a UDP datagram with 0 bytes of data is waiting to be read.
>0: The number of bytes waiting to be read.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

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. Wait for **tcp_tick()** to return zero 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

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_abort, sock_tick

sock_dataready

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

DESCRIPTION

Returns the number of bytes waiting to be read. If the socket is in ASCII mode, this function returns zero if a newline character has not been read or the buffer is not full. For UDP sockets, the function returns the number of bytes in the next datagram.

This function cannot tell the difference between no bytes to read and either a blank line or a UDP datagram with no data. For this reason, use **sock_bytesready()** instead.

PARAMETERS

s	Pointer to a socket.

RETURN VALUE

0: No bytes to read;
or newline not yet read if the socket is in ASCII mode;
or (for DC 7.05 and later) if a UDP datagram has 0 bytes of data waiting to be read.

>0: Number of bytes ready to read.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_bytesready

sockerr

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

DESCRIPTION

Gets the last ASCII error message recorded for the specified socket. Use of this function will introduce a lot of string constants in root memory. For production programs, it is better to use error numbers (without translation to strings).

PARAMETERS

s Pointer to a socket.

RETURN VALUE

Pointer to the string that represents the last error message for the socket.

NULL pointer if there have been no errors.

If the symbol **SOCKERR_NO_RETURN_NULL** is defined, then if no error occurred the string "OK" will be returned instead of a **NULL** pointer.

The error messages are read-only; do not modify them!

LIBRARY

NETERRNO.LIB

SEE ALSO

sock_error, sock_perror

EXAMPLE

```
char *p;
...
if ( p = sockerr( s ))
   printf("Socket closed with error '%s'\n\r", p );
```

sock_error

```
int sock_error(void *s, int clear);
```

DESCRIPTION

Return the most recent error number for the specified socket, which may be a TCP or UDP socket. Up to two error codes may be queued to a socket.

PARAMETERS

s	socket
clear	0: do not clear the returned error condition. 1: clear the returned error from the socket. You can call this function again to get the next older error code (if any).

RETURN VALUE

0: no error.
!0: one of the **NETERR_*** constants defined in **NETERRNO.LIB**.

LIBRARY

NETERRNO.LIB

SEE ALSO

sockerr, sock_perror

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_established()** 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

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_bytesready, sockstate

sock_fastread

```
int sock_fastread( tcp_Socket *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.

PARAMETERS

s	Pointer to a socket.
dp	Buffer to put bytes that are read.
len	Maximum number of bytes to write to the buffer.

RETURN VALUE

≥**0**: Success, number of bytes read.
-1: Error.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_read, sock_fastwrite, sock_write

EXAMPLE

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 thus yields poor performance.

```
do {
   /* this function does not block */
   len = sock_fastread( s, buffer, sizeof(buffer)-1 );
   if (len>0) {
      buffer[ len ] = 0;
      printf( "%s", buffer );
   }
} while(tcp_tick(s));
```

sock_fastwrite

```
int sock_fastwrite( tcp_Socket *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 <= 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

s	Pointer to a socket.
dp	Buffer to be written.
len	Maximum number of bytes to write to the socket.

RETURN VALUE

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

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_flush

```
void sock_flush( tcp_Socket *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

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_flushnext, sock_fastwrite, sock_write, sockerr

sock_flushnext

```
void sock_flushnext( tcp_Socket *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 **DCRTCP.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

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_write, sock_fastread, sock_read, sockerr, sock_flush, sock_flushnext

sock_getc

```
int sock_getc( tcp_Socket *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.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

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

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

EXAMPLE

```
do {
   if (sock_bytesready( s ) > 0)
      putchar( sock_getc( s ));
} while (tcp_tick(s));
```

sock_gets

```
int sock_gets( tcp_Socket *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 the function **sock_mode()** or the macro **tcp_set_ascii()**.

This function cannot be used on UDP sockets after **sock_recv_init()** is called.

PARAMETERS

s	Pointer to a socket
text	Buffer to put the string.
len	Max length of buffer.

RETURN VALUE

0: Either the buffer is empty or the buffer has room and the connection can get more data, but no '\r' or '\n' was read.
>0: The length of the string.
-1: Function was called with a UDP socket (valid for Dynamic C 7.05 and later).

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_puts, sock_putc, sock_getc, sock_read, sock_write

EXAMPLE

```
sock_mode( s, TCP_MODE_ASCII );
do {
   if (sock_bytesready( s ) > 0) {
      sock_gets( s, buffer, sizeof(buffer)-1 );
      puts( buffer );
   }
} while (tcp_tick( s );
```

sock_init

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

DESCRIPTION

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

The return value indicates whether the network is available. If the return code is 1or 3, then the network is not usable. If the return code is 2, then the network is usable because suitable fallbacks were defined; particularly **MY_IP_ADDRESS**. It is only possible to get return codes 2 or 3 is **USE_DHCP** is defined. Typically, return codes 2 or 3 mean that there is no DHCP server available within approximately 30 seconds from calling **sock_init()**. The timeout can be adjusted by setting the global variable **_bootptimeout** to a suitable value before calling **sock_init()**.

If **USE_DHCP** and **MY_IP_ADDRESS** are both defined, then by default the global variable **_survivebootp** will be set to TRUE. If **_survivebootp** is set to FALSE before calling **sock_init()**, then the fallbacks will not be used. That is, return code 2 will never be returned.

If you use **ucos2.lib**, be sure to call **OSInit()** before calling **sock_init()**.

RETURN VALUE

0: OK.
1: Ethernet packet driver initialization failed.
2: DHCP failed, using fallback definitions.
3: DHCP failed, no fallbacks defined.
Other: reserved.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_mode

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

DESCRIPTION

Change some of the socket options. Depending on whether **s** is a TCP or UDP socket, you may pass OR'd combinations of the following flags in the mode parameter. For a TCP socket, only the **TCP_MODE_*** flags are relevant. For a UDP socket, only the **UDP_MODE_*** flags are relevant. Do not use the wrong flags for the given socket type.

Note: it is more convenient, faster, and safer to use the macro equivalent, if it is only desired to change one mode at a time. If you use this function, then you must specify the setting of all relevant flags (TCP or UDP).

The macros do not do socket locking so, strictly speaking, µC/OS users should call this function.

TCP Modes:

TCP_MODE_ASCII | TCP_MODE_BINARY (default)

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.

Macros:

```
tcp_set_binary(s)
tcp_set_ascii(s)
```

TCP_MODE_NAGLE (default) | 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.

Macros:

```
tcp_set_nagle(s)
tcp_set_nonagle(s)
```

sock_mode (continued)

UDP Modes:

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.

Macros:

```
udp_set_chk(s)
udp_set_nochk(s)
```

UDP_MODE_NOICMP (default) | UDP_MODE_ICMP

Marks this socket for receipt of ICMP error messages. The messages are queued like normal received datagrams, and read using **udp_recvfrom()**, which returns -3 when ICMP messages are returned instead of normal datagrams. Only ICMP messages which are relevant to the current binding of the socket are queued.

Macros:

```
udp_set_noicmp(s)
udp_set_icmp(s)
```

UDP_MODE_NODICMP (default) | UDP_MODE_DICMP

Marks this socket as the default receiver of ICMP messages which cannot be assigned to a particular UDP socket. This would be used for UDP sockets that are used with many different **sendto** addresses, since the ICMP message may refer to a message sent some time ago (with different destination address than the most recent). Only one UDP socket should be set with this mode.

Macros:

```
udp_set_nodicmp(s)
udp_set_dicmp(s)
```

PARAMETERS

s	Pointer to a socket.
mode	New mode for specified socket.

RETURN VALUE

Resulting mode flags

SEE ALSO

inet_addr

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_perror

```
void sock_perror(void *s, const char * prefix);
```

DESCRIPTION

Prints out the most recent error messages for a socket, and clear the errors. This calls **sockerr()** and **printf()**, so it should only be called for debugging a new application. The output is in the format:

[TCP|UDP] socket (ipaddr:port -> ipaddr:port) msg1; msg2

where **msg1** and, possibly, **msg2** are the most recent error messages. The initial string is "TCP" or "UDP" for open sockets, or may be "Closed" if the socket is currently closed (either TCP or UDP). Up to two error codes may be queued to a socket.

If there are no errors, nothing is printed.

PARAMETERS

s	Pointer to TCP or UDP socket.
prefix	Pointer to text to add to generated messages, or **NULL**.

LIBRARY

NETERRNO.LIB

SEE ALSO

sock_error, sockerr

sock_preread

```
int sock_preread( tcp_Socket *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

s	Pointer to a socket structure.
dp	Buffer to preread into.
len	Maximum number of bytes to preread.

RETURN VALUE

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

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_fastread, sock_fastwrite, sock_read, sock_write

sock_putc

```
byte sock_putc( tcp_Socket *s, byte c );
```

DESCRIPTION

A single character is placed on the output buffer. In the case of '\n', the buffer is flushed as described under **sock_flushnext**. No other ASCII character expansion is performed.

Note that **sock_putc** uses **sock_write**, and thus may block if the output buffer is full. See **sock_write** for more details.

PARAMETERS

s	Pointer to a socket.
c	Character to send.

RETURN VALUE

The character **c**.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_read, sock_write, sock_fastread, sock_fastwrite, sock_mode

sock_puts

```
int sock_puts( tcp_Socket *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. In binary mode, the string is sent as is.

Note that **sock_puts()** uses **sock_write()**, and thus may block if the output buffer is full. See **sock_write()** for more details.

PARAMETERS

s	Pointer to a socket.
dp	Buffer to read the string from.

RETURN VALUE

≥**0**: Length of string in **dp**.
-1: Function was called with a UDP socket (valid for Dynamic C 7.05 and later).

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_gets, sock_putc, sock_getc, sock_read, sock_write

sock_rbleft

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

DESCRIPTION

Determines the number of bytes available in the receive buffer.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

Number of bytes available in the receive buffer.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_rbsize, sock_rbused, sock_tbsize, sock_tbused, sock_tbleft

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

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_rbleft, sock_rbused, sock_tbsize, sock_tbused, sock_tbleft

sock_rbused

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

DESCRIPTION

Returns 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

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_rbleft, sock_tbsize, sock_tbused, sock_tbleft

sock_read

```
int sock_read( tcp_Socket *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 on UDP sockets after **sock_recv_init()** is called.

PARAMETERS

s	Pointer to a socket.
dp	Buffer to store bytes that are read.
len	Maximum number of bytes to write into the buffer.

RETURN VALUE

Number of bytes read: Success.
-1: Error.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_fastread, sock_fastwrite, sock_write, sockerr

EXAMPLE

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 thus yields poor performance.

```
do {
   len = sock_bytesready( s );
   if (len > 0) {
      if (len > sizeof( buffer) - 1)
      {                                    // If too many bytes, read
         len = sizeof( buffer ) - 1;       // some now, read the rest
      }                                    // next time around.
      sock_read( s, buffer, len );
      buffer[ len ] = 0;
      printf( "%s", buffer );
   }
} while ( tcp_tick( s ));
```

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. This function is not available starting with Dynamic C 7.05 (see Section 2.3).

PARAMETERS

s	Pointer to a UDP socket.
buffer	Buffer to put datagram.
maxlength	Length of buffer.

RETURN VALUE

>0: Length of datagram.
0: No datagram found.
-1: Receive buffer not initialized with **sock_recv_init()**.

LIBRARY

DCRTCP.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"
#memmap xmem

#use "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. This function is not available starting with Dynamic C 7.05 (see Section 2.3).

PARAMETERS

s	Pointer to UDP socket.
hisip	IP of remote host, according to UDP header.
hisport	Port of remote host.
buffer	Buffer to put datagram in.
len	Length of buffer.

RETURN VALUE

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

LIBRARY

DCRTCP.LIB

SEE ALSO

sock_recv, sock_recv_init

sock_recv_init

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

DESCRIPTION

This function is not available starting with Dynamic C 7.05 (see Section 2.3).

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

s	Pointer to a UDP socket.
space	Buffer of temporary storage space to store newly received packets.
len	Size of the buffer.

RETURN VALUE

0

LIBRARY

DCRTCP.LIB

SEE ALSO

sock_recv_from, sock_recv

sock_resolved

```
int sock_resolved(void * s)
```

DESCRIPTION

Check whether the socket has a valid destination hardware address. This is typically used for UDP sockets, but may also be used for TCP sockets. If this function returns zero (FALSE), then any datagrams you send using **udp_send()** or **udp_sendto()** may not be transmitted because the destination hardware address is not known.

If the current destination IP address of the socket is zero (i.e. the socket is passively opened), this function returns zero, since datagrams cannot be transmitted from a passively opened socket.

If **udp_bypass_arp()** is in effect, the return value from this function is unaffected, however datagrams will still be sent to the specified hardware address (since the normal resolve process is bypassed).

Note that a hardware address may become invalid after being valid, since the underlying ARP table may need to purge the entry. This would be rare, but if any UDP application needs to ensure that all packets are actually transmitted, which is a questionable goal since UDP is unreliable, then this function should be consulted before each send. If this function returns 0, then the UDP socket should be re-opened.

The hardware address may also be invalidated if **udp_sendto()** is called with a different destination IP address, that has not been determined based on an incoming datagram.

This function is not required for TCP sockets, since the TCP library handles these details internally.

PARAMETER

s	Pointer to open TCP or UDP socket

RETURN VALUE:

0: Destination hardware address not valid
!0: Destination hardware address resolved OK.

LIBRARY

NET.LIB

SEE ALSO

udp_extopen, arpresolve_start, arpresolve_check, udp_waitopen, udp_sendto, udp_bypass_arp

sock_set_tos

```
void sock_set_tos(void* s, byte tos);
```

DESCRIPTION

Set the IP "Type Of Service" field in outgoing packets for this socket. The given TOS will be in effect until the socket is closed. When a socket is opened (or re-opened), the TOS will be set to the default (**TCP_TOS** or **UDP_TOS** as appropriate). If not overridden, the defaults are zero (**IPTOS_DEFAULT**) in both cases.

PARAMETERS

s Pointer to open TCP or UDP socket.

tos Type Of Service. This should be one of the following values:

- **IPTOS_DEFAULT** - Default service
- **IPTOS_CHEAP** - Minimize monetary cost
- **IPTOS_RELIABLE** - Maximize reliability
- **IPTOS_CAPACIOUS** - Maximize throughput
- **IPTOS_FAST** - Minimize delay
- **IPTOS_SECURE** - Maximize security.

Other value may be used (since TOS is just a number between 0 and 255), but this should only be done for experimental purposes.

LIBRARY

NET.LIB

SEE ALSO

sock_set_ttl

sock_set_ttl

```
void sock_set_ttl(void* s, byte ttl);
```

DESCRIPTION

Set the IP “Time To Live” field in outgoing packets for this socket. The given TTL will be in effect until the socket is closed. When a socket is opened (or re-opened), the TTL will be set to the default (**TCP_TTL** or **UDP_TTL** as appropriate). If not overridden, the defaults are 64 in both cases.

PARAMETERS

s	Pointer to open TCP or UDP socket.
ttl	Time To Live. This is a value between 1 and 255. A value of zero is also accepted, but will have undesirable consequences.

LIBRARY

NET.LIB

SEE ALSO

sock_set_tos

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.

"**BAD**" more than one bit set.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_established, sock_dataready

EXAMPLE

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

sock_tbleft

```
int sock_tbleft( 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

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_rbsize, sock_rbused, sock_rbleft, sock_tbsize, sock_tbused

```
if ( sock_tbleft( 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

NET.LIB (Prior to DC 7.05, this was DCRTCP.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

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_rbsize, sock_rbused, sock_rbleft, sock_tbsize, sock_tbleft

sock_tick

```
void 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

s	Pointer to a socket.
optional_status_ptr	Pointer to status word.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was 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 non-zero the macro returns with the status word set to the value of **fptr**'s return value.

This macro has been deprecated in Dynamic C version 7.20.

PARAMETERS

s	Pointer to a socket.
seconds	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 **sock_delay**, a system variable set on configuration. Typically **sock_delay** is about 20 seconds, but can be set to something else in **main()**.
fptr	Function to call repeatedly while waiting. This is a user-supplied function.
status	Pointer to a status word.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

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** returns non-zero, the macro returns.

This macro has been deprecated in Dynamic C version 7.20.

PARAMETERS

s	Pointer to a socket.
seconds	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 **sock_delay**, a system variable set on configuration. Typically **sock_delay** is about 20 seconds, but can be set to something else in **main()**.
fptr	Function to call repeatedly while waiting. This is a user-supplied function.
status	Pointer to a status word.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

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. It returns immediately for UDP sockets.

On an error, the macro jumps to a local, user-supplied **sock_err** label. If **fptr** returns non-zero, the macro returns.

This macro has been deprecated in Dynamic C version 7.20.

PARAMETERS

s	Pointer to a socket.
seconds	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 **sock_delay**, a system variable set on configuration. Typically **sock_delay** is about 20 seconds, but can be set to something else in **main()**.
fptr	Function to call repeatedly while waiting.
status	A pointer to the status word. If this parameter is **NULL**, 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. **status** 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

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_write

```
int sock_write( tcp_Socket *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

s	Pointer to a socket
dp	Pointer to a buffer.
len	Maximum number of bytes to write to the buffer.

RETURN VALUE

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

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

sock_read, sock_fastwrite, sock_fastread, sockerr, 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

s	Pointer to a TCP socket.
fn	User-defined function.

RETURN VALUE

0

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

tcp_clearreserve

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

DESCRIPTION

This function causes the **DCRTCP.LIB** stack to handle a socket connection to the specified port normally. This undoes the action taken by **tcp_reserveport()**.

PARAMETERS

port	Port to use.

RETURN VALUE

None.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.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.

Note that there are specific (and safer) functions for modifying some of the common parameters.

This function is deprecated. It is highly recommended that you do NOT use it, since it uses strings, hence taking up lots of root data storage.

PARAMETERS

name — Setting to be changed. The possible parameters are:

MY_IP_ADDRESS: host IP address (use **sethostid()** instead)
MY_NETMASK
MY_GATEWAY: host's default gateway
MY_NAMESERVER: host's default nameserver
MY_HOSTNAME
MY_DOMAINNAME: host's domain name (use **setdomainname()** instead)

value — The value to assign to **name**.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

tcp_open, sock_close, sock_abort, sethostid, setdomainname, sethostname

tcp_extlisten

```
int tcp_extlisten( tcp_Socket *s, int iface, word lport,
   longword remip, word port, dataHandler_t datahandler, word
   reserved, long buffer, int buflen );
```

DESCRIPTION

This function tells the TCP/IP engine that an incoming session for a particular port will be accepted. The **buffer** and **buflen** parameters allow a user to supply a socket buffer, instead of using a socket buffer from the pool. **tcp_extlisten()** is an extended version of **tcp_listen()**.

PARAMETERS

s	Pointer to the socket's data structure.
iface	Local interface on which to open the socket (not yet implemented, use **IF_DEFAULT** for now).
lport	Port to listen on.
remip	IP address to accept connections from or 0 for all.
port	Port to accept connections from or 0 for all.
datahandler	Function to call when data is received, **NULL** for placing data in the socket's receive buffer. *Some details for implementation of this service have not been finalized, and it is recommended the user insert a value of* **NULL** *for the present time.*
reserved	Set to 0 for now. This parameter is for compatibility and possible future use.
buffer	Address of user-supplied socket buffer in xmem. This is the return value of **xalloc()**. If **buffer** is 0, the socket buffer for this socket is pulled from the buffer pool defined by the macro **MAX_TCP_SOCKET_BUFFERS**.
buflen	Length of user-supplied socket buffer.

RETURN VALUE

0: Failure.
1: Success.

LIBRARY

TCP.LIB

SEE ALSO

tcp_listen

tcp_extopen

```
int tcp_extopen(tcp_Socket* s, int iface, word lport, longword
   remip, word port, dataHandler_t datahandler, long buffer, int
   buflen);
```

DESCRIPTION

Actively creates a session with another machine. The **buffer** and **buflen** parameters allow a user to supply a socket buffer, instead of using a socket buffer from the pool. **tcp_extopen()** is an extended version of **tcp_open()**.

s	Pointer to socket's data structure.
iface	Local interface on which to open the socket (not yet implemented, use **IF_DEFAULT** for now).
lport	Our port, zero for the next one available in the range 1025-65536.
remip	IP address to connect to.
port	Port to connect to.
datahandler	Function to call when data is received, **NULL** for placing data in the socket's receive buffer. *Some details for implementation of this service have not been finalized, and it is recommended the user insert a value of* **NULL** *for the present time.*
buffer	Address of user-supplied socket buffer in xmem. This is the return value of **xalloc()**. If **buffer** is 0, the socket buffer for this socket is pulled from the buffer pool defined by the macro **MAX_TCP_SOCKET_BUFFERS**.
buflen	Length of user-supplied socket buffer.

RETURN VALUE

0 if open was not able resolve the remote computer's hardware address,
!0 otherwise.

LIBRARY

TCP.LIB

SEE ALSO

tcp_open

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, it 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_NUMRETRYS** 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

s	Pointer to a socket.
timeout	Period of inactivity, in seconds, before sending a keepalive or 0 to turn off keepalives.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

TCP.LIB

SEE ALSO

sock_fastread, sock_fastwrite, sock_write, sockerr

tcp_listen

```
int tcp_listen( tcp_Socket *s, word lport, longword remip, word
   port, int (*signal_handler), word reserved );
```

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()** 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

s	Pointer to a socket.
lport	Port to listen on (the local port number).
remip	IP address of the remote host to accept connections from or 0 for all.
port	Port to accept connections from or 0 for all.
signal_handler	This function is called if the connection is either closed or reset. The parameter for **signal_handler** is the pointer to a function which will be called when the socket is either closed or reset. *Some details for implementation of this service have not been finalized, and it is recommended the user insert a value of* **NULL** *for the present time.*
reserved	Set to 0 for now. This parameter is for compatibility and possible future use.

RETURN VALUE

0: Failure.

1: Success.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was 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;
    s = &telnetsock;
    tcp_listen( s, port, 0L, 0, NULL, 0);

    while (!sock_established(s) && sock_bytesready(s)==-1){
       tcp_tick(NULL);
    }
    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 ... */
    do {
       if (sock_bytesready(s) >= 0) {
          sock_gets(s, buffer, sizeof(buffer)-1);
          puts ( buffer);
       }
    } while (tcp_tick(s));
}
main() {
    sock_init();
    telnets( TELNET_PORT);
    exit( 0 );
}
```

tcp_open

```
int tcp_open( tcp_Socket *s, word lport, longword remip, 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()** to determine if there is data to be read from the buffer.

PARAMETERS

s	Pointer to a socket structure.
lport	Our local port. Use zero for the next available port in the range 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, **FINGER** or **TELNET** clients can use any local port value, so pass the value of zero for **lport** and let **DCRTCP.LIB** pick one for you.
remip	IP address to connect to.
port	Port to connect to.
signal_handler	This function is called if the connection is either closed or reset. The parameter for **signal_handler** is the pointer to a function which will be called when the socket is either closed or reset. *Some details for implementation of this service have not been finalized, and it is recommended the user insert a value of* **NULL** *for the present time.*

RETURN VALUE

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

LIBRARY

TCP.LIB (Prior to DC 7.05, this was 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"
#memmap xmem

#use "dcrtcp.lib"

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

main() {
    word status;
    word port;
    longword host;
    tcp_Socket 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 );

    while (!sock_established(&tsock) && sock_bytesready(&tsock)==-1){
       tcp_tick(NULL);
    }
    puts("Socket is established");
    sock_close( &tsock );
    exit( 0 );
}
```

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

TCP.LIB (Prior to DC 7.05, this was 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. **DCRTCP.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

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

tcp_open, sock_close, sock_abort, sock_tick

udp_bypass_arp

```
void udp_bypass_arp(udp_Socket *s, eth_address * eth);
```

DESCRIPTION

Override the normal Address Resolution Protocol for this UDP socket. This is sometimes necessary for special purposes such as if the ethernet address is to remain fixed, or if the ethernet address is not obtainable using ARP. The great majority of applications should not use this function.

If ARP bypass is in effect for a UDP socket, then **udp_sendto()** will never return the -2 return code.

PARAMETERS

s	UDP socket
eth	Pointer to override address. If **NULL**, then resume normal operation i.e. use ARP to resolve ethernet addresses. Note that the specified ethernet address must be in static storage, since only the pointer is stored.

LIBRARY

UDP.LIB

SEE ALSO

udp_sendto, udp_waitsend, sock_resolved

udp_close

```
void udp_close(udp_Socket *ds);
```

DESCRIPTION

This function closes a UDP connection.

PARAMETERS

ds	Pointer to socket's data structure.

LIBRARY

UDP.LIB

udp_extopen

```
int udp_extopen( udp_Socket *s, int iface, word lport, longword
   remip, word port, dataHandler_t datahandler, long buffer, int
   buflen );
```

DESCRIPTION

This function is an extended version of **udp_open()**. It opens a socket on a given network interface (**iface**) on a given local port (**lport**). The **iface** parameter is not currently supported and should be **IF_DEFAULT**. The remote end of the connection is specified by **remip** and **port**. The following table explains the possible combinations and what they mean.

REMIP	Effect of REMIP value
0	The connection completes when the first datagram is received, supplying both the remote IP address and the remote port number. Only datagrams received from that IP/port address will be accepted.
-1	All remote hosts can send datagrams to the socket. All outgoing datagrams will be sent to the broadcast address on the specified port.
>0	If the remote IP address is a valid IP address and the remote port is 0, the connection will complete when the first datagram is received, supplying the remote port number. If the remote IP address and the remote port are both specified when the function is called, the connection is complete at that point.

The **buffer** and **buflen** parameters allow a user to supply a socket buffer, instead of using a socket buffer from the pool.

If **remip** is non-zero, then the process of resolving the correct destination hardware address is started. Datagrams cannot be sent until **sock_resolved()** returns TRUE. If you attempt to send datagrams before this, then the datagrams may not get sent. The exception to this is if **remip** is -1 (broadcast) in which case datagrams may be sent immediately after calling this function.

udp_extopen (continued)

PARAMETERS

s	Pointer to socket's data structure.
iface	Local interface on which to open the socket (not yet implemented—use **IF_DEFAULT** for now).
lport	Local port.
remip	Acceptable remote IP, or 0 for all.
port	Acceptable remote port, or 0 for all.
datahandler	Function to call when data is received, **NULL** for placing data in the socket's receive buffer.
buffer	Address of user-supplied socket buffer in xmem. If **buffer** is 0, the socket buffer for this socket is pulled from the buffer pool defined by the macro **MAX_UDP_SOCKET_BUFFERS**.
buflen	Length of user-supplied socket buffer.

RETURN VALUE:

!0: Success.
0: Failure; error opening socket, e.g., a buffer could not be allocated.

LIBRARY

UDP.LIB

SEE ALSO

udp_open, sock_resolved

udp_open

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

DESCRIPTION

This function opens a UDP socket on the given local port (**lport**). The remote end of the connection is specified by **remip** and **port**. The following table explains the possible combinations and what they mean.

Table 1.

REMIP	Effect of REMIP value
0	The connection completes when the first datagram is received, supplying both the remote IP address and the remote port number. Only datagrams received from that IP/port address will be accepted.
-1	All remote hosts can send datagrams to the socket. All outgoing datagrams will be sent to the broadcast address on the specified port.
>0	If the remote IP address is a valid IP address and the remote port is 0, the connection will complete when the first datagram is received, supplying the remote port number. If the remote IP address and the remote port are both specified when the function is called, the connection is complete at that point.

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

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 **remip** is set to -1. Essentially, the broadcast and nonbroadcast protocols cannot co-exist.

Be sure that you have allocated enough UDP socket buffers with **MAX_UDP_SOCKET_BUFFERS**. Note that this macro defaults to 0, so any usage of **udp_open()** requires a definition of **MAX_UDP_SOCKET_BUFFERS** in your program.

PARAMETERS

s	Pointer to a UDP socket.
lport	Local port
remip	Acceptable remote IP, 0 to connect on first datagram, or -1 for broadcast.
port	Acceptable remote port, or 0 to connect on first datagram.
datahandler	Function to call when data is received. **NULL** for placing data in the socket's receive buffer.

udp_open (continued)

RETURN VALUE

0: Failure (e.g., a buffer could not be allocated).
!0: Success.

LIBRARY

UDP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

udp_extopen

udp_peek

```
int udp_peek(udp_Socket* s, _udp_datagram_info * udi);
```

DESCRIPTION

Look into the UDP socket receive buffer to see if there is a datagram ready to be read using **udp_recvfrom()**. This function does not remove the datagram from the buffer, but it allows the application to determine the full details about the next datagram, including whether the datagram was broadcast.

The returned data is put in ***udi**. **udi** must point to a valid data structure, or be **NULL**. The data structure is:

```
typedef struct {
  longword remip;    // Remote host IP address
  word     remport;  // Remote host port number
  int      len;      // Length of datagram
  byte     flags;    // Flags
  byte     iface;    // Interface number
} _udp_datagram_info;
```

The **flags** field may have one of the following values:

UDI_ICMP_ERROR	This is an ICMP error entry.
UDI_TOS_MASK	Type-of-service bit mask.
UDI_BROADCAST_LL	Received on broadcast link-layer address.
UDI_BROADCAST_IP	Received on broadcast network (IP) address.

PARAMETERS

s	UDP socket to check
udi	Where to store the returned information.

RETURN VALUE

1: A normal datagram is in the receive buffer.
0: No datagram waiting.
-3: An ICMP error message is in the receive buffer - this will only be returned if **udi** parameter is not **NULL**.

LIBRARY

UDP.LIB

SEE ALSO

udp_recvfrom

udp_recv

```
int udp_recv(udp_Socket* s, char* buffer, int len)
```

DESCRIPTION

Receives a single UDP datagram on a UDP socket. If the buffer is not large enough for the datagram, the datagram is truncated, and the remainder discarded.

PARAMETERS

s	Pointer to socket's data structure.
buffer	Buffer where the UDP datagram will be stored.
len	Maximum length of the buffer.

RETURN VALUE

≥**0**: Number of bytes received.
-1: No datagram waiting.
<-1: Error.

LIBRARY

UDP.LIB

SEE ALSO

udp_recvfrom, udp_send, udp_sendto, udp_open

udp_recvfrom

```
int udp_recvfrom(udp_Socket* s, char* buffer, int len,
   longword* remip, word* remport);
```

DESCRIPTION

Receive a single UDP datagram on a UDP socket. remip and remport should be pointers to the locations where the remote IP address and remote port from which the datagram originated are placed. If the buffer is not large enough for the datagram, then the datagram will be truncated, with the remainder being discarded.

If and only if the **UDP_MODE_ICMP** or **UDP_MODE_DICMP** modes are set for this socket, then a return code of -3 indicates that an ICMP error message is being returned in the buffer instead of a normal datagram. In this case, **buffer** will contain fixed data in the form of a structure of type **_udp_icmp_message**. The definition of this structure is:

```
typedef struct {
word myport;          // Originating port on this host
byte icmp_type;       // One of the ICMPTYPE_* values
byte icmp_code;       // The corresponding ICMP code
} _udp_icmp_message;
```

Please see **sock_mode** for more information about the modes **UDP_MODE_ICMP** and **UDP_MODE_DICMP**.

PARAMETERS

s	Pointer to socket's data structure.
buffer	Buffer where the UDP datagram will be stored.
len	Maximum length of the buffer.
remip	IP address of the remote host of the received datagram.
remport	Port number of the remote host of the received datagram.

RETURN VALUE

≥**0**: Number of bytes received.
-1: No datagram waiting.
-2: Error - not a UDP socket.
-3: The returned buffer contains an ICMP error which was queued previously.

LIBRARY

UDP.LIB

SEE ALSO

udp_recv, udp_send, udp_sendto, udp_open, udp_peek

udp_send

```
int udp_send(udp_Socket* s, char* buffer, int len);
```

DESCRIPTION

Sends a single UDP datagram on a UDP socket. It will not work for a socket for which the **remip** parameter to **udp_open()** was 0, unless a datagram has first been received on the socket. If the **remip** parameter to **udp_open()** was -1, the datagram will be send to the broadcast address.

PARAMETERS

s	Pointer to socket's data structure.
buffer	Buffer that contains the UDP datagram
len	Length of the UDP datagram.

RETURN VALUE

≥**0**: Number of bytes sent.
-1: Failure.

LIBRARY

UDP.LIB

SEE ALSO

udp_sendto, udp_recv, udp_recvfrom, udp_open

udp_sendto

```
int udp_sendto(udp_Socket* s, char* buffer, int len, longword
   remip, word remport);
```

DESCRIPTION

Sends a single UDP datagram on a UDP socket. It will send the datagram to the IP address and port specified by **remip** and **remport**. Note that this function can be used on a socket that has been "connected" to a different remote host and port.

PARAMETERS

s	Pointer to socket's data structure.
buffer	Buffer that contains the UDP datagram.
len	Length of the UDP datagram.
remip	IP address of the remote host.
remport	Port number of the remote host.

RETURN VALUE

≥**0**: Success, number of bytes sent.
-1: Failure.
-2: Failed because hardware address not resolved.

LIBRARY

UDP.LIB

SEE ALSO

udp_send, udp_recv, udp_recvfrom, udp_open

udp_waitopen

```
int udp_waitopen( udp_Socket *s, int iface, word lport, longword
   remip, word port, dataHandler_t datahandler, long buffer, int
   buflen, longword millisecs );
```

DESCRIPTION

This function is identical to **udp_extopen()**, except that it waits a specified amount of time for the hardware address of the destination to be resolved.

While waiting, this function calls **tcp_tick()**.

PARAMETERS

s	Pointer to socket.
iface	Local interface on which to open the socket (not yet implemented so use **IF_DEFAULT** for now).
lport	Local port.
remip	Acceptable remote ip, or 0 for all.
port	Acceptable remote port, or 0 for all.
datahandler	Function to call when data is received, **NULL** for placing data in the sockets receive buffer.
buffer	Address of user-supplied socket buffer in xmem, 0 to use a buffer from the socket buffer pool.
buflen	Length of user-supplied socket buffer.
millisecs	Maximum milliseconds to wait for the hardware address to be resolved.

RETURN VALUE

>0: Successfully opened socket.
0: Timed out without resolving address.
-1: Error opening socket (e.g., buffer could not be allocated).

LIBRARY

UDP.LIB

SEE ALSO

udp_extopen, sock_resolved

udp_waitsend

```
int udp_waitsend(udp_Socket* s, char* buffer, int len, longword
   remip, word remport, word millisecs);
```

DESCRIPTION

This is identical to **udp_sendto()**, except that it will block for up to the specified amount of time waiting for the hardware address to be resolved. Normally, you should not have to specify more than 100ms for the time out. If it takes longer than this, the destination is probably unavailable.

PARAMETERS

s	UDP socket on which to send the datagram.
buffer	Buffer that contains the UDP datagram.
len	Length of the UDP datagram.
remip	IP address of the remote host.
remport	Port number of the remote host.
millisecs	Number of milliseconds to wait for hardware address resolution. Reasonable values are between 50 and 750 milliseconds.

RETURN VALUE

≥**0**: Number of bytes sent.
-1: Failure (invalid UDP socket etc.).
-2: Failure (timed out, no datagram sent).

LIBRARY

UDP.LIB

SEE ALSO

udp_sendto, udp_recvfrom, udp_bypass_arp

2.9 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 for DNS lookups. It defaults to 4. If libraries such as **HTTP.LIB** or **FTP_SERVER.LIB** are used, you must provide enough sockets in **MAX_SOCKETS** for them also. This macro has been replaced by **MAX_TCP_SOCKET_BUFFERS** and **MAX_UDP_SOCKET_BUFFERS**.

MAX_SOCKET_LOCKS

For µC/OS-II support. This macro defines the number of socket locks to allocate. It defaults to **MAX_TCP_SOCKET_BUFFERS** + **MAX_UDP_SOCKET_BUFFERS**.

This macro is necessary because we can no longer calculate the number of socket locks needed based on the number of socket buffers, now that the user can manage their own socket buffers.

MAX_TCP_SOCKET_BUFFERS

Starting with Dynamic C version 7.05, this macro determines the maximum number of TCP sockets with preallocated buffers. If **MAX_SOCKETS** is defined, then **MAX_TCP_SOCKET_BUFFERS** will be assigned the value of **MAX_SOCKETS** for backwards compatibility. If neither macro is defined, **MAX_TCP_SOCKET_BUFFERS** defaults to 4.

MAX_UDP_SOCKET_BUFFERS

Starting with Dynamic C version 7.05, this macro determines the maximum number of UDP sockets with preallocated buffers. It defaults to 0.

MY_DOMAIN

This macro is the initial value for the domain portion of the controller's address. At runtime, it can be overwritten by **tcp_config()** and **setdomainname()**.

MY_GATEWAY

This macro gives the default value for the controllers default gateway. At runtime, it can be overwritten by **tcp_config()**.

MY_IP_ADDRESS

This macro is the default IP address for the controller. At runtime, it can be overwritten by **tcp_config()** and **sethostid()**.

MY_NAMESERVER

This macro is the default value for the primary name server. At runtime, it can be overwritten by **tcp_config()**.

MY_NETMASK

This macro is the default netmask for the controller. At runtime, it can be overwritten by **tcp_config()**.

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. This macro has been replaced by **TCP_BUF_SIZE** and **UDP_BUF_SIZE**.

TCP_BUF_SIZE

Starting with Dynamic C 7.05, TCP and UDP socket buffers are sized separately. **TCP_BUF_SIZE** defines the buffer sizes for TCP sockets. It defaults to 4096 bytes. Backwards compatibility exists with earlier version of Dynamic C: if **SOCK_BUF_SIZE** is defined, **TCP_BUF_SIZE** is assigned the value of **SOCK_BUF_SIZE**. If **SOCK_BUF_SIZE** is not defined, but **tcp_MaxBufSize** is, then **TCP_BUF_SIZE** will be assigned the value of **tcp_MaxBufSize***2.

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**.

In Dynamic C versions 6.56 and earlier, **tcp_MaxBufSize** determines the size of the input and output buffers for TCP and UDP sockets. The **sizeof(tcp_Socket)** will be about 200 bytes more than double **tcp_MaxBufSize**. The optimum value for local Ethernet connections is greater than the Maximum Segment Size (MSS). The MSS is 1460 bytes. You may want to lower **tcp_MaxBufSize**, which defaults to 2048 bytes, to reduce RAM usage. It can be reduced to as little as 600 bytes.

tcp_MaxBufSize will work slightly differently in Dynamic C versions 6.57 and higher. In these later versions the buffer for the UDP socket will be **tcp_MaxBufSize*2**, which is twice as large as before.

UDP_BUF_SIZE

Starting with Dynamic C 7.05, TCP and UDP socket buffers are sized separately. **UDP_BUF_SIZE** defines the buffer sizes for UDP sockets. It defaults to 4096 bytes. Backwards compatibility exists with earlier version of Dynamic C: if **SOCK_BUF_SIZE** is defined, **UDP_BUF_SIZE** is assigned the value of **SOCK_BUF_SIZE**. If **SOCK_BUF_SIZE** is not defined, but **tcp_MaxBufSize** is, then **UDP_BUF_SIZE** will be assigned the value of **tcp_MaxBufSize***2.

3. Server Utility Library

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 or sometimes the server spec 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 or sometimes just users list.

3.1.3 FormVar Structure

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

- A **server_spec** index that references the variable to be modified. This is the location of the form variable in the 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      // Indicates an unused entry
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.
SERVER_WRITABLE   //  server will allow client(s) to write to its files.
```

3.2.4 Configuration Macros

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. Strings must include a null character, so with its default value of 20, strings in this structure may be at most 19 characters long

SAUTH_MAXUSERS

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 192 for an example program.

3.4 Function Reference

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

sauth_adduser

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

DESCRIPTION

Adds a user to the TCP/IP users list.

PARAMETERS

username	Name of the user.
password	Password of the user.
servermask	Bitmask representing valid servers (e.g. **SERVER_HTTP**, **SERVER_FTP**).

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_removeuser

sauth_authenticate

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

DESCRIPTION

Authenticate a user.

PARAMETERS

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

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_getuserid

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

DESCRIPTION

Gets the user index for a user.

PARAMETERS

username	User's name.
server	Server for which we are looking up.

RETURN VALUE

≥**0**: Success, index of user in the TCP/IP users list.
-1: Failure.

LIBRARY

ZSERVER.LIB

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.

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_removeuser

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

DESCRIPTION

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

PARAMETERS

userid Index in TCP/IP users list.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sauth_adduser

sauth_setpassword

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

DESCRIPTION

Sets the password for a user.

PARAMETERS

userid Index of user in TCP/IP users list.

password User's new password

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sauth_setwriteaccess

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

DESCRIPTION

Sets the write accessibility of a user.

PARAMETERS

sauth	Index of the user in the TCP/IP users list.
writeaccess	Set to **1** to give write access, **0** 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. Make sure that **SSPEC_MAXSPEC** is large enough to hold this new entry. This function is currently only useful for the HTTP server.

PARAMETERS

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

RETURN VALUE

≥**0**: Success; location of form in 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. Make sure that **SSPEC_MAXSPEC** is large enough to hold this new entry.

PARAMETERS

name	Name of the new file.
filenum	Number of the file in the file system.
servermask	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. Make sure that **SSPEC_MAXSPEC** is large enough to hold this new entry. This function is currently only useful for HTTP servers.

PARAMETERS

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

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

form	Index of the form in the TCP/IP servers' object list.
var	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. Make sure that **SSPEC_MAXSPEC** is large enough to hold this new entry.

PARAMETERS

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

RETURN VALUE

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

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_addfsfile, 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. Make sure that **SSPEC_MAXSPEC** is large enough to hold this new entry. This function is currently only useful for the HTTP server.

PARAMETERS

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

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. Make sure that **SSPEC_MAXSPEC** is large enough to hold this new entry.

PARAMETERS

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

RETURN VALUE

-1: Failure.

≥**0**: Success, the location of the file in the TCP/IP servers' object list.

LIBRARY

ZSERVER.LIB

SEE ALSO

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

sspec_addxmemvar

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

DESCRIPTION

Add a variable located in extended memory to the TCP/IP servers' object list. Make sure that **SSPEC_MAXSPEC** is large enough to hold this new entry. Currently, this function is useful only for the HTTP server.

PARAMETERS

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

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. Make sure that **SSPEC_MAXSPEC** is large enough to hold this new entry. 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

sspec	Location of the object in the TCP/IP servers' object list that will be aliased.
name	Name field of the **ServerSpec** 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

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

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: Failure.
!NULL: 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: Failure.
!NULL: Success, pointer to requested function.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_addfunction

sspec_getfvdesc

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

NULL: Failure.
!NULL: 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

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

RETURN VALUE

-1: Failure;
Type of form entry element on success:
HTML_FORM_TEXT is a text box.
HTML_FORM_PULLDOWN is a pull-down menu.

LIBRARY

ZSERVER.LIB

sspec_getfvlen

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

DESCRIPTION

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

PARAMETERS

form	**server_spec** index of the form.
var	Index (into the **FormVar** 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

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

RETURN VALUE

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

LIBRARY

ZSERVER.LIB

sspec_getfvnum

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

DESCRIPTION

Gets the number of variables in a form.

PARAMETERS

form	**server_spec** 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

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

RETURN VALUE

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

LIBRARY

ZSERVER.LIB

sspec_getfvoptlistlen

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

DESCRIPTION

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

PARAMETERS

form	**server_spec** index of the form.
var	Index (into the **FormVar** 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

form	**server_spec** index of the form.
var	Index (into the **FormVar** 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

form	**server_spec** index of the form.
var	Index into the array of **FormVar** 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_getpreformfunction

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

DESCRIPTION

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

PARAMETERS

form spec index of the form

RETURN VALUE

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

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_setpreformfunction, sspec_setformfunction

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 124). 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: Failure.
!NULL: Success, pointer to variable.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_readvariable

sspec_getvarkind

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

0: Failure.
INT8|INT16|INT32|FLOAT32|PTR16: Success.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getvaraddr, sspec_getvartype, sspec_gettype

sspec_getvartype

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

DESCRIPTION

Gets the type of the variable in the TCP/IP servers' object list.

PARAMETERS

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

RETURN VALUE

SSPEC_ERROR: Failure.
SSPEC_ROOTVAR or **SSPEC_XMEMVAR**: Success.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getvaraddr, sspec_getvarkind, sspec_gettype

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

sspec	Index into the array of **ServerSpec** structures.
buffer	The buffer to put the file contents into.
offset	The offset from the start of the file, in bytes, at which copying should begin.
len	The number of bytes to copy.

RETURN VALUE

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

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getlength, sspec_getfileloc

sspec_readvariable

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

DESCRIPTION

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

PARAMETERS

sspec	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

form	Index into the array of **ServerSpec** structures.
function	Index into the array of **ServerSpec** structures. This is the return value of the function **sspec_addfunction()**.

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

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

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sspec_setformprolog

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

DESCRIPTION

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

PARAMETERS

form	Index into the array of **ServerSpec** structures.
function	Index into the array of **ServerSpec** structures. This is the return value of **sspec_addfunction()**.

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

form	**server_spec** index of the form.
title	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

form	**server_spec** index of the form.
var	Index (into the **FormVar** array) of the variable.
varcheck	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

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

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sspec_setfventrytype

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sspec_setfvfloatrange

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

DESCRIPTION

Sets the range of a float.

PARAMETERS

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

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

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

form	**server_spec** index of the form.
var	Index (into the **FormVar** array) of the variable.
len	Length of the variable.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sspec_setfvname

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

DESCRIPTION

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

PARAMETERS

form	**server_spec** index of the form.
var	Index (into the **FormVar** array) of the variable.
name	Display name of the variable.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sspec_setfvoptlist

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

DESCRIPTION

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

PARAMETERS

form	**server_spec** index of the form.
var	Index (into the **FormVar** array) of the variable.
list[]	Array of string values that the variable can assume.
listlen	Length of the array.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sspec_setfvrange

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

DESCRIPTION

Sets the range of an integer.

PARAMETERS

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

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sspec_setfvreadonly

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

DESCRIPTION

Sets the form variable to be read-only.

PARAMETERS

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

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

sspec_setpreformfunction

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

DESCRIPTION

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

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

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

PARAMETERS

form	Spec index of the form.
fptr	Pointer to user function to be called just before form generation

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getpreformfunction

sspec_setrealm

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

DESCRIPTION

Sets the realm field of a **ServerSpec** structure for HTTP authentication purposes. Setting this field enables authentication for the given entry in the TCP/IP servers' object list. Authentication can be turned off again by passing "" as the realm parameter to this function.

PARAMETERS

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

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sspec_getrealm

sspec_setsavedata

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

DESCRIPTION

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

PARAMETERS

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

RETURN VALUE

0: 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

sspec	Index into the array of **ServerSpec** structures.
uid	Index into the array of **ServerAuth** structures (identifies user).

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

ZSERVER.LIB

SEE ALSO

sauth_adduser, sspec_getusername

4. HTTP Server

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

type This field tells the server if the entry is a file, variable or function (**HTTPSPEC_FILE**, **HTTPSPEC_VARIABLE** or **HTTPSPEC_FUNCTION**, respectively).

name 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 **type** (the first field).

data The third field is the physical address of the entity.

addr 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.

vartype This field describes the type of variable. Supported types are : **INT8** **INT16**, **PTR16**, **INT32**, and **FLOAT32**.

format The format field describes the **printf** format specifier used to display the variable.

realm 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. Users can override **HTTP_MAXNAME** (which defaults to 20 characters) in their source file. If the function pointer given is **NULL**, then the default handler (which sends the content verbatim) is used.

```
typedef struct {
    char extension[10];
    char type[HTTP_MAXNAME];
    int  (*fptr)(/* HttpState* */);
} 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. The field that are available for use are described in Section 4.1.4.1 *HttpState Fields.*

```
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, 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 abort_notify;
    char cancel;
    char username[HTTP_MAXNAME];
    char password[HTTP_MAXNAME];
    char cookie[HTTP_MAXNAME];
    int headerlen;
    int headeroff;
    /* other - don't touch */
    char tag[HTTP_MAXNAME];
    char value[HTTP_MAXNAME];

   /* Support for conditional SSI (error feedback etc) */
   int cond[HTTP_MAX_COND];     /* Condition numbers (default 0) */
   /*  Optional User Data. Cleared on every new connection. */
#ifdef HTTP_USERDATA_SIZE
    char userdata[ HTTP_USERDATA_SIZE];
#endif
} HttpState;
```

4.1.4.1 HttpState Fields

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

s
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.

substate
subsubstate
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 **http_handler()** call, in which the CGI function will be called again. These variables are initialized to 0 before the CGI function is called for the first time. Hence, the first state of a state machine using substate should be 0.

timeout
This value can be used by the CGI function to implement an internal timeout.

main_timeout
This value holds the timeout that is used by the web server. The web server checks against this timeout on every call of **http_handler()**. When the web server changes states, it resets **main_timeout**. 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:
state->main_timeout = set_timeout(HTTP_TIMEOUT);
HTTP_TIMEOUT is the number of seconds until the web server will time out. It is 16 seconds by default.

buffer[]
A buffer that the developer can use to put data to be transmitted over the socket. It is of size **HTTP_MAXBUFFER**.

p
Pointer into the buffer given above.

method
This should be treated as read-only. It holds the method by which the web request was submitted. The value is either **HTTP_METHOD_GET** or **HTTP_METHOD_POST**, for the GET and POST request methods, respectively.

url[]
This should be treated as read-only. It holds the URL by which the current web request was submitted. . If there is GET-style form information, then that information will follow the first **NULL** byte in the url array. The form information will itself be **NULL**-terminated. If the information in the url array is truncated to **HTTP_MAXURL** bytes, the truncated information is also **NULL**-terminated.

version This should be treated as read-only. This holds the version of the HTTP request that was made. It can be **HTTP_VER_09**, **HTTP_VER_10**, or **HTTP_VER_11** for 0.9, 1.0, or 1.1 requests, respectively.

content_type[] This should be treated as read-only. This buffer holds the value from the Content-Type header sent by the client.

content_length This should be treated as read-only. This variable holds the length of the content sent by the client. It matches the value of the Content-Length header sent by the client.

has_form This should be treated as read-only. If the value is 1 there is a GET style form, after the \0 byte in **url[]**.

abort_notify Set to **!0** in user-defined formprolog function to indicate that the formepilog function needs to be called on an abort condition. If the epilog function is reached normally, this field must be set to zero. This prevents the formepilog function from being called one more time on a connection abort.

cancel This should be treated as read-only. It is intended for when the user-defined functions, which may be called before and after an HTML form is submitted, are used for locking resources.

If the formprolog function was called and then the connection is aborted before the formepilog function can be called, **cancel** is set to 1 and the formepilog function is called exactly once. If the epilog function was already called but returned zero (not finished yet), then it is called again if the connection is aborted, except if **cgi_redirectto()** has been called from the epilog function. In that case the epilog function is not called after an abort.

username[] Read-only buffer has username of the user making the request, if authentication took place.

password[] Read-only buffer has password of the user making the request, if authentication took place.

cookie[] Read-only buffer contains the value of the cookie "DCRABBIT" (see **http_setcookie()** for more information).

headerlen
headeroff These variables can be used together to cause the web server to flush data from the **buffer[]** array in the **HttpState** structure. **headerlen** should be set to the amount of data in **buffer[]**, and **headeroff** should be set to 0 (to indicate the offset into the array). The next time the CGI function is called the data in **buffer[]** will be flushed to the socket.

cond[] Support for conditional SSI (error feedback etc).

userdata[]	This field is included if **HTTP_USERDATA_SIZE** is defined. It is an optional user data area. The area is cleared to zero when the structure is initialized, otherwise it is not touched. Its size must be greater than zero.

4.2 Configuration Macros

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

HTTP_MAX_COND

Support for conditional SSI (error feedback etc). It defaults to 4.

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.

HTTP_PORT

This macro allows the user to override the default port. Define it before the line **#use http.lib**.

HTTP_USERDATA_SIZE

Defining this macro causes "char userdata[]" to be added to the HttpState structure. Define your structure before **#use http.lib**

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

In your own code, access it like:

```
mystate = (struct UserStateData *) state->userdata;
```

TIMEZONE

This macro specifies the distance in hours you are from Coordinated Universal Time (UTC), 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.

4.2.1 Customizing HTTP headers

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

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

state	Pointer to the state structure for the calling web server.
buffer	The buffer in which the header(s) can be written.
bytes	The number of bytes available in the buffer.

Typically, the macro would be defined by the user before **http.lib** is used, like in the following:

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

Then, for the above to work, the **my_headers()** function must be defined by the user, like the following:

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

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

4.3 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.

Program Name: `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**. This program displays four 'lights' and four buttons to toggle them. Users can browse to the device and change the status of the lights.

Program Name: samples/tcpip/http/ssi.c

```
#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"

#memmap 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];
```

Program Name: ssi.c (continued)

```
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;
}
```

Program Name: ssi.c (continued)

```
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.

For increased flexibility, a CGI function is responsible for outputting its own HTTP headers. Information about HTTP headers can be found at:

http://deesse.univ-lemans.fr:8003/Connected/RFC/1945/

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.

4.3.2.2.1 Connection Abort Condition

There are two fields in the **HttpState** structure that allow a CGI function to appropriately respond to a connection abort condition. The user may set the field **abort_notify** to **!0** in a CGI function to request that the CGI function be called one more time with the **cancel** field set to one if a connection abort occurs.

4.3.3 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 187, 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 the following example, in the first two cases , it is the text input field, which is a single-line text entry box. The NAME parameter defines what the name of that particular input variable is, so that when the information is returned to the server, then the server can associate it with a particular variable. The VALUE parameter defines the current value of the parameter. The SIZE parameter defines how long the text entry box is (in characters).

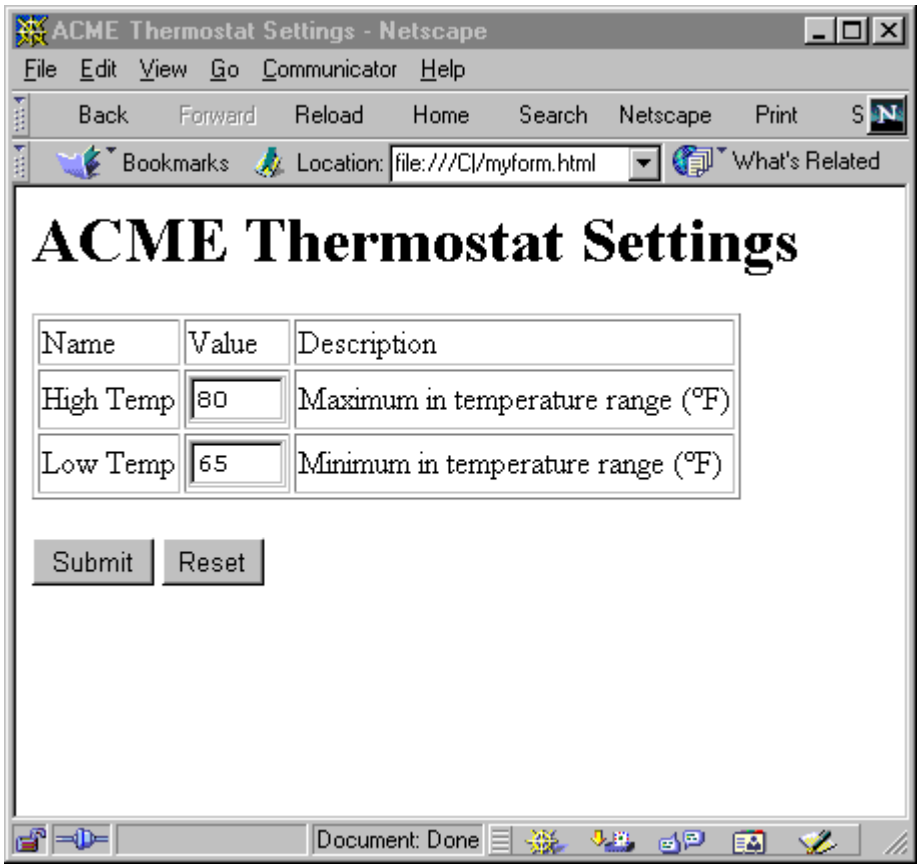
At the end of the HTML page in our example, the Submit and Reset buttons are defined with the INPUT tag. These use the special types “submit” and “reset”, since these buttons have special purposes. When the submit button is pressed, the form is submitted by calling the CGI program “**`myform`**”.

4.3.3.1 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[]`**. These code snippets are from **`/samples/tcpip/http/post.c`**.

```
/* 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';
       }
    }
}
```

```
/* Parse the url-encoded POST data into FORMSpec structure */
/* e.g., parse 'foo=bar&baz=qux' into the struct) */
int parse_post(HttpState* state) {
    auto int retval;
    while(1) {
       retval = sock_fastread(&state->s, state->p, 1);
       if(0 == retval) {
          *state->p = '\0';
          parse_token(state);
          return 1
       }
       /* should this only be '&'? (allow the eoln as valid text?) */
       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)
          return 1;         // input too long
    }
    return 0;               // end of data - loop again to give it time to write more
}
```

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. This function is from **`/samples/tcpip/http/post.c`**.

```
int submit(HttpState* state){
   auto int i;
   if(state->length) {           // buffer to write out
      if(state->offset < state->length) {
         state->offset += sock_fastwrite(&state->s, state->buffer +
          (int)state->offset,(int)state->length - (int)state -> offset);
      } else {
         state->offset = 0;
         state->length = 0;
      }
   } else {
      switch(state->substate) {
      case 0:
         strcpy(state->buffer, "HTTP/1.0 200 OK\r\n\r\n");
         state->length = strlen(state->buffer);
         state->offset = 0;
         state->substate++;
         break;
      case 1:
         strcpy(state->buffer,"<html><head><title>Results</title>
          </head><body>\r\n");
         state->length = strlen(state->buffer);
         state->substate++;
         break;
      case 2:                      // init the FORMSpec data
         FORMSpec[0].value[0] = '\0';
         FORMSpec[1].value[0] = '\0';
         state->p = state->buffer;
         state->substate++;
         break;
      case 3:                      // parse the POST information
         if(parse_post(state)) {
            sprintf(state->buffer, "<p>Username: %s<p>\r\n<p>Email:
              %s<p>\r\n", FORMSpec[0].value, FORMSpec[1].value);
            state->length = strlen(state->buffer);
            state->substate++;
         }
         break;
      case 4:
         strcpy(state->buffer,"<p>Go <a href=\"/\">home</a></body>
          </html>\r\n");
         state->length = strlen(state->buffer);
         state->substate++;
         break;
      default:
         state->substate = 0;
         return 1;
      }
   }
   return 0;
}
```

4.3.4 HTML Forms Using Zserver.lib

In this section, we will step through an example program, **/samples/tcpip/http/form1.c**, that uses HTML forms. Through this step-by-step explanation, the method of using the functions in **zserver.lib** will become 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.

```
var = sspec_addvariable("failure", fail, PTR16, "%s", SERVER_HTTP);
var = sspec_addfv(form, var);
sspec_setfvname(form, var, "Failure Action");
sspec_setfvdesc(form, var, "Action to take in case of air-conditioning
                 failure");
sspec_setfvlen(form, var, 20);
```

This line associates an option list with a form variable. The third parameter gives the developer-defined option array, and the fourth parameter gives the length of the array. The form variable can now only take on values listed in the option list.

```
sspec_setfvoptlist(form, var, fail_options, 4);
```

This function sets the type of form element that is used to represent the variable. The default is **HTML_FORM_TEXT**, which is a standard text entry box. This line sets the type to **HTML_FORM_PULLDOWN**, which is a pull-down menu.

```
sspec_setfventrytype(form, var, HTML_FORM_PULLDOWN);
```

Finally, this code sets up the last variable. Note that it is a float, so **FLOAT32** is given in the **sspec_addvariable()** call. The last function call is **sspec_setfvfloatrange()** instead of **sspec_setfvrange()**, since this is a floating point variable.

```
var = sspec_addvariable("humidity", &humidity, FLOAT32, "%.2f",
                        SERVER_HTTP);
var = sspec_addfv(form, var);
sspec_setfvname(form, var, "Humidity");
sspec_setfvdesc(form, var, "Target humidity (between 0.0 and 1.0)");
sspec_setfvlen(form, var, 8);
sspec_setfvfloatrange(form, var, 0.0, 1.0);
```

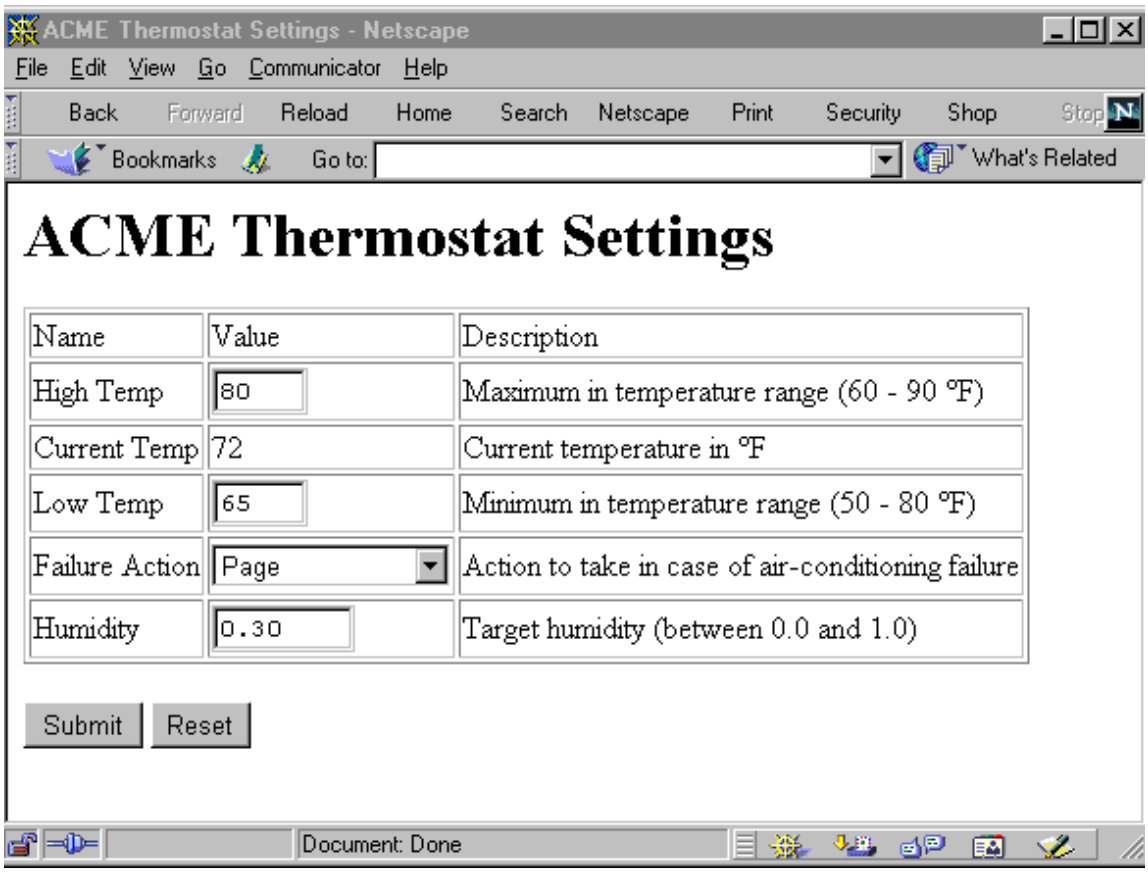
These calls create aliases in the TCP/IP servers' object list for the HTML form. That is, the same form can now be generated by requesting "**index.html**" or "**/**". Note that **sspec_aliasspec()** should be called after the form has already been set up. The aliasing is done by creating a new entry in the TCP/IP servers' object list and copying the original entry into the new entry. Note that aliasing can also be done for files and other types of server objects.

```
sspec_aliasspec(form, "index.html");
sspec_aliasspec(form, "/");
```

These lines complete the sample program. They initialize the TCP/IP stack and web server, and run the web server.

```
  sock_init();
  http_init();
  while (1) {
      http_handler();
  }
}
```

This is the form that is generated:


ACME Thermostat Settings - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Go to: What's Related
ACME Thermostat Settings
Name Value Description
High Temp 80 Maximum in temperature range (60 - 90 °F)
Current Temp 72 Current temperature in °F
Low Temp 65 Minimum in temperature range (50 - 80 °F)
Failure Action Page Action to take in case of air-conditioning failure
Humidity 0.30 Target humidity (between 0.0 and 1.0)
Submit Reset
Document: Done

4.4 Function Reference

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

state	Current server struct, as received by the CGI function.
url	Fully qualified URL to redirect to.

RETURN VALUE

None - sets the state, so the CGI must immediately return with a value of **0**.

LIBRARY

HTTP.LIB

SEE ALSO

cgi_sendstring

cgi_sendstring

```
void cgi_sendstring(HttpState* state, char* str);
```

DESCRIPTION

Sends a string to the user. You should immediately issue a "**return 0;**" after calling this function. The CGI is considered finished when you call this, and will be in an undefined state. This function greatly simplifies a CGI handler because it allows you to generate your page in a buffer, and then let the library handle writing it to the network.

PARAMETERS

state	Current server struct, as received by the CGI function.
str	String to send.

RETURN VALUE

None - sets the state, so the CGI must immediately return with a value of **0**.

LIBRARY

HTTP.LIB

SEE ALSO

cgi_redirectto

http_addfile

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

DESCRIPTION

Adds a file to the TCP/IP servers' object list.

PARAMETERS

name	Name of the file (e.g., "**/index.html**").
location	Address of the file data. (from **#ximport**)

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

HTTP.LIB

SEE ALSO

http_delfile

http_contentencode

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

DESCRIPTION

Converts a string to include HTTP transfer-coding "tokens" (such as @ (decimal) for at-sign) where appropriate. Encodes these characters: "<>@%#&"

Source string is **NULL-**byte terminated. Destination buffer is bounded by **len**. This function is reentrant.

PARAMETERS

dest	Buffer where encoded string is stored.
src	Buffer holding original string (not changed)
len	Size of destination buffer.

RETURN VALUE

dest: There was room for all conversions.
NULL: Not enough room.

LIBRARY

HTTP.LIB

SEE ALSO

http_urldecode

http_date_str

```
char *http_date_str(char *buf);
```

DESCRIPTION

Print the date (time zone adjusted) into the given buffer. This assumes there is room!

PARAMETERS

buf	The buffer to write the date into. This requires at least 30 bytes in the destination buffer.

RETURN VALUE

A pointer to the string.

LIBRARY

HTTP.LIB

SEE ALSO

http_handler

http_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_findname

```
HttpSpecAll http_findname(char* name);
```

DESCRIPTION

Finds a spec entry, searching first in RAM, then in FLASH.

PARAMETERS

name Name, in text, of the spec to find.

RETURN VALUE

The spec entry.

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_nextfverr

```
void http_nextfverr( char* start, char** name, char** value,
   int* error, char** next );
```

DESCRIPTION

Gets the information for the next variable in the HTML form error buffer. If any of the last four parameters in the function call are **NULL**, then those parameters will not have a value returned. This is useful if you are only interested in certain variable information.

PARAMETERS

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

RETURN VALUE

None, although information is returned in the last four parameters.

LIBRARY

HTTP.LIB

http_parseform

```
int http_parseform(int form, HttpState* state);
```

DESCRIPTION

Parses the returned form information. It expects a POST submission. This function is useful for a developer who only wants the parsing functionality and wishes to generate forms herself. Note that the developer must still build the array of **FormVar**s and use the **server_spec** table. This function will not, however, automatically display the form when used by itself. If all variables satisfy all integrity checks, then the variables' values are updated. If any variables fail, then none of the values are updated, and error information is written into the error buffer If this function is used directly, the developer must process errors.

PARAMETERS

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

RETURN VALUE

0 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

buf	Buffer to store cookie-generation header.
value	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 URL-escaped "tokens" (such as %20 (hex) for space) into actual values. Changes "+" into a space. String can be **NULL** terminated; it is also bounded by a specified string length. This function is reentrant.

PARAMETERS

dest	Buffer where decoded string is stored.
src	Buffer holding original string (not changed).
len	Maximum size of string (**NULL** terminated strings can be shorter).

RETURN VALUE

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

name	Name of the function (e.g., "**/foo.cgi**").
fptr	Function pointer to the handler, that must take **HttpState*** as an argument. This function should return an **int** (**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

name	Name of the variable.
variable	Pointer to the variable.
type	Type of variable. The following types are supported: **INT8**, **INT16**, **INT32**, **PTR16**, **FLOAT32**
format	Standard printf format string. (e.g., "%d")

RETURN VALUE

0: Success.
1: Failure (no room).

LIBRARY

HTTP.LIB

SEE ALSO

shtml_delvariable

shtml_delfunction

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

DESCRIPTION

Deletes a function from the 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

5. FTP Client

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

5.1 Configuration Macros

The following macros may be defined in a **#define** statement before the inclusion of **FTP_CLIENT.LIB** in an application program. Note that strings must contain the **NULL** byte, so if a maximum string length is 16, the maximum number of characters is 15.

FTP_MAX_DIRLEN
The default is 64, which is the maximum string length of a directory name.

FTP_MAX_FNLEN
The default is 16, which is the maximum string length of a file name.

FTP_MAX_NAMELEN
The default is 16 which is the maximum string length of usernames and passwords.

FTP_MAXLINE
The default is 256, which is both the maximum command line length and data chunk size that can be passed between server and client.

FTP_TIMEOUT
The default is 16, which is the number of seconds that pass before a time out occurs.

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. Failure can occur if the host address is zero, if **length** is negative, or if the internal control socket to the FTP server cannot be opened (e.g. because of lack of socket buffers).

PARAMETERS

host	Host IP address of FTP server.
port	Port of FTP server, 0 for default.
username	Username of account on FTP server.
password	Password of account on FTP server.
mode	Mode of transfer: **FTP_MODE_UPLOAD** or **FTP_MODE_DOWNLOAD**. You may also OR in the value **FTP_MODE_PASSIVE** to use passive mode transfer (important if you are behind a firewall).
filename	Filename to get/put.
dir	Directory file is in, **NULL** for default directory.
buffer	Buffer to get/put the file from/to. Must be **NULL** if a data handler function will be used. See **ftp_data_handler()** for more details.
length	On upload, length of file; on download size of buffer. This parameter limits the transfer size to a maximum of 32767 bytes. For larger transfers, it will be necessary to use a data handler function.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

ftp_client_tick, ftp_data_handler

ftp_client_tick

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

DESCRIPTION

Tick function to run the FTP daemon. Must be called periodically. The return codes are not very specific. You can call **ftp_last_code()** to get the integer value of the last FTP message received from the server. See RFC959 for details. For example, code 530 means that the client was not logged in to the server.

RETURN VALUE

FTPC_AGAIN (0): still pending, call again.
FTPC_OK (1): success (file transfer complete).
FTPC_ERROR (2): failure (call **ftp_last_code()** for more details).
FTPC_NOHOST (3): failure (Couldn't connect to remote host).
FTPC_NOBUF (4): failure (no buffer or data handler).
FTPC_TIMEOUT (5): warning (Timed out on close: data may or may not be OK).
FTPC_DHERROR (6): error (Data handler error in **FTPDH_END** operation).

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

ftp_client_setup, ftp_client_filesize, ftp_client_xfer, ftp_last_code

ftp_client_filesize

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

DESCRIPTION

Returns the byte count of data transferred. This function is deprecated in favor of **ftp_client_xfer()**, which returns a long value.

If the number of bytes transferred was over 32767, then this function returns 32767 which may be misleading.

RETURN VALUE

Size, in bytes.

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

ftp_client_setup, ftp_data_handler, ftp_client_xfer

ftp_client_xfer

```
longword ftp_client_xfer(void);
```

DESCRIPTION

Returns the byte count of data transferred. Transfers of over 2^{32} bytes (about 4GB) are not reported correctly.

RETURN VALUE

Size, in bytes.

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

ftp_client_setup, ftp_data_handler, ftp_client_filesize

ftp_data_handler

```
void ftp_data_handler(int (*dhnd)(), void * dhnd_data, word
   opts);
```

DESCRIPTION

Sets a data handler for further FTP data transfer(s). This handler is only used if the "buffer" parameter to **ftp_client_setup()** is passed as **NULL**.

The handler is a function which must be coded according to the following prototype:

```
int my_handler(char * data, int len, longword offset, int
   flags, void * dhnd_data);
```

This function is called with **data** pointing to a data buffer, and **len** containing the length of that buffer. **offset** is the byte number relative to the first byte of the entire FTP stream. This is useful for data handler functions that do not wish to keep track of the current state of the data source. **dhnd_data** is the pointer that was passed to **ftp_data_handler()**.

flags contains an indicator of the current operation:

- **FTPDH_IN**: data is to be stored on this host (obtained from an FTP download).
- **FTPDH_OUT**: data is to be filled with the next data to upload to the FTP server.
- **FTPDH_END**: data and len are irrelevant: this marks the end of data, and gives the function an opportunity to e.g. close the file. Called after either in or out processing.
- **FTPDH_ABORT**: end of data; error encountered during FTP operation. Similar to END except the transfer did not complete. Can use this to e.g. delete a partially written file.

The return value from this function depends on the in/out flag. For **FTPDH_IN**, the function should return **len** if the data was processed successfully and download should continue; -1 if an error has occurred and the transfer should be aborted. For **FTPDH_OUT**, the function should return the actual number of bytes placed in the data buffer, or -1 to abort. If zero is returned, then the upload is terminated normally. For **FTPDH_END**, the return code should be zero for success or -1 for error. If an error is flagged, then this is used as the return code for **ftp_client_tick()**. For **FTPDH_ABORT**, the return code is ignored.

ftp_data_handler (continued)

PARAMETERS

dhnd	Pointer to data handler function, or **NULL** to remove the current data handler.
dhnd_data	A pointer which is passed to the data handler function. This may be used to point to any further data required by the data handler such as an open file descriptor.
opts	Options word (currently reserved, set to zero).

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

ftp_client_setup

ftp_last_code

```
int ftp_last_code(void);
```

DESCRIPTION

Returns the most recent message code sent by the FTP server. RFC959 describes the codes in detail. This function is most useful for error diagnosis in the case that an FTP transfer failed.

Return Value

Error code; a number between 0 and 999. Codes less than 100 indicate that an internal error occurred e.g. the server was never contacted.

LIBRARY

FTP_CLIENT.LIB

SEE ALSO

ftp_client_setup, ftp_client_tick

5.3 Sample FTP Transfer

Program Name: samples/tcpip/ftp/ftp_client.c

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

#memmap xmem
#use "dcrtcp.lib"
#use "ftp_client.lib"


#define REMOTE_HOST "10.10.6.19"
#define REMOTE_PORT 0

main() {
    char buf[2048];
    int ret, i, j;

    printf("Calling sock_init()...\n");
    sock_init();

/* Set up the ftp transfer. This is to the host defined above,
 with a normal anonymous/e-mail password login info. A get
 of the file "bar" is requested, which will be stored in
'buf.'*/

   printf("Calling ftp_client_setup()...\n");

   if(ftp_client_setup(resolve(REMOTE_HOST),REMOTE_PORT, anony-
     mous", "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,
         since we requested a download.*/

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

6. FTP Server

This chapter documents the FTP server. The following information is included:

- configuration macros.
- the default file handlers.
- how to assign replacement file handlers.
- what to do when there is a firewall.
- API functions.
- commands accepted by the server.
- reply codes generated by the server.
- sample code demonstrating a working FTP server.

The library **FTP_SERVER.LIB** implements the File Transfer Protocol for the server side of a connection. FTP uses two TCP connections to transfer a file. The FTP server does a passive open on well-known port 21 and then listens for a client. This is the command connection. The server receives commands through this port and sends reply codes. The second TCP connection is for the actual data transfer.

Anonymous FTP is supported. Most FTP servers on the Internet use the identifier “anonymous,” so since FTP clients expect it, this is the identifier that is recommended, but any string (with a maximum length of **SAUTH_MAXNAME**) may be used.

6.1 Configuration Macros

The configuration macros control various conditions of the server’s operation. Read through them to understand the default conditions. Any changes to these macros may be made in the server application with **#define** statements before inclusion of **FTPSERVER.LIB**.

FTP_EXTENSIONS

The macro is not defined by default. Define it to allow the server to recognize the **DELE**, **SIZE** and **MDTM** commands.

FTP_MAXSERVERS

The default is 1: the number of simultaneous connections the FTP server can support. Each server requires a significant amount of RAM (2500 bytes by default, though this can change through **SOCK_BUF_SIZE** or **tcp_MaxBufSize** (deprecated)).

FTP_MAXLINE

The default is 256: the number of bytes of the working buffer in each server. This is also the maximum size of each network read/write. The default value of 256 is the minimum value that allows the server to function properly.

FTP_NODEFAULTHANDLERS
This macro is undefined. Define it to eliminate the code for the default file handlers. You must then provide your own file handlers.

FTP_TIMEOUT
The default is 16: the number of seconds to wait for data from the remote host before terminating the connection. In a high-latency network this value may need to be increased to avoid premature closures. No one likes premature closures.

6.2 File Handlers

Default file handlers are provided. The defaults access the server spec list, which is set up using **sspec_addxmemfile()**, **sauth_adduser()** etc. (See Section 3.1.1 on page 123 for more information on the server spec list.) The default file handlers are used when **NULL** is passed to the initialization function **ftp_init()**.

6.2.1 Replacing the Default Handlers

The **FTPhandlers** structure contains function pointers to the file handlers. This structure may be passed to **ftp_init()** to replace the default file handlers. You must provide at least a dummy function for all handlers.

```
typedef struct {
   int (*open)();
   int (*read)();
   int (*write)();
   int (*close)();
   long (*getfilesize)();
   int (*dirlist)();
   int (*cd)();
   int (*pwd)();
#ifdef FTP_EXTENSIONS
   long (*mdtm)();
   int (*delete)();
#endif
} FTPhandlers;
```

6.2.2 File Handlers Specification

Detailed function descriptions for the default handlers are found here. Additional information is provided in these descriptions when the default handler does not cover the entire function specification.

The default file handlers are found in **FTPSERVER.LIB**.

ftp_dflt_open

```
int ftp_dflt_open(char *name, int options, int uid, int cwd);
```

DESCRIPTION

Opens a file. If a file is successfully opened, the returned value is passed to subsequent handler routines to identify the particular file or resource, as the 'fd' parameter. If necessary, you can use this number to index an array of any other state information needed to communicate with the other handlers. The number returned should be unique with respect to all other open resource instances, so that your handler does not get confused if multiple FTP data transfers are active simultaneously.

Note that the specified file to open may be an absolute or relative path: if the handler supports the concept of directories, then it should handle the path name appropriately and not just assume that the file is in the current directory. If the filename is relative, then the **cwd** parameter indicates the current directory.

PARAMETERS

name	The file to open.
options	File access options: **O_RDONLY** (marks file as read-only). **O_WRONLY** (not currently supported by the default handler). **O_RDWR** (not used since it's not supported by the FTP protocol).
uid	The userid of the currently logged-in user.
cwd	Current directory (not currently supported by the default handler).

RETURN VALUE

≥**0**: File descriptor of the opened file.
FTP_ERR_NOTFOUND: File not found.
FTP_ERR_NOTAUTH: Unauthorized user.
FTP_ERR_BADMODE: Requested option (2nd parameter) is not supported.
FTP_ERR_UNAVAIL: Resource temporarily unavailable.

ftp_dflt_getfilesize

```
long ftp_dflt_getfilesize(int fd);
```

DESCRIPTION

Return the length of the specified file. This is called immediately after open for a read file. If the file is of a known constant length, the correct length should be returned. If the resource length is not known (perhaps it is generated on-the-fly) then return -1. For write operations, the maximum permissible length should be returned, or -1 if not known.

PARAMETERS

fd	The file descriptor returned when the file was opened.

RETURN VALUE

≥**0**: The size of the file in bytes.
-1: The length of the file is not known.

ftp_dflt_read

```
int ftp_dflt_read(int fd, char *buf, long offset, int len);
```

DESCRIPTION

Read file identified by **fd**. The file contents at the specified offset should be stored into **buf**, up to a maximum length of **len**. The return value should be the actual number of bytes transferred, which may be less than **len**. If the return value is zero, this indicates normal end-of-file. If the return value is negative, then the transfer is aborted. Each successive call to this handler will have an increasing offset. If the getfilesize handler returns a non-negative length, then the read handler will only be called for data up to that length — there is no need for such read handlers to check for EOF since the server will assume that only the specified amount of data is available.

The return value can also be greater than **len**. This is interpreted as "I have not put anything in **buf**. Call me back when you (the server) can accept at least **len** bytes of data." This is useful for read handlers which find it inconvenient to retrieve data from arbitrary offsets, for example a log reader which can only access whole log records. If the returned value is greater than the server can ever offer, then the server aborts the data transfer. The handler should never ask for more than **FTP_MAXLINE** bytes.

PARAMETERS

fd	The file descriptor returned when the file was opened.
buf	Pointer to the buffer to place the file contents.
offset	Offset in the file at which copying should begin.
len	The number of bytes to read.

RETURN VALUE

0: EOF.
>**0**: The number of bytes read into **buf**.
-1: Error, transfer aborted.

ftp_dflt_write

```
int ftp_dflt_write(int fd, char *buf, long offset, int len);
```

DESCRIPTION

The default write handler does nothing but return zero.

The specification states that the handler may write the file identified by **fd**. **buf** contains data of length **len**, which is to be written to the file at the given offset within the file. The return value must be equal to **len**, or a negative number if an error occurs (such as out of space).

The FTP server does not handle partial writes: the given data must be completely written or not at all. If the return code is less than **len**, an error is assumed to have occurred. Note that it is up to the handler to ensure that another FTP server is not accessing a file which is opened for write. The open call for the other server should return **FTP_ERR_UNAVAIL** if the current server is writing to a file.

PARAMETERS

fd	The file descriptor returned when the file was opened.
buf	Pointer to the data to be written.
offset	Offset in the file at which to start.
len	The number of bytes to write.

RETURN VALUE

≥**0**: The number of bytes written. If this is less than **len**, an error occurred.
-1: Error.

ftp_dflt_close

```
int ftp_dflt_close(int fd);
```

DESCRIPTION

The default close handler does nothing but return zero.

The handler may close the specified file and free up any temporary resources associated with the transfer.

PARAMETERS

fd	The file descriptor returned when the file was opened.

RETURN VALUE

0

ftp_dflt_list

```
int ftp_dflt_list (int item, char *line, int listing, int uid,
   int cwd);
```

DESCRIPTION

Returns the next file for the FTP server to list. The file name is formatted as a string.

PARAMETERS

item	Index number starting at zero for the first function call. Subsequent calls should be one plus the return value from the previous call.
line	Pointer to location to put the formatted string.
listing	Boolean variable to control string form: **0**: print file name, permissions, date, etc. **1**: print file name only.
uid	The currently logged-in user.
cwd	The current working directory.

RETURN VALUE

≥**0**: File descriptor for last file listed.
-1: Error.

ftp_dflt_cd

```
int ftp_dflt_cd(int cwd, char * dir, int uid);
```

DESCRIPTION

Change to new "directory." This is called when the client issues a CWD command. The FTP server itself has no concept of what a directory is —this is meaningful only to the handler.

PARAMETERS

cwd	Integer representing the current directory.
dir	String that indicates the new directory that will become the current directory. The interpretation of this string is entirely up to the handler. The **dir** string will be passed as ".." to move up one level.
uid	The currently logged-in user.

RETURN VALUE

0: No such directory exists.
-1: Root directory.
>0: Anything that is meaningful to the handler.

ftp_dflt_pwd

```
int ftp_dflt_pwd(int cwd, char * buf);
```

DESCRIPTION

Print the current directory, passed as **cwd**, as a string. The result is placed in **buf**, whose length may be assumed to be at least (**FTP_MAXLINE**-6). The return value is ignored.

PARAMETERS

cwd	The current directory.
buf	Pointer to buffer to put the string.

RETURN VALUE

The return value is ignored.

ftp_dflt_mdtm

```
unsigned long ftp_dflt_mdtm(int fd);
```

DESCRIPTION

This handler function is called when the server receives the FTP command **MDTM**. The return value of this handler function is the number of seconds that have passed since January 1, 1980. A return value of zero will cause the reply code 213 followed by a space and then the value 19800101000000 (yyyymmddhhmmss) to be sent by the server.

The FTP server assumes that this return value is in UTC (Coordinated Universal Time). If **SEC_TIMER** is running in local time, the handler should make the necessary time zone adjustment so that the return value is expressed in UTC.

The handler is only recognized if **FTP_EXTENSIONS** is defined.

PARAMETERS

fd File descriptor for the currently opened file.

RETURN VALUE

The number of seconds that have passed since January 1, 1980. The default handler always returns zero. The number of seconds will be converted to a date and time value of the form yyyymmddhhmmss.

ftp_dflt_delete

```
int ftp_dflt_delete(char * name, int uid, int cwd);
```

DESCRIPTION

The default handler does not support the delete command. It simply returns the error code for an unauthorized user.

The delete handler is only recognized by the server if **FTP_EXTENSIONS** is defined. It is called when the **DELE** command is received. The given file name (possibly relative to **cwd**) should be deleted.

PARAMETERS

name	Pointer to the name of a file.
uid	The currently logged-in user.
cwd	The current directory.

RETURN VALUE

0: File was successfully deleted .
FTP_ERR_NOTFOUND: File not found.
FTP_ERR_NOTAUTH: Unauthorized user.
FTP_ERR_BADMODE: Requested option (2nd parameter) is not supported.
FTP_ERR_UNAVAIL: Resource temporarily unavailable.

6.3 Functions

The API functions described here, initialize and run the FTP server.

ftp_init

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

DESCRIPTION

Initializes the FTP server. You can optionally specify a set of handlers for controlling what the server presents to the client. This is done with function pointers in the FTPhandlers structure.

PARAMETERS

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

RETURN VALUE

None.

LIBRARY

FTP_SERVER.LIB

SEE ALSO

ftp_tick

ftp_set_anonymous

```
int ftp_set_anonymous(int uid);
```

DESCRIPTION

Set the "anonymous" user ID. Resources belonging to this userID may be accessed by any user. A typical use of this function would be

```
ftp_set_anonymous (sauth_adduser("anonymous", "",
   SERVER_FTP));
```

which defines an "anonymous" login for the FTP server. This only applies to the FTP server. The username "anonymous" is recommended, since most FTP clients use this for hosts which have no account for the user.

PARAMETER

uid	The user ID to use as the anonymous user. This should have been defined using **sauth_adduser()**. Pass -1 to set no anonymous user.

RETURN VALUE

Same as the **uid** parameter, except -1 if **uid** is invalid.

LIBRARY

FTP_SERVER.LIB

SEE ALSO

sauth_adduser

ftp_tick

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

DESCRIPTION

Once **ftp_init** has been called, **ftp_tick** must be called periodically to run the server. This function is non-blocking.

LIBRARY

FTP_SERVER.LIB

SEE ALSO

ftp_init

6.4 Sample FTP Server

This code demonstrates a simple FTP server, using the ftp library. The user "anonymous" may download the file "rabbitA.gif," but not "rabbitF.gif." The user "foo" (with password "bar") may download "rabbitF.gif," but also "rabbitA.gif," since files owned by the anonymous user are world-readable.

Program Name: samples/tcpip/ftp_server.c

```
#define MY_IP_ADDRESS "10.10.6.105"    // All fields in this section must be
#define MY_NETMASK "255.255.255.0"     //   changed to match local
#define MY_GATEWAY "10.10.6.19"        //   network settings.

#memmap xmem
#use "dcrtcp.lib"
#use "ftp_server.lib"
#ximport "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);
   ftp_set_anonymous(user);
   sspec_setuser(file, user);
   sspec_setuser(sspec_addxmemfile("test1", rabbit1_gif,
    SERVER_FTP), user);
   sspec_setuser(sspec_addxmemfile("test2", rabbit1_gif,
    SERVER_FTP), user);
   /* Set up the second file and user */
   file = sspec_addxmemfile("rabbitF.gif", rabbit1_gif,
    SERVER_FTP);
   user = sauth_adduser("foo", "bar", SERVER_FTP);
   sspec_setuser(file, user);
   sspec_setuser(sspec_addxmemfile("test3", rabbit1_gif,
    SERVER_FTP), user);
   sspec_setuser(sspec_addxmemfile("test4", rabbit1_gif,
    SERVER_FTP), user);
   sock_init();
   ftp_init(NULL);                 // use default handlers
   tcp_reserveport(FTP_CMDPORT);   // Port 21

   while(1) {
      ftp_tick();
   }
}
```

Each user may execute the "dir" or "ls" command to see a listing of the available files. The listing shows only the files that the logged-in user can access.

6.5 Getting Through a Firewall

If a client is behind a firewall, it is incumbent upon the client to request that the server do a passive open on its data port instead of the normal active open. This is so that the client can then do an active open using the passively opened data port of the server, thus getting through the firewall.

Typically the server would not be behind a firewall.

6.6 FTP Server Commands

The following commands are recognized by the FTP server. The reply codes sent in response to these commands are detailed in Section 6.7 on page 235. They are noted here to associate them with the commands that may cause them to be sent.

Table 2. Recognized FTP Server Commands

Command	Description	Possible Reply Codes
ABOR	The current data transfer completes before the abort command is read by the server.	226
CDUP	A special case of CWD (Change Working Directory); the parent of the working directory is changed to be the working directory.	250, 431
CWD	Changes working directory.	250, 431
DELE	Delete the specified file.	250, 450, 550
LIST	Displays list of files requested by its argument in ls -l format. This gives extra information about the file.	150, 226, 425
MDTM	Shows the last modification time of the specified file.	213, 250, 450, 550
MODE	Confirms the mode of data transmission. Only stream mode is supported.	200, 504
NLST	Displays list of files requested by its argument, with names only. This allows an application to further process the files.	150, 226, 425
NOOP	Specifies no action except that the server send an OK reply. It does not affect any parameters or previously entered commands.	200
PASV	Requests a passive open on a port that is not the default data port. The server responds with the host and port address on which it is listening.	227, 452
PORT	Changes the data port from the default port to the port specified in the command’s argument. The argument is the concatenation of a 32-bit internet host address and a 16-bit TCP port address.	200

Table 2. Recognized FTP Server Commands

Command	Description	Possible Reply Codes
PWD	Prints the working directory name.	257
QUIT	Closes the control connection. If a data transfer is in progress, the connection will not be closed until it has completed.	221
RETR	Transfers a copy of the file specified in the pathname argument from the server to the client.	150, 226, 425, 550
SIZE	Returns the size of the specified file.	213, 250, 450, 550
STOR	Stores a file from the client onto the server. The file will be overwritten if it already exists at the specified pathname, or it will be created if it does not exist.	150, 226, 250 425, 450, 452, 550
STRU	Confirms the supported structure of a file. Only file-structure is supported: a continuous stream of data bytes.	200, 504
SYST	Sends the string “RABBIT2000.”	215
TYPE	Confirms the transfer type. The types IMAGE (binary), ASCII and Local with 8-bit bytes are all supported and are treated the same.	200, 504

6.7 Reply Codes to FTP Commands

The FTP server replies to all of the commands that it receives. The reply consists of a 3-digit number followed by a space and then a text string explaining the reply. All reply codes sent from the FTP server are listed here.

Reply Code	Reply Text
150	File status okay; about to open data connection.
200	Command okay.
202	Command not implemented, superfluous at this site.
211	System status, or system help reply.
213	File status
214	Help message. On how to use the server or the meaning of a particular non-standard command. This reply is useful only to the human user.
215	System type.
220	Service ready for new user.
221	Service closing connection.
226	Closing data connection. Requested file action successful (for example, file transfer or file abort).
227	Entering Passive Mode (h1,h2,h3,h4,p1,p2).
230	User logged in, proceed
250	Requested file action okay, completed.
257	"PATHNAME" created.
331	User name okay, need password.
425	Can't open data connection.
450	Requested file action not taken. File unavailable (e.g., file busy).
452	Requested action not taken. Insufficient storage space in system.
502	Command not implemented.
504	Command not implemented for that parameter.
530	Not logged in.
550	Requested action not taken. File unavailable (e.g., file not found, no access).

The text used for the reply codes, may be slightly different than what is shown here. It will be context specific.

7. TFTP Client

TFTP.LIB implements the Trivial File Transfer Protocol (TFTP). This standard protocol (internet RFC783) is a lightweight protocol typically used to transfer bootstrap or configuration files from a server to a client host, such as a diskless workstation. TFTP allows data to be sent in either direction between client and server, using UDP as the underlying transport.

This library fully implements TFTP, but as a client only.

Compared with more capable protocols such as FTP, TFTP:

- has no security or authentication
- is not as fast because of the step-by-step protocol
- uses fewer machine resources.

Because of the lack of authentication, most TFTP servers restrict the set of accessible files to a small number of configuration files in a single directory. For uploading files, servers are usually configured to accept only certain file names that are writable by any user. If these restrictions are acceptable, TFTP has the advantage of requiring very little 'footprint' in the client host.

7.0.1 BOOTP/DHCP

In conjunction with DHCP/BOOTP and appropriate server configuration, TFTP is often used to download a kernel image to a diskless host. The target TCP/IP board does not currently support loading the BIOS in this way, since the BIOS and application program are written to non-volatile flash memory. However, the downloaded file does not have to be a binary executable - it can be any reasonably small file, such as an application configuration file. TFTP and DHCP/BOOTP can thus be used to administer the configuration of multiple targets from a central server.

Using TFTP with BOOTP/DHCP requires minimal additional effort for the programmer. Just **#define** the symbol **DHCP_USE_TFTP** to an integer representing the maximum allowable boot file size (1-65535). See the description of the variables **_bootpsize**, **_bootpdata** and **_bootperror** on page 6 for further details.

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.
};
```

7.0.2.1 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 xram (see **xalloc()**).

TFTP Session

A session can be either a single download (get) or upload (put). The functions ending with 'x' are versions that use a data structure allocated in extended memory, for applications that are constrained in their use of root data memory.

tftp_init

```
int tftp_init(struct tftp_state * ts);
```

DESCRIPTION

This function prepares for a TFTP session and is called to complete initialization of the TFTP state structure. Before calling this function, some fields in the structure **tftp_state** must be set up as follows:

```
ts->state    = <0 for read, 1 for write>
ts->buf_addr = <physical address of xmem buffer>
ts->buf_len  = <length of physical buffer, 0-65535>
ts->my_tid   = <UDP port number. Set 0 for default>
ts->sock     = <address of UDP socket (udp_Socket *),or NULL to
                  use DHCP/BOOTP socket>
ts->rem_ip   = <IP address of TFTP server host, or zero to use
                  default BOOTP host>
ts->mode     = <one of the following constants:
                   TFTP_MODE_NETASCII (ASCII text)
                   TFTP_MODE_OCTET (8-bit binary)
                   TFTP_MODE_MAIL (Mail)>
strcpy(ts->file, <remote filename or mail address>)
```

Note that mail mode can only be used to write mail to the TFTP server, and the file name is the e-mail address of the recipient. The e-mail message must be ASCII-encoded and formatted with RFC822 headers. Sending e-mail via TFTP is deprecated. Use SMTP instead since TFTP servers may not implement mail.

PARAMETERS

ts Pointer to **tftp_state**.

RETURN VALUE

0: OK.
-4: Error, default socket in use.

LIBRARY

TFTP.LIB

tftp_initx

```
int tftp_initx(long ts_addr);
```

DESCRIPTION

This function is called to complete initialization of the TFTP state structure, where the structure is possibly stored somewhere other than in the root data space. This is a wrapper function for **tftp_init()**. See that function description for details.

PARAMETERS

ts_addr Physical address of TFTP state (struct **tftp_state**)

RETURN VALUE

0: OK
-1: Error, default socket in use

LIBRARY

TFTP.LIB

tftp_tick

```
int tftp_tick(struct tftp_state * ts);
```

DESCRIPTION

This function is called periodically in order to take the next step in a TFTP process. Appropriate use of this function allows single or multiple transfers to occur without blocking. For multiple concurrent transfers, there must be a unique **tftp_state** structure, and a unique UDP socket, for each transfer in progress. This function calls **sock_tick()**.

PARAMETERS

ts	Pointer to TFTP state. This must have been set up using **tftp_init()**, and must be passed to each call of **tftp_tick()** without alteration.

RETURN VALUE

1: OK, transfer not yet complete.
0: OK, transfer complete
-1: Error from remote side, transfer terminated. In this case, the ts_addr->file field will be overwritten with a **NULL**-terminated error message from the server.
-2: Error, could not contact remote host or lost contact.
-3: Timed out, transfer terminated.
-4: (not used)
-5: Transfer complete, but truncated -- buffer too small to receive the complete file.

LIBRARY

TFTP.LIB

tftp_tickx

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

DESCRIPTION

This function is a wrapper for calling **tftp_tick()**, where the structure is possibly stored somewhere other than in the root data space. See that function description for details.

PARAMETERS

ts_addr Physical address of TFTP state (struct **tftp_state**).

RETURN VALUE

1: OK, transfer not yet complete.
0: OK, transfer complete
-1: Error from remote side, transfer terminated. In this case, the ts_addr->file field will be overwritten with a **NULL**-terminated error message from the server.
-2: Error, could not contact remote host or lost contact.
-3: Timed out, transfer terminated.
-4: (not used)
-5: Transfer complete, but truncated -- buffer too small to receive the complete file.

LIBRARY

TFTP.LIB

tftp_exec

```
int tftp_exec( char put, long buf_addr, word * len, int mode,
   char * host, char * hostfile, udp_Socket * sock );
```

DESCRIPTION

Prepare and execute a complete TFTP session, blocking until complete.This function is a wrapper for **tftp_init()** and **tftp_tick()**. It does not return until the complete file is transferred or an error occurs. Note that approximately 750 bytes of free stack will be required by this function.

PARAMETERS

put	**0**: get file from remote host; **1**: put file to host.
buf_addr	Physical address of data buffer.
len	Length of data buffer. This is both an input and a return parameter. It should be initialized to the buffer length. On return, it will be set to the actual length received (for a get), or unchanged (for a put).
mode	Data representation: 0=NETASCII, 1=OCTET (binary), 2=MAIL.
host	Remote host name, or **NULL** to use default BOOTP host.
hostfile	Name of file on remote host, or e-mail address for mail.
sock	UDP socket to use, or **NULL** to re-use BOOTP socket if available.

RETURN VALUE

0: OK, transfer complete.
-1: Error from remote side, transfer terminated. In this case, **ts_addr->file** will be overwritten with a **NULL**-terminated error message from the server.
-2: Error, could not contact remote host or lost contact.
-3: Timed out, transfer terminated
-4: sock parameter was **NULL**, but BOOTP socket was unavailable.

LIBRARY

TFTP.LIB

8. SMTP Mail Client

SMTP (Simple Mail Transfer Protocol) is one of the most common ways of sending e-mail. SMTP is a simple text conversation across a TCP/IP connection. The SMTP server usually resides on TCP port 25 waiting for clients to connect.

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_mailtick()** 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 administer 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_mailtick** no longer returns **SMTP_PENDING**.

PARAMETERS

to	String containing the e-mail address of the destination.
from	String containing the e-mail address of the source.
subject	String containing the subject of the message.
message	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, long 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_mailtick** no longer returns **SMTP_PENDING**.

PARAMETERS

to	String containing the e-mail address of the destination.
from	String containing the e-mail address of the source.
subject	String containing the subject of the message.
message	Physical address in xmem containing the message. (The message must NOT contain the byte sequence "\r\n.\r\n" (CRLF.CRLF), as this is used to mark the end of the e-mail, and will be appended to the e-mail automatically.)
messagelen	Length of the message in xmem.

RETURN VALUE

None

LIBRARY

SMTP.LIB

smtp_mailtick

```
int smtp_mailtick(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 you 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_mailtick** 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_mailtick 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. For an example of using **smtp_sendmailxmem()**, see the sample program **samples\tcpip\smtp\smtpxmem.c**.

Program Name: /samples/tcpip/smtp/smtp.c

```
/* 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_mailtick()==SMTP_PENDING)
      continue;

   if(smtp_status()==SMTP_SUCCESS)
      printf("Message sent\n");
   else
      printf("Error sending message\n");
}
```

9. POP3 Client

Post Office Protocol version 3 (POP3) is probably the most common way of retrieving e-mail from a remote server. Most e-mail programs, such as Eudora, MS-Outlook, and Netscape's e-mail client, use POP3. The protocol is a fairly simple text-based chat across a TCP socket, normally using TCP port 110.

There are two ways of using **POP3.LIB**. The first method provides a raw dump of the incoming e-mail. This includes all of the header information that is sent with the e-mail, which, while sometimes useful, may be more information than is needed. The second method provides a parsed version of the e-mail, with the sender, recipient, subject-line, and body-text separated out.

In both methods, each line of e-mail has CRLF stripped from it and '\0' appended to it.

9.1 Configuration

The POP3 client can be configured through the following macros:

POP_BUFFER_SIZE

This will set the buffer size for **POP_PARSE_EXTRA** in bytes. These are the buffers that hold the sender, recipient and subject of the e-mail. **POP_BUFFER_SIZE** defaults to 64 bytes.

POP_DEBUG

This will turn on debug information. It will show the actual conversation between the device and the remote mail server, as well as other useful information.

POP_NODELETE

This will stop the POP3 library from removing messages from the remote server as they are read. By default, the messages are deleted to save storage space on the remote mail server.

POP_PARSE_EXTRA

This will enable the second mode, creating a parsed version of the e-mail as mentioned above. The POP3 library parses the incoming mail more fully to provide the Sender, Recipient, Subject, and Body fields as separate items to the call-back function.

9.2 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

username	The **username** of the account to access.
password	The **password** of the account to access.
server	The IP address of the server to connect to, as returned from **resolve()**.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

POP3.LIB

pop3_tick

```
int pop3_tick(void)
```

DESCRIPTION

A standard tick function, to run the daemon. Continue to call it as long as it returns **POP_PENDING**.

RETURN VALUE

POP_PENDING: Transfer is not done; call **pop3_tick** again.
POP_SUCCESS: All e-mails were received successfully.
POP_ERROR: Unknown error occurred.
POP_TIME: Session timed-out. Try again, or use **POP_TIMEOUT** to increase the time-out length.

LIBRARY

POP3.LIB

9.5 Sample Receiving of E-mail

This program connects to a POP3 server and downloads e-mail form it.

Program Name: /samples/tcpip/pop3/parse_extra.c

```
#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.

10. Telnet

The library, **Vserial.lib**, implements the telecommunications network interface, known as telnet. The implementation is a telnet-to-serial and serial-to-telnet gateway. This chapter is divided into two parts. The first part describes the library from Dynamic C version 7.05 and later. The second part describes the library prior to 7.05.

10.1 Telnet (Dynamic C 7.05 and later)

This implementation is more general than the previous one. Any of the four serial ports can be used and other I/O streams can be added. Multiple connections are supported by the use of unique gateway identifiers.

10.1.1 Setup

To use a serial port, the circular buffers must be initialized. For instance, if serial port A is used by an application, then the following macros must be defined in the program:

```
#define AINBUFSIZE   31
#define AOUTBUFSIZE  31
```

It might be necessary to have bigger buffers for some applications.

10.1.1.1 Low-level Serial Routines

A table to hold the low-level I/O routines must be defined as type **VSerialSpec**.

```
typedef struct {
  int id;          // unique ID to match w/ calls to listen/open
  int (*open)();   // serial port routines, or
  int (*close)();  //  serial port compatible routines.
  int (*tick)();
  int (*rdUsed)();
  int (*wrFree)();
  int (*read)();
  int (*write)();
} VSerialSpec;
```

For each serial port (A, B, C and D), there is a pre-defined macro in **VSERIAL.LIB**:

```
#define VSERIAL_PORTA(id) { (id), serAopen, serAclose, NULL,
   serArdUsed, serAwrFree, serAread, serAwrite }
```

The parameter being passed to **VSERIAL_PORTA** is the unique gateway identifier mentioned earlier. This value is chosen by the developer when entries are made to the array of type **VSerialSpec** (also known as the spec table).

10.1.1.2 Configuration Macros

VSERIAL_DEBUG

Turns on debug messages.

VSERIAL_NUM_GATEWAYS

The number of telnet sessions must be defined and must match the number of entries in the spec table.

10.1.2 Function Reference (Dynamic C 7.05 and later)

vserial_close

```
int vserial_close(int id);
```

DESCRIPTION

Closes the specified gateway. This will not only terminate any network activity, but will also close the serial port.

PARAMETERS

id	ID of the gateway to change, as specified in the spec table.

RETURN VALUE

0: Success;
1: Failure.

LIBRARY

VSERIAL.LIB

vserial_init

```
int vserial_init ( void );
```

DESCRIPTION

Initializes the daemon and parses the spec table.

RETURN VALUE

0: Success;
1: Failure.

LIBRARY

VSERIAL.LIB

vserial_keepalive

```
int vserial_keepalive ( int id, long timeout );
```

DESCRIPTION

This function sets the keepalive timer to generate TCP keepalives after **timeout** periods of inactivity. This helps detect if the connection has gone bad.

Keepalives should be used at the application level, but if that is not possible, then **timeout** should be set so as to not overload the network. The standard timeout is two hours, and should be set sooner than that only for a Very Good Reason.

PARAMETERS

id	Unique gateway identifier.
timeout	Number of seconds of inactivity allowed before a TCP keepalive is sent. A value of 0 shuts off keepalives.

RETURN VALUE

0: Success;
1: Failure.

LIBRARY

VSERIAL.LIB

vserial_listen

```
int vserial_listen(int id, long baud, int port, long
   remote_host, int flags);
```

DESCRIPTION

Listens on the specified port for a telnet connection. The gateway process is started when a connection request is received. On disconnect, re-listen happens automatically.

PARAMETERS

id	ID of the gateway to change, as specified in the spec table.
baud	The parameter to send to the **open()** serial port command; it's usually the baud rate.
port	The local TCP port to listen on.
remote_host	The remote host from whom to accept connections, or 0 to accept a connection from anybody.
flags	Option flags for this gateway. Currently the only valid bit flags are **VSERIAL_COOKED** to strip out telnet control codes, or 0 to leave it a raw data link.

RETURN VALUE

0: Success;
1: Failure.

LIBRARY

VSERIAL.LIB

vserial_open

```
int vserial_open(int id, long baud, int port, long remote_host,
   int flags, long retry);
```

DESCRIPTION

Opens a connection to a remote host and maintains it, starting the gateway process.

PARAMETERS

id	ID of the gateway to change, as specified in the spec table.
baud	The parameter to send to the **open()** serial port command; it's usually the baud rate.
port	The TCP port on the remote host to connect to.
remote_host	The remote host to connect to.
flags	Option flags for this gateway. Currently the only valid bit flags are **VSERIAL_COOKED** to strip out telnet control codes, or 0 to leave it a raw data link.
retry	The retry timeout, in seconds. When a connection fails, or if the connection was refused, we will wait this number of seconds before retrying.

RETURN VALUE

0: Success;
1: Failure.

LIBRARY

VSERIAL.LIB

vserial_tick

```
int vserial_tick(void);
```

DESCRIPTION

Runs the telnet daemon - must be called periodically.

RETURN VALUE

0: Success;
1: Failure.

But call it periodically no matter the return value! An error message can be seen when 1 is returned if you **#define VSERIAL_DEBUG** at the top of your program.

LIBRARY

VSERIAL.LIB

10.1.3 Sample Program (Dynamic C 7.05 and later)

```
/*****************************************************************
 * vserial.c
 * This demonstrates the use of the new VSERIAL.LIB, which provides
 * a gateway between serial ports or serial-port-like devices, and
 * a telnet-style TCP socket.
 *****************************************************************/

#define MY_IP_ADDRESS  "10.10.6.105"
#define MY_NETMASK  "255.255.255.0"
#define MY_GATEWAY  "10.10.6.1"

/*
 * Each gateway mapping must be uniquely identified with a number.
 * Macros are used for code readability.
 */
#define GATEWAY_PORTC 1

/*
 * Serial buffer sizes have to be defined any time the serial ports
 * are used, because of how RS232.LIB works.
 */

#define CINBUFSIZE  31
#define COUTBUFSIZE 31
```

```
/* Uncomment this to see debug messages */
//#define VSERIAL_DEBUG

/*
 * The number of gateways that will be specified. This must match the
 * number of rows in the VSerialSpecTable that is defined below.
 */
#define VSERIAL_NUM_GATEWAYS 1

#use "vserial.lib"
/*
 * This table defines the low-level serial routines used to talk to
 * the serial port hardware. Each row is one possible hardware
 * gateway. Because the built-in Rabbit serial ports will be used
 * often, shortcut-macros are defined for each of the ports, A-D.
 * They take as a parameter an identifier such that they can be
 * referenced by the vserial_* functions below.
 */
const VSerialSpec VSerialSpecTable[] = {
      VSERIAL_PORTC(GATEWAY_PORTC),
};

main()
{
      sock_init();

      /* Initilize the vserial library (parse the above structures)*/
      if(vserial_init()) {
         printf("Error starting vserial library!\n");
         exit(-1);
      }

      /* Enable our first serial->tcp mapping  */
      if(vserial_listen(GATEWAY_PORTC,57600,23,0L,VSERIAL_COOKED)) {
         printf("Error listening!\n");
         exit(-1);
      }

      /*
       * Force the tcp connection to be persistent. This causes
       * TCP Keepalives to be sent on the socket periodically. It is
       * important to note that this can cause a large amount of
       * network traffic over time.
       */
      if(vserial_keepalive(GATEWAY_PORTC,30)) {
         printf("Error setting keepalive!\n");
         exit(-1);
      }

      /* run it */
      for(;;) {
         vserial_tick();
      }
}
```

10.2 Telnet (pre-Dynamic C 7.05)

10.2.1 Configuration Macros

SERIAL_PORT_SPEED

The baud rate of the serial port. Defaults to 115,200 bps.

TELNET_COOKED

#define this to have telnet control codes stripped out of the data stream (useful if you are actually Telneting to the device from another box; should probably NOT be defined if you are using two devices as a transparent bridge over the Ethernet).

10.2.2 Function Reference

telnet_init

```
int telnet_init(int which, longword addy, int port);
```

DESCRIPTION

Initializes the connection.

PARAMETERS

which	Is one of the following: **TELNET_LISTEN**—Listens on a port for incoming connections. **TELNET_RECONNECT**—Connects to a remote host, and reconnects if the connection dies. **TELNET_CONNECT**—Connects to a remote host, and terminates if the connection dies.
addy	IP address of the remote host, or 0 if we are listening.
port	Port to bind to if we are listening, or the port of the remote host to connect to.

RETURN VALUE

0: Success;
1: Failure.

LIBRARY

VSERIAL.LIB

telnet_tick

```
int telnet_tick(void);
```

DESCRIPTION

Must be called periodically to run the daemon.

RETURN VALUE

0: Success (call it again);
1: Failure; **TELNET_CONNECT** died, or a fatal error occurred.

LIBRARY

VSERIAL.LIB

telnet_close

```
void telnet_close(void);
```

DESCRIPTION

Terminates any connections currently open, and shuts down the daemon.

LIBRARY

VSERIAL.LIB

10.2.3 An Example Telnet Server

```
/*
 * Telnet Server: Listens on a telnet port for a connection, and
 * transparently passes data on to the serial port
 */

// Initilize 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

#memmap xmem
#use "dcrtcp.lib"
#use "vserial.lib"

/*
 * 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();
}
```

10.2.3.1 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
#memmap 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();
}
```

11. General Purpose Console

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:

- Login name and password protection.
- Default and custom Console commands.
- Default and custom error messages.
- Help text for Console commands, including custom commands
- Multiple I/O streams that can be used simultaneously.
- A fail-safe backup system for configuration data.

11.2.1 Using other Dynamic C Libraries

An application program that uses the Console must include the lines

```
#use "filesystem.lib" // if using the improved file system
                      // that is available starting with
                      // Dynamic C 7.05, substitute "fs2.lib"
                      // for "filesystem.lib"
#use "zconsole.lib"
```

Dynamic C TCP/IP functionality may be used by a Console application program by including the appropriate libraries.

11.3 Login Name and Password

There is a sample program, **/samples/tcpip/LOGINCONSOLE.C**, that demonstrates the use of the login name/password functionality for the Console. The console command functions: **con_loginname()**, **con_loginpassword()** and **con_logout()** are described in Section 11.4.3.1 starting on page 272. The structure that saves the name and password information can be backed up using the backup macro **CONSOLE_BACKUP_LOGIN**. Please see Section 11.7 starting on page 290 for details on the backup system.

11.4 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.

11.4.1 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 11.4.3.1 on page 272.

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 **#ximport**ed text files.

11.4.1.1 Help Text for General Cases

There are two cases in **Zconsole.lib** where help text is needed, but is not associated with a particular command. It is still necessary to allocate a **ConsoleCommand** structure to access the help text. The first case is the help overview given when **HELP** is entered by itself. The **command** field should be “” and the **cmdfunc** field should be **NULL**.

```
{ "", NULL, help_txt },
```

The second case is **HELP SET**. This is an overview of the family of SET commands, i.e. commands that set configuration values. For **HELP SET**, the **command** field should be “SET” and the **cmdfunc** field should be **NULL**.

```
{ "SET", NULL, help_set_txt },
```

This second case illustrates the general case of displaying help for a family of commands. The family name can not be the name of a command.

11.4.2 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.

11.4.3 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 }
};
```

11.4.3.1 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 11.4.3.2 on page 277. 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. A command that uses this function takes one parameter: the name of the file to delete.

con_echo()

This function turns on or off the echoing of characters on a particular I/O stream. That is, it does not affect echoing globally, but only for the I/O stream on which it is issued. A command that uses this function takes one parameter: **ON | OFF**.

con_get()

This function displays a file from the file system. It works in the following fashion (if the command is called “GET”):

- ASCII mode: usage: "get <filename>"

  The file is then sent, followed by the usual OK message.

- BINARY mode: usage: "get <filename> <size in bytes>"

  The message "LENGTH <len>" will be sent, indicating length of the file to be sent, and then the file will be sent, but not more than <size in bytes> bytes.

con_getv()

This function displays the value of the given variable. The variable is displayed using the printf-style format specifier given in the **`createv`** command. A command that uses this function takes one parameter: the name of the variable.

con_help()

This function implements the help system for the Console. A 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 file system and their file sizes. A command that uses this function takes no parameters.

con_list_variables()

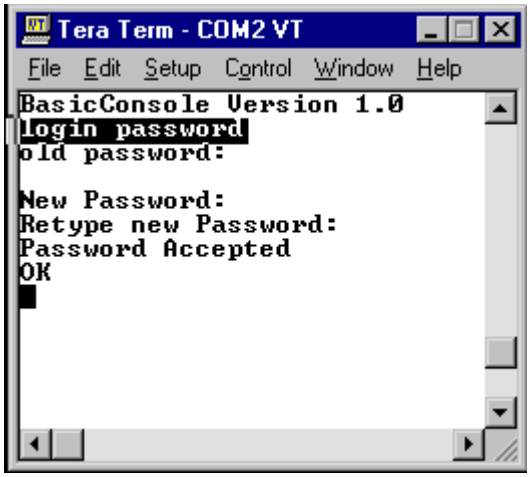
This function displays the names and types of all variables. A command that uses this function takes no parameters.

con_loginname()

This function stores an identifier that will be remembered across power cycles (with battery-backed RAM). The existence of the identifier will be used to prompt the user of a new Console session. Before Console access to the controller is allowed, a valid identifier must be entered in response to the prompt. A command that uses this function takes one parameter: an identifier that will be used as the login name.

con_loginpassword()

This function stores an identifier that will be remembered across power cycles (with battery-backed RAM). The existence of the identifier will be used to prompt the user for a password after a login name has been entered. Before Console access to the controller is allowed, a valid identifier must be entered in response to the prompt. A command that uses this function takes no parameters on the command line, but requires a series of user inputs in response to prompts. In the following screen shot, the command is named “login password,” and is typed in by the user. All other screen text shown here was printed by the Console.



If no identifier is stored for the password, a **<CR>** must be sent in response to the prompt for the old password.

> **NOTE:** A login name must be stored by a command using **`con_loginname()`** for a login password to be applicable, i.e. if a password has been stored but no login name, new Console sessions will not prompt for the password or a login name. If a login name is applicable, but there is no password, new Console sessions will prompt for the login name and grant access after a valid name is entered without prompting for a password.

con_logout()

This function exits the current Console session and begins a new session by entering the initialization state of the Console. A command that uses this function takes no parameters.

con_mail()

This function sends e-mail to the server specified by **`con_mail_server()`**, with the return address specified by **`set_mail_from()`**. A command that uses this function takes one parameter: the destination e-mail address. If the command is named mail, the usage is:

```
"mail destination@where.com"
```

The first line of the message will be used as the subject, and the other lines are the body. The body is terminated with a ^D or ^Z (0x04 or 0x1A).

con_put()

This function creates a new file in the file system for use with the HTTP server. It works in the following fashion (if the command is called “PUT”):

- ASCII mode: usage: "put <filename>"
  The file is then sent, terminating with a ^D or ^Z (0x04 or 0x1A).
- BINARY mode: usage: "put <filename> <size in bytes>"
  The file is then sent, and must be exactly the specified number of bytes in length.

Note that ASCII mode is only useful for text files, since the Console will ignore non-displayable characters. In binary mode, the put command will time out after **`CON_TIMEOUT`** seconds of inactivity (60 by default).

con_putv()

This function updates the value of a variable. A command that uses this function takes two parameters: the name of the variable, and the new value for the variable.

con_reset_files

This function removes all web files.

con_reset_variables

This function removes all web variables.

con_set_gateway()

This function changes the gateway of the board. A command that uses this function takes one parameter: the new gateway in dotted quad notation, e.g., 192.168.1.1.

con_set_ip()

This function changes the IP address of the board. A command that uses this function takes one parameter: the new IP address in dotted quad notation, e.g., 192.168.1.112.

con_set_param

This function sets the parameter for the current I/O device. Depending on the I/O device, this value could be a baud rate, a port number or a channel number. A command that uses this function takes one parameter: the value for the I/O device parameter.

con_set_mail_from

This function sets the return address for all e-mail messages. This address will be added to the outgoing e-mail and should be valid in case the e-mail needs to be returned. A command that uses this function takes one parameter: the return address.

con_set_mail_server

This functions identifies the SMTP server to use. A command that uses this function takes one parameter: the IP address of the SMTP server.

con_set_nameserver()

This function changes the name server for the board. A command that uses this function takes one parameter: the IP address of the new name server in dotted quad notation, e.g., 192.168.1.1.

con_set_netmask()

This function changes the netmask of the board. A command that uses this function takes one parameter: the new netmask in dotted quad notation, e.g., 255.255.255.0.

con_show()

This function displays the current configuration of the board (IP address, netmask, and gateway). If the developer's application has configuration options she would like to show other than the IP address, netmask, and gateway, she will probably want to implement her own version of the show command. The new show command can be modelled after **`con_show()`** in **`ZConsole.lib`**. A command that uses this function takes no parameters.

11.4.3.2 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 11.4.1. 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 time out.
} 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 **#ximport**ed, along with all of the default command function help files that are being used.

11.4.4 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
}
```

11.4.4.1 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_SMTPERROR      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_SMTPERROR,       "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."
}
```

11.4.4.2 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.

11.5 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) ();
   void (*close)();
   int (*tick) ();
   int (*puts) ();
   int (*rdUsed) ();
   int (*wrUsed) ();
   int (*wrFree) ();
   int (*read) ();
   int (*write) ();
} ConsoleIO;
```

11.5.1 How to Include an I/O Method

Each supported I/O method is determined at compile time, i.e., each supported I/O method must have an entry in **console_io[]**.

11.5.2 Predefined I/O Methods

Several predefined I/O methods are in **Zconsole.lib**. They will be included by entering their respective macros in **console_io[]**.

```
const ConsoleIO console_io[] = {
   CONSOLE_IO_SERA(baud rate),
   CONSOLE_IO_SERB(baud rate),
   CONSOLE_IO_SERC(baud rate),
   CONSOLE_IO_SERD(baud rate),
   CONSOLE_IO_SP(channel number),
   CONSOLE_IO_TELNET(port number),
}
```

The macros expand to the appropriate set of pre-defined handler functions, e.g.,

```
#define CONSOLE_IO_SERA(param) { param, serAopen, serAclose,
  NULL, conio_serAputs, serArdUsed, serAwrUsed, serAwrFree,
  serAread, serAwrite}
```

11.5.2.1 Serial Ports

There are predefined I/O methods for all four of the serial ports on a Rabbit board. The baud rate is set by passing it to the macro. See above.

11.5.2.2 Telnet

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.

11.5.2.3 Slave Port

The Rabbit slave port is an 8-bit bidirectional data port. The Console runs on the slave processor. Two drivers are needed.

11.5.2.3.1 Slave Port Driver

The slave port driver is implemented by **SLAVE_PORT.LIB**. For an application to use the slave port:

- The driver must be installed by including the library in the program.
- A call to **SPinit(mode)** must be made to initialize the driver.
- A function to process 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 provided in **SP_STREAM.LIB**. The handler is set up automatically by the Console when the slave port I/O method is included. The macro, **CONSOLE_IO_SP**, expands to the I/O functions defined in **SP_STREAM.LIB**.

11.5.2.3.2 Master Connected to Rabbit Slave Port

The master controller board can be another Rabbit processor or something else.

The master also needs a driver for its end of the slave port connection. An example of the software needed on the master is given in **MASTER_SERIAL.LIB**. The software on the master controller is, of course, specific to the task at hand. In the example driver provided, most of the work is done by the slave, making minimal changes necessary to the code on the master.

11.5.2.4 Custom I/O Methods

To define a custom I/O method, you must add a structure of type **ConsoleIO** to **console_io[]**. This structure holds the common handler functions for the I/O method. The tick function may have a **NULL** pointer, but the rest of the function pointers must be valid pointers to functions.

11.5.3 Multiple I/O Streams

Each I/O method has its own state machine in the Console. That means that each I/O method is independent of the others and they can all be used simultaneously. This imposes the important restriction that all command handlers be able to run simultaneously on different I/O streams or support proper locking for functions that cannot be performed simultaneously.

11.6 Console Execution

Normally, the Console will communicate over a serial link. The physical connection will differ slightly from board to board. Basically, you will need a 3 wire (GND, RXD, TXD) serial cable. In order to execute the Console several initialization steps must be taken at the beginning of an application program.

11.6.1 File System Initialization

The Console depends on the file system that is included with Dynamic C. There are actually two file systems: FS1 was the first Dynamic C file system. The second one, FS2 (introduced with Dynamic C 7.05), is an improved file system.

Besides defining the macro that directs the file system to EEPROM memory and including the appropriate library, i.e.,

```
#define FS_FLASH
#use "filesystem.lib" // If using the improved file system
                      // that is available starting with
                      // Dynamic C 7.05, substitute “fs2.lib”
                      // for “filesystem.lib”
```

the application program must initialize the file system with a call to **fs_init()**.

11.6.2 Serial Buffers

If the pre-defined serial I/O methods are used, the circular buffers used for I/O data can be resized from their default values of 31 bytes by using macros. For example, if **CONSOLE_IO_SERIALC** is included in **console_io[]**, then lines similar to the following can be in the application program:

```
#define CINBUFSIZE 1023
#define COUTBUFSIZE 255
```

In general, these buffers can be smaller for slower baud rates, but must be larger for faster baud rates.

11.6.3 Using TCP/IP

To use the TCP/IP functionality of 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**.

11.6.4 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 data structures. It must be called before **console_tick()** is called for the first time. This function also loads the configuration information from the file system.

RETURN VALUE

0: Success;
1: No configuration information found.
<0: Indicates an error loading the configuration data;
-1 indicates an error reading the 1st set of information,
-2 the 2nd set, and so on.

LIBRARY

zconsole.lib

console_tick

```
void console_tick(void);
```

DESCRIPTION

This function needs to be called periodically in an application program to allow the Console time for processing.

LIBRARY

zconsole.lib

11.6.5 Useful Console Function

Most of the following functions are only useful for creating custom commands.

con_backup

```
int con_backup(void);
```

DESCRIPTION

This function backs up the current configuration.

RETURN VALUE

0: Success
1: Failure

LIBRARY

zconsole.lib

SEE ALSO

con_backup_reserve, con_load_backup

con_backup_bytes

```
long con_backup_bytes(void);
```

DESCRIPTION

Returns the number of bytes necessary for each backup configuration file. Note that enough space for 2 of these files needs to be reserved. This function is most useful when **ZCONSOLE.LIB** is being used with **FS2.LIB.**

RETURN VALUE

Number of bytes needed for a backup configuration file.

LIBRARY

zconsole.lib

SEE ALSO

con_backup_reserve

con_backup_reserve

```
void con_backup_reserve(void);
```

DESCRIPTION

Reserves space for the configuration information in the file system. For more information on the file system see the *Dynamic C User's Manual*.

LIBRARY

zconsole.lib

SEE ALSO

con_backup, con_load_backup, con_backup_bytes

con_chk_timeout

```
int con_chk_timeout(unsigned long timeout);
```

DESCRIPTION

Checks whether the given timeout value has passed.

RETURN VALUE

0: Timeout has not passed
!0: Timeout has passed

LIBRARY

zconsole.lib

SEE ALSO

con_set_timeout

con_load_backup

```
int con_load_backup(void);
```

DESCRIPTION

Loads the configuration from the file system.

RETURN VALUE

0: Success
1: No configuration information found
<0: Failure
-1: error reading 1st set of information
-2: error reading 2nd set of information, and so on

LIBRARY

zconsole.lib

SEE ALSO

con_backup, con_backup_reserve

con_reset_io

```
void con_reset_io(void);
```

DESCRIPTION

Resets all I/O methods by calling **close()** and **open()** on each of them.

LIBRARY

zconsole.lib

con_set_backup_lx

```
void con_set_backup_lx(FSLXnum backuplx);
```

DESCRIPTION

Sets the logical extent (LX) that will be used to store the backup configuration data. For more information on the file system see the *Dynamic C User's Manual*. This is only useful in conjunction with **FS2.LIB**. This should be called once before **console_init()**. Care should be taken that enough space is available in this logical extent for the configuration files. See **con_backup_bytes()** for more information.

PARAMETER

backuplx LX number to use for backup

LIBRARY

zconsole.lib

SEE ALSO

con_set_files_lx, con_backup_bytes

con_set_files_lx

```
void con_set_files_lx(FSLXnum fileslx);
```

DESCRIPTION

Sets the logical extent (LX) that will be used to store files. For more information on the file system see the *Dynamic C User's Manual*. This is only useful in conjunction with **FS2.LIB**. This should be called once before **console_init()**.

PARAMETER

fileslx LX number to use for files.

LIBRARY

zconsole.lib

SEE ALSO

con_set_backup_lx

con_set_user_idle

```
void con_set_user_idle(void (*funcptr)());
```

DESCRIPTION

Sets a user-defined function that will be called when the console (for a particular I/O channel) is idle. The user-defined function should take an argument of type **ConsoleState*** .

LIBRARY

zconsole.lib

SEE ALSO

con_set_user_timeout

con_set_timeout

```
unsigned long con_set_timeout(unsigned int seconds);
```

DESCRIPTION

Returns the value that **MS_TIMER** should have when the number of seconds given have elapsed.

LIBRARY

zconsole.lib

SEE ALSO

con_chk_timeout

con_set_user_timeout

```
void con_set_user_timeout(void (*funcptr)());
```

DESCRIPTION

Sets a user-defined function that will be called when a timeout event has occurred. The user-defined function should take an argument of type **ConsoleState***.

LIBRARY

zconsole.lib

SEE ALSO

con_set_user_idle

console_disable

```
void console_disable(int which);
```

DESCRIPTION

Disable processing for the designated console in the **console_io[]** array. This function, along with **console_enable()**, allows the sharing of the console port with some other processing.

PARAMETER

which The console to disable.

LIBRARY

zconsole.lib

SEE ALSO

console_init, console_enable

console_enable

```
void console_enable(int which);
```

DESCRIPTION

Enable processing for the designated console in the **console_io[]** array. This function, along with **console_disable()**, allows the sharing of the console port with some other processing.

PARAMETER

which The console to enable.

LIBRARY

zconsole.lib

SEE ALSO

console_init, console_disable

11.6.6 Console Execution Choices

The Console can be used interactively with a terminal emulator or by sending commands from a program running on a device connected to the controller that is running the Console.

11.6.6.1 Terminal Emulator

To manually enter Console commands from a keyboard and view results in the Stdio window you must:

1. Run Dynamic C, version 7.05 or later.
2. Open a terminal emulator. Windows HyperTerminal comes with Windows. It does not work with binary files, only ASCII. Tera Term, available for free download at

   `http://hp.vector.co.jp/authors/VA002416/teraterm.html`

   can handle both ASCII and binary.
3. Configure the terminal emulator as follows:

   COM port: (1 or 2) to which 3-wire serial cable is connected
   Baud Rate: 57,600 bps
   Data Bits: 8
   Parity: None
   Stop Bits: 1
   Flow Control: None

The terminal emulator should now accept 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.

11.7 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.

11.7.1 Data Structure for Backup System

The developer must define an array called **`console_backup[]`** of **`ConsoleBackup`** structures.

```
typedef struct {
   void* data;
   int len;
   void (*postload)();
   void (*presave)();
} ConsoleBackup;
```

data
This is a pointer to the data to be backed up.

len
This is how many bytes of data need to be backed up.

postload
This is a function pointer to a function that is called after configuration data is loaded, in case the developer needs to do something with the newly loaded configuration data.

presave
This is a function pointer that is called just before the configuration data is saved so that the developer can fill in the data structure to be saved. The functions referenced by **`postload()`** and **`presave()`** should have the following prototype:

```
void my_preload(void* dataptr);
```

The **`dataptr`** parameter is the address of the configuration data (the same as the data pointer in the **`ConsoleBackup`** structure).

11.7.2 Array Definition for Backup System

```
const ConsoleBackup console_backup[] = {
   CONSOLE_BASIC_BACKUP,       // echo state, baud rate/port number
   CONSOLE_TCPIP_BACKUP,
   CONSOLE_HTTP_BACKUP,
   CONSOLE_SMTP_BACKUP
   CONSOLE_BACKUP_LOGIN
   { my_data, my_data_len, my_preload, my_presave }
}
```

`CONSOLE_BASIC_BACKUP` causes backup of the echo state (on or off), baud rate and port number information.

`CONSOLE_TCPIP_BACKUP` causes backup of the IP addresses of the controller board and the IP address of its 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.

`CONSOLE_BACKUP_LOGIN` causes backup of the **`ConsoleLogin`** structure which stores the login name and password strings.

11.8 Console Macros

Zconsole.lib offers many macros that change the behavior of the Console.

CON_BACKUP_FILE1

The file number used for the first backup file. For FS1, this number must be in the range 128-143, so that **fs_reserve_blocks()** can be used to guarantee free space for the backup files. Defaults to 128 for FS1. Defaults to 254 for FS2.

CON_BACKUP_FILE2

Same as above, except this is for the second backup file. Two files are used so that configuration information is preserved even if the power cycles while configuration data is being saved. For FS1, this number must be in the range 128-143. Defaults to 129 for FS1. Defaults to 255 for FS2.

CON_BUF_SIZE

Changes the size of the data buffer that is allocated for each I/O method. If the baud rate or transfer speed is too great for the Console to keep up, then increasing this value may help avoid dropped characters. It is allocated in root data space. It defaults to 1024 bytes.

CON_CMD_SIZE

Changes the size of the command buffer that is allocated for each I/O method. This limits the length of a command line. It is allocated in root data space. Defaults to 128 bytes.

CON_CMD_DATA_SIZE

Default is 16. Adjusts the size of the user data area within the state structure so that user commands may preserve arbitrary information across calls. The user data area is allocated in root data space.

CON_HELP_VERSION

This macro should be defined if the developer wants a version message to be displayed when the HELP command is issued with no parameters. If this macro is defined, then the macro **CON_VERSION_MESSAGE** must also be defined.

CON_INIT_MESSAGE

Defines the message that is displayed on all Console I/O methods upon startup. Defaults to “Console Ready\r\n”.

CON_MAIL_BUF_SIZE

Maximum length of a mail message. Defaults to 1024.

CON_MAIL_FROM_SIZE

Maximum length of mail from address to **NULL** terminator. Default to 51.

CON_MAIL_SERV_SIZE

Maximum length of mail server name and **NULL** terminator. Defaults to 51.

CON_MAX_NAME

Default is 10: maximum number of characters for a login name. This value must be equal to or less than **CON_CMD_DATA_SIZE**.

CON_MAX_PASSWORD

Default is 10: maximum number of characters for a login password.

CON_SP_RDBUF_SIZE
Size of the slave port read buffer. Defaults to 255.

CON_SP_WRBUF_SIZE
Size of the slave port write buffer. Defaults to 255.

CON_TIMEOUT
Adjusts the number of seconds that the Console will wait before cancelling the current command. The timeout can be adjusted in user code in the following manner:

```
state->timeout = con_set_timeout(CON_TIMEOUT);
```

This is useful for custom user commands so that they can indicate when something "meaningful" has happened on the Console (such as some data being successfully transferred).

CON_VAR_BUF_SIZE
Adjusts the size of the variable buffer, in which values of variables can be stored for use with the HTTP server. It is allocated in xmem space. Defaults to 1024 bytes.

CON_VERSION_MESSAGE
This defines the version message to display when the HELP command is issued with no parameters. It is not defined by default, so has no default value.

11.9 Sample Program

```
/*********************************************************************
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 file system 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
```

```
#memmap 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 file system. 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. Custom commands always take a
 * ConsoleState* as an argument (a pointer to the state structure
 * for the given I/O stream), and return an int. The return value
 * should be 0 when the command wishes to be called again on the
 * next console_tick(), 1 when the command has successfully
 * finished processing, or -1 when the command has finished due
 * to an error.
 */
int hello_world(ConsoleState* state)
{
      state->conio->puts("Hello, World!\r\n");
      return 1;
}
```

```
void main(void)
{
      /*
       * 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();
      }
}
```

12. PPP Driver

The PPP packet driver is a set of libraries in Dynamic C that allows the user to establish a PPP (Point-to-Point Protocol) link over a full-duplex serial line between a Rabbit-based controller and another system that supports PPP.

A common use of the PPP protocol is the transfer of IP packets between a remote host and an Internet Service Provider (ISP) over a modem connection. The PPP packet driver supports the transfer of Internet Protocol (IP) data and is compatible with all TCP/IP libraries for the Rabbit.

The PPP packet driver was derived from source code originally written by Darby Corporate Solutions (DCS).

12.1 PPP Libraries

The PPP driver is in two library files.

PPPLINK.LIB contains:

- The interrupt service routine for transmitting and receiving characters over the serial link. It also handles the insertion and detection of escape characters and CRC generation and checking.
- **PPPflowcontrolOn()**, **PPPflowcontrolOff()**, and **PPPclose()**

PPP.LIB contains:

- Routines for setting up and running the PPP connection.

A third library, **MODEM.LIB**, contains functions for controlling an external modem through a full RS232 link.

12.2 Operation Details

The first step is to configure whatever transport medium will be used for the PPP connection. For directly connecting a serial line to the peer, the two serial data lines TX and RX may be adequate. The most common situation, however, will be some sort of modem.

12.2.1 The Modem Interface

The interface between a modem and a controller is either a true RS232 interface or a variation on RS232 that uses TTL voltage levels for all of the signals. The latter are used by board-mounted modem modules. If an external modem is used, an RS232 transceiver chip is needed to convert RS232 voltages to logic signals and vice versa. A full RS232 connection has 3 outputs and 5 inputs from the controllers point of view. In RS232 terminology, the controller is referred to as the DTE (Data Terminal Equipment). Modems and other peripherals are referred to as DCE's (Data Communications Equipment).

The specifics of a dial-up PPP connection are dependant on the modem hardware and the ISP.

12.2.1.1 Rabbit Pin Connections to Modem

The modem control library, **MODEM.LIB**, defines default connections to the Rabbit as follows:

RS232 Signal	Rabbit Pin	Direction
DTR	PB6	out
RTS	PB7	out
CTS	PB0	in
DCD	PB2	in
RI	PB3	in
DSR	PB4	in
TD	PC2	out
RD	PC3	in

12.2.2 Flow Control

Hardware flow control is implemented for the Rabbit PPP system. It follows the RS232 convention of using Ready To Send (RTS) and Clear To Send (CTS) lines. Flow control is usually required for baud rates above 9600. Flow control can be enabled or disabled by **PPPflowcontrolOn()** and **PPPflowcontrolOff()**, respectively. Flow control is off by default.

12.2.3 Serial Port C

By default the PPP link is established using serial port C. It can be changed, but it requires some **#define** changes near the top of **PPPLINK.LIB** in the section starting with **PPP_SERDR**. If a modem is used, some rewriting of **MODEM.LIB** is also required.

12.3 Software Implementation Overview

The first stage in dial-up PPP is to establish a modem connection with the ISP. The function **ModemInit()** opens the serial port, then detects if there is a modem connected and ready. It does this by sending "AT" to the modem a set number of times until it receives an "OK" response. This should work with any Hayes-compatible modem, which is the standard today. At this point the modem is ready and commands can be sent to it using **ModemSend()**. Remember to include a carriage return "\r" at the end of each command sent.

The function **ModemExpect()** is used to wait for a character sequence to occur. Normally the first use of this in a program is to determine that the modem has connected. When a connection occurs, the modem will send a string along the lines of "CONNECT AT x" or something similar. **ModemExpect()** can be set to listen for this. Once connected, the ISP may either attempt PPP negotiation immediately, or request a user name and password first. In the latter case, a sequence of **ModemSend()** and **ModemExpect()** calls are used to handle this (see See "Authentication Sequence" on page 304).

Eventually the ISP will begin PPP negotiation. At this stage **ModemClose()** should be called to shutdown normal serial operation. After calling **sock_init()** and doing any other necessary TCP/IP initialization, **PPPinit()** is called, followed by any necessary PPP option initialization, and finally a call to **PPPstart()**.

Once the PPP connection is established through a successful call to **PPPstart()**, the user can send packets to the peer using the TCP/IP libraries.

12.3.1 Defining Network Parameters

The following parameters must be defined at compile time:

// Sets the TCP/IP stack to use PPP. This should be "**pppoe.lib**" when using PPP over Ethernet.
```
#define PKTDRV "ppp.lib"
```

// In the most common case, the Rabbit will be dialing into an ISP. The ISP will usually wish to
// assign the IP address. Setting the IP address to 0.0.0.0 indicates that the Rabbit does not have a
// valid address when started. If the Rabbit has a permanent address or will be dialed into,
// **MY_IP_ADDRESS** should be set to a proper IP address.
```
#define MY_IP_ADDRESS "0.0.0.0"
```

// This is a parameter intended for Ethernet and other shared network schemes. Since PPP is a
// single point-to-point link, all traffic must be routed through a peer. There is no such thing as
// "local" traffic. A netmask of 255.255.255.255 causes all addresses to be routed to.
```
#define MY_NETMASK "255.255.255.255"
```

// This is the address of the host that will perform routing for the Rabbit. (With PPP this is always
// the peer.) If the ISP assigns a gateway machine to use, then define **MY_GATEWAY** to that. If the
// gateway is not known, defining **PPP_PEERROUTE** will make a gateway out of whatever
// machine you connect to, i.e. the Rabbit will use whatever address the peer uses when identifying
// itself during PPP negotiation. **PPP_PEERROUTE** will work under most circumstances, but a
// static gateway address may be needed for special cases.
```
#define MY_GATEWAY "10.1.1.1"
```

// This works the same as with Ethernet: it defines a host that will resolve names into IP addresses.
```
#define MY_NAMESERVER "10.1.1.2"
```

12.3.1.1 IP Addresses

When the Rabbit and the peer are establishing a connection, they negotiate what their IP addresses will be. When the Rabbit is connecting to an ISP, an IP address will be assigned to it by the ISP. In some cases, such as the Rabbit acting as a dial-in ISP, IP addresses for the Rabbit and the peer should be set by the Rabbit. This is done using **PPPnegotiateIP()**.

12.3.2 Configuration Options

The following configuration options are supported by the Rabbit PPP system:

Table 3. Configuration Options

LCP Configuration Option Type Field	Meaning of Option Type
01	MRU (Maximum-Receive-Unit)
02	ACCM (Async-Control-Character-Map)
03	Auth (Authentication-Type): PAP only
05	Magic Number
07	PFC (Protocol-Field-Compression)
08	ACFC (Address-and-Control-Field-Compression)

For more information on these options, refer to RFC 1661: The Point-to-Point Protocol (PPP) at:

http://rfc.asuka.net/rfc/rfc1661.html

12.3.3 Authentication

The PPP library supports an optional authentication phase. Both the authentication of a peer and being authenticated by a peer are done using Password Authentication Protocol (PAP). This is a simple two-way handshake only done upon initial link establishment.

The most common case is when the Rabbit must authenticate itself to the ISP it is connecting to. This is configured using **PPPsetAuthenticatee()**, which sets the username and password the Rabbit will use.

A different situation arises when the Rabbit needs to authenticate a connecting peer. This is necessary when the Rabbit is being dialed into. **PPPsetAuthenticator()** sets a name and password that will be required from the peer before a connection will be established.

12.3.3.1 Authentication Sequence

A common situation with dial-up PPP is that an ISP will want to authenticate the dialer before PPP negotiation. There are no real standards for doing this, so each ISP is potentially different. The best way to develop a correct sequence of **ModemSend()** and **ModemExpect()** commands is to connect to the ISP using a terminal program on a PC. You can then take note of the necessary sequence to start PPP negotiation.

Here is a hypothetical session as seen by a terminal program. Characters typed in and sent to the ISP or the modem are displayed in **bold**.

```
AT
OK
ATDT5554545
OK
CONNECT 28800
Welcome to someisp.com
Login?rabbit
Password:Ilikecarrots
Logging in as rabbit
Start PPP $*($}}}}}$}$#$#${@#>>}}FF}}$}
```

From this session we could use **ModemSend()** and **ModemExpect()** to create a dial-up function like this:

```
int myDialUp(){
   if(ModemOpen(57600) == 0){
      return 0;
   }
   if(ModemInit() == 0){
      return 0;
   }
   ModemSend("ATDT5554545\r");
   if (ModemExpect("OK", 2000) == 0)){
      return 0;     //something is wrong with the modem
   }
   if(ModemExpect("CONNECT", 30000) == 0){
      return 0;     //didn't connect to the ISP
   }
   if(ModemExpect("Login?", 5000) == 0){
      return 0;
   }
   ModemSend("rabbit\r");
   if(ModemExpect("word:", 5000) == 0){
      return 0;
   }
   ModemSend("Ilikecarrots\r");
   if(ModemExpect("PPP", 5000) == 0){
      return 0;     //probably a failed login
   }
   ModemClose();
   sock_init();
   PPPinit(57600);
   PPPflowcontrolOn();
   return 1; //all done
}
```

As you can see, **ModemExpect()** will pick up any part of the received string. Clever use of this allows the initialization to be fairly generic, but subtle differences between ISP's will often require customized sequences such as this.

12.3.4 Link Teardown

Tearing down the link must also be done in stages. First, a terminate request must be sent to the peer. This is done with **PPPshutdown()**. **PPPshutdown()** will return once an acknowledgement has been sent by the peer, or after a time out period. This is followed by a call to **PPPclose**, which unloads the PPP serial driver. If the connection is via a modem, the modem must then be hung up. First the regular serial driver is reopened with **ModemOpen()**. **ModemHangup()** sends the hang up and reset commands to the modem. Finally, a call to **ModemClose()** shuts down the serial driver.

12.4 Functions

This section describes the functions that compose the PPP driver and the functions for modem control.

Using Cofunctions

Establishing a PPP connection over a modem is time-consuming. Depending on the baud rate negotiated by the modem, the whole process can take 30 seconds or more. Much of this time is spent by the controller waiting for a response from the other end. In a practical application, where the controller has other tasks to perform, this may be unacceptable. For this, there are cofunction versions of all of the functions that wait for responses from the peer. There are still parts of the initialization process that create delays, but the effect is much smaller.

CofModemExpect

```
int CofModemExpect(char *send_string, unsigned long timeout);
```

DESCRIPTION

Listens for a specific string to be sent by the modem. Yields to other tasks while waiting for input.

PARAMETERS

send_string A **NULL**-terminated string to listen for.

timeout Maximum wait in milliseconds for a character.

RETURN VALUE

1: The expected string was received.
0: A timeout occurred before receiving the string.

LIBRARY

MODEM.LIB

CofModemHangup

```
int CofModemHangup();
```

DESCRIPTION

Sends "ATH" and "ATZ" commands. Yields to other tasks while waiting for responses.

RETURN VALUE

1: Success
0: Modem not responding

LIBRARY

MODEM.LIB

CofModemInit

```
int CofModemInit();
```

DESCRIPTION

Resets modem with AT, ATZ commands. Yields to other tasks while waiting for responses.

RETURN VALUE

1: Success
0: Modem not responding

LIBRARY

MODEM.LIB

CofModemSend

```
void CofModemSend(char *send_string);
```

DESCRIPTION

Sends a string to the modem. Yields to other tasks while sending.

PARAMETERS

send_string A **NULL**-terminated string to be sent to the modem.

LIBRARY

MODEM.LIB

CofPPPshutdown

```
int CofPPPshutdown(unsigned long timeout);
```

DESCRIPTION

Sends a Link Terminate Request packet. Waits for the link to be torn down.

PARAMETERS

timeout	Number of milliseconds to wait before giving up on a response from the peer. Yields to other tasks while waiting.

RETURN VALUE

1:Shutdown succeeded
0: Shutdown timed out

LIBRARY

PPP.LIB

CofPPPstart

```
int CofPPPstart(unsigned long timeout, int retry);
```

DESCRIPTION

Starts link negotiation process with a connected peer. Yields to other tasks.

PARAMETERS

timeout	The number of milliseconds to wait between phases of negotiation before starting over.
retry	Number of times to retry the connection

RETURN VALUE

1: Negotiation succeeded;
0: A link could not be negotiated.

LIBRARY

PPP.LIB

ModemClose

```
void ModemClose();
```

DESCRIPTION

Closes the serial driver down.

LIBRARY

MODEM.LIB

ModemConnected

```
int ModemConnected();
```

DESCRIPTION

Returns true if the DCD line is asserted, meaning the modem is connected to a remote carrier.

RETURN VALUE

1: DCD line is active
0: DCD inactive (nothing connected)

LIBRARY

MODEM.LIB

ModemExpect

```
int ModemExpect(char *send_string, unsigned long timeout);
```

DESCRIPTION

Listens for a specific string to be sent by the modem.

PARAMETERS

send_string	A **NULL**-terminated string to listen for.
timeout	Maximum wait in milliseconds for a character

RETURN VALUE

1: The expected string was received
0: A timeout occurred before receiving the string

LIBRARY

MODEM.LIB

ModemHangup

```
int ModemHangup();
```

DESCRIPTION

Sends "ATH" and "ATZ" commands

RETURN VALUE

1: Success
0: Modem not responding

LIBRARY

MODEM.LIB

ModemInit

```
int ModemInit();
```

DESCRIPTION

Resets modem with AT, ATZ commands.

RETURN VALUE

1: Success
0: Modem not responding

LIBRARY

MODEM.LIB

ModemOpen

```
int ModemOpen(unsigned long baud);
```

DESCRIPTION

Starts up communication with an external modem.

PARAMETERS

baud	The baud rate for communicating with the modem.

RETURN VALUE

1: External modem detected
0: Not connected to external modem

LIBRARY

MODEM.LIB

ModemReady

```
int ModemReady();
```

DESCRIPTION

Returns true if the DSR line is asserted.

RETURN VALUE

1: DSR line is active
0: DSR inactive (nothing connected)

LIBRARY

MODEM.LIB

ModemRinging

```
int ModemRinging();
```

DESCRIPTION

Returns true if the RI line is asserted, meaning that the line is ringing.

RETURN VALUE

1: RI line is active
0: RI inactive (nothing connected)

LIBRARY

MODEM.LIB

ModemSend

```
void ModemSend(char *send_string);
```

DESCRIPTION

Sends a string to the modem.

PARAMETERS

send_string A **NULL**-terminated string to be sent to the modem.

LIBRARY

MODEM.LIB

ModemStartPPP

```
void ModemStartPPP();
```

DESCRIPTION

Hands control of the serial line over to the PPP driver.

LIBRARY

MODEM.LIB

PPPclose

```
void PPPclose();
```

DESCRIPTION

Closes the serial port and unloads the PPP interrupt service routine.

LIBRARY

PPPLINK.LIB

PPPinit

```
void PPPinit(unsigned long baud)
```

DESCRIPTION

Initializes the PPP driver, sets parameters. Must be called immediately following a call to **sock_init()**.

PARAMETERS

baud	The baud rate of the serial port PPP is running on (port C by default).

LIBRARY

PPP.LIB

PPPflowcontrolOff

```
void PPPflowcontrolOff()
```

DESCRIPTION

Deactivates hardware flow control for the serial link.

LIBRARY

PPPLINK.LIB

PPPflowcontrolOn

```
void PPPflowcontrolOn()
```

DESCRIPTION

Activates hardware flow control for the serial link. The pins used for flow control are defined in **PPPLINK.LIB** as follows:

PPP_CTSPORT: the port address for the CTS input line.
PPP_CTSPIN: the pin number of the CTS input line.
PPP_RTSPORT: the port address of the RTS output line.
PPP_RTSSHADOW: the name of the port's shadow register.
PPP_RTSPIN: the pin number of the RTS output line.

LIBRARY

PPPLINK.LIB

PPPstart

```
int PPPstart(unsigned long timeout, int retry);
```

DESCRIPTION

Starts link negotiation process with a connected peer.

PARAMETERS

timeout	Number of milliseconds to wait between phases of negotiation before starting over.
retry	Number of times to retry the connection.

RETURN VALUE

1: Negotiation succeeded;
0: A link could not be negotiated.

LIBRARY

PPP.LIB

PPPnegotiateIP

```
void PPPnegotiateIP(unsigned long local_ip, unsigned long
   remote_ip);
```

DESCRIPTION

Sets PPP driver to negotiate IP addresses for itself and the remote peer. Otherwise, the system will rely on the remote peer to set addresses.

PARAMETERS

local_ip	IP number to use for this PPP connection.
remote_ip	IP number that the remote peer should be set to.

LIBRARY

PPP.LIB

PPPnegotiateDNS

```
void PPPnegotiateDNS(unsigned long dns_ip);
```

DESCRIPTION

Sets PPP driver to configure a DNS address for the remote peer.

PARAMETERS

dns_ip IP number for the DNS server

LIBRARY

PPP.LIB

PPPsetAuthenticatee

```
void PPPsetAuthenticatee(char *username, char *password);
```

DESCRIPTION

Sets the driver up to send a PAP authentication message to a peer when requested.

PARAMETERS

username The username to send to the peer. The argument string is not copied, so the argument string must stay constant.

password The password to send to the peer. The argument string is not copied, so the argument string must stay constant

LIBRARY

PPP.LIB

PPPsetAuthenticator

```
void PPPsetAuthenticator(char *username, char *password);
```

DESCRIPTION

Sets the driver up to require a PAP authentication message from a peer. Negotiation will fail unless the peer sends the specified username/password pair. This function is generally used when the Rabbit is acting as a dial-in server.

PARAMETERS

username The user name that the peer must match for the link to proceed.

password The password that the peer must match for the link to proceed.

LIBRARY

PPP.LIB

PPPshutdown

```
int PPPshutdown(unsigned long timeout);
```

DESCRIPTION

Sends a Link Terminate Request packet. Waits for link to be torn down.

PARAMETERS

timeout Number of milliseconds to wait before giving up on a response from the peer.

RETURN VALUE

1: Shutdown succeeded
0: Shutdown timed out

LIBRARY

PPP.LIB

ResetPPP

```
void ResetPPP( );
```

DESCRIPTION

Under normal operations, this function will not be needed; the modem control functions make it unnecessary. There are, however, conditions that may make it useful.

LIBRARY

PPP.LIB

Index

Numerics

A

B

C

D

E

F

H

I

M

N

P

R

S

T

U

W