InterNiche's embTCP User's Guide for PIC32MX/MZ (MPLABX Tools)

51 E. Campbell Ave
Suite 160
Campbell, CA. 95008

InterNiche Technologies Inc.
email: Sales@iNiche.com
support: http://www.iniche.com/support

Revised: November 6, 2013

Table of Contents

Overview

This Technical reference is provided with InterNiche Technologies' **embTCP** and **embDUAL** networking "C" libraries. The purpose of this document is to provide enough information so that a moderately experienced "C" programmer with a reasonable understanding of TCP/IP protocols can develop client and/or server applications using MPLABX development tools.

The primary features of this library are:

- Small footprint
- Pre-ported to the FreeRTOS Operating System (source code included)
- "Device Locked" to PIC32MX/MZ **Important Note:** The software described in this document will not run on any component other than the PIC32MX/MZ. For support of another controller, contact InterNiche Sales: Sales@iNiche.com
- Sample Applications demonstrating simultaneous IPv4 and IPv6 operation (if using embDUAL, otherwise examples only function for IPv4 communication)
- Menu System and Command Line Interface
- DEBUG and Non-DEBUG "C" libraries are provided.

- Flexible Packet Buffer mechanism
- TCP, IPv4 and UDP (also IPv6 if using embDUAL)
- Multicast support for IPv4 (also IPv6 if using embDUAL)
- BSD Sockets
- DHCP Client (may be disabled)
- DNS Client
- Ping application
- Ethernet driver
- An Ethernet interface as well as a loopback interface (`127.0.0.1` and `::1`) are supported.
- An internal clock rate of 20 ticks per second

A Note About this Document

Unless specifically mentioned otherwise, the term **embTCP** is intended to apply to both the `embTCP` and `embDUAL` embedded library products.

Product Requirements

Installation

Before you start using this product, it is important that you have successfully built, downloaded and executed some small program to your PIC32MX/MZ based board using MPLABX development tools. It is not particularly important that this program run the FreeRTOS operating system, but only that you have some end-to-end experience with your entire development environment and that you have confidence that your hardware works.

Product Registration

As provided, embedded Libraries contain license information that will only allow it to operate for a finite period of time before halting. Registration is as simple as visiting www.TCPIPStack.com, submitting a simple form and checking your email for a `tcp_license.obj` file that should be used instead of `tcp_unregistered.obj`

Development Environment

1. Begin with a project known to build, download and run on your PIC32MX/MZ-based board
2. Add source files to the project:
    - `embsrc` directory: Add every file
    - If you already have FreeRTOS, skip to the next section. For PIC32MX use FreeRTOSv7.3.0; PIC32MZ use FreeRTPSv7.5.2
    - `FreeRTOSv7.x.x` directory: Add every source code file
    - `FreeRTOSv7.x.x/Source/portable/MPLAB/PIC32xx` directory: Add every source code file
    - `FreeRTOSv7.x.x/Source/portable/MemMang` directory: add the management scheme most appropriate for your environment. (Note: During development of embTCP, the file `heap_3.c` was most heavily tested of the three methods provided with FreeRTOS.)
3. Add the following library files (`Project Properties | Conf | Libraries`)
    - `emblibs/libembtcp-debug.a`
    - `emblibs/tcp_license.o` **
4. Add the following Preprocessor include directories (`Project Properties | Conf | XC32 (Global Options) | xc32-as | Preprocessor Include directories`)
5. `emb_h`
6. Add the following include directories (`Project Properties | Conf | XC32 (Global Options) | xc32-gcc | Include directories`)
    - `FreeRTOSv7.x.x/Source/include`
    - `FreeRTOSv7.x.x/Source/portable/MPLAB/PIC32xx`
    - e`mb_h`
7. Add the following additional options `(Project Properties | Conf | XC32 (Global Options) | xc32-gcc | Additional options)`
    - `-minterlink-mips16`
8. Add the linker options `(Project Properties | Conf | XC32 (Global Options) | xc32-ld)`
    - Heap size (bytes) – `71680`
    - Minimum stack size (bytes) - `2048`
9. Modify the following project setting for `FreeRTOSv7.x.x/source/portable/MPLAB/PIC32xx/port.c (Project Properties | Conf | XC32 (Global Options) | xc32-gcc)`:
    - Clear the checkbox labeled "Genearte 16-bit code">

** NOTE: If you have not yet registered your product, use `tcp_unregistered.o` instead. Registration will enable full use of the library and is accomplished by visiting www.TCPIPStack.com.

Project Integration

This product is delivered pre-integrated with the FreeRTOS Operating System (v7.x.x). If you already have a FreeRTOS based product, then integration of your new communications libraries consists of following these steps:

- Locate your project's `main()` function and invoke `nichestack_init()` after any operating system initialization.

Similarly, if you are beginning with the OS as provided by InterNiche:

- Examine the file `inmain.c` and call `nichestack_init()` immediately following the call to `TK_OS_INIT()`.

Special Note Regarding FreeRTOS and embTCP/embDUAL

1. ***All application tasks which will take advantage of embTCP networking must be created with `TK_CREATE()` instead of the `xTaskCreate()` function. All parameters are the same.***
2. ***The maximum number of tasks that will be be created using `TK_CREATE()` must be communicated to embTCP by setting the `max_user_tasks` variable before stack initialization.***

---

example1.c - A TCP Echo Client

The TCP Echo Client example application is found in the directory tcp_examples/example1. It connects to a TCP echo server over IPv4 or IPv6, periodically sending echo requests and then validating that the data in the received response exactly matches the data sent.

Before compiling this example, you must change the "svraddrstr" parameter in example1.c to the IP address of a TCP echo server. The other parameters should not be changed for this initial test. Most Windows and Linux systems have a running TCP echo server that will respond to echo requests.

Note: The TCP Echo Client task of `example1` begins running at the end of system initialization. It will immediately attempt to connect to the TCP Echo Server running at the specified address. This means the server needs to be running when the embTCP system is initialized. Otherwise, the connection request will timeout and the task will return to its task loop, where it will be deleted.

Installing and Running example1

To include this example application in your project:

1. Add `tcp_examples/example1/example1.c` to the project as is appropriate for your development tools.
2. Modify the `get_mac_address()` routine in `tcpdata.c` to correctly obtain the MAC address for your board. For debugging purposes, you could temporarily set the MAC address to any non-multicast value unique to your local network. (In a non-multicast address, `local_mac[0]` will have an even value).
3. Edit `tcpdata.c` and locate the function call to `in_v4addrcfg( )`. If you wish to configure the use of a DHCP server to configure your project, change the final parameter from `0` to `NETF_DHCPC`. Otherwise, change the IPv4 addresses to values that work for your local network
4. Edit example1.c and change the default value of the `svraddrstr` parameter from `0.0.0.0` to the IP address of a TCP echo server. If you do not have an available TCP echo server, feel free to ignore the error messages that appear on "stdout".
5. Edit the file `embsrc/inmain.c` and remove the comments surrounding the call to `example_init()`.
6. Verify that the target board is connected to the LAN, and that a DHCP Server exists on the network
7. Compile, download, run

Once your project begins to execute, it will display a message similar to the following on your "stdout":

```
InterNiche Embedded TCP/IP, v2.1emb (xxxxxxxx)  (FreeRTOS)
Copyright 2013 by InterNiche Technologies. All rights reserved.

Licensed to: COMPANY, email@example.com
For PRODUCTNAME XXXX-XXXX-XXXX-XXXX

Acquired IP address via DHCP client for interface: et1
IP address : 10.0.0.211
Subnet Mask: 255.255.255.0
Gateway    : 10.0.0.1
```

The `XX`s in the IP address above represent the TCP address that has been assigned to the system.

This message indicates that embTCP was able to able to communicate with the DHCP server over the network in order to configure a DHCP address.

Shortly after the system has acquired a DHCP address, you should see a series of messages starting with:

```
tcp_echo_client is starting
sending TCP echo request 1 to xx.xx.xx.xx
TCP echo: received correct response
```

Additional TCP echo request and receive messages should appear about once a second until the command completes.

Sample Application Walkthrough

The file example1.c begins with some configuration parameters. With the default configuration, 10 echo requests of 255 bytes each will be sent to port 7 of the server specified by svraddrstr. TCP echo servers listen for requests on port 7. The 10 requests will be sent at the rate of 1 per second.

The TCPCLIENT structure holds the input and output buffers and the variables that are maintained across all of the echo requests.

The next section defines the TCP task info structure.

The function TK_ENTRY(tk_tc_echo) would normally be the main task loop. Most tasks run for the life of the system. However, the TCP echo task runs only long enough to send the configured number of echo requests. When it returns to the task loop, the task is deleted. Tasks must not exit or call return, but they can delete themselves.

The tec_init() routine does the following:

- Converts the IP address string to a binary number
- Allocates a TCPCLIENT structure.
- For each IP protocol (IPv4 and IPv6):
    - Sets the tec->nextsend to the current time so that an echo request will be sent as soon as we are ready.
    - Fills out the address structure that will be used to contact the server.
    - Opens a socket and calls t_connect(). The connect routine blocks and will not return until either the connection has been made or it times out.
    - Calls t_setsockopt() to put the socket in non-blocking mode.
- Sets the tec->nextsend to the current time so that an echo request will be sent as soon as we are ready.
- Calls tcp_send_an_echo to send the first echo request.
- If there is no error, it calls tcecho_loop() to send the rest of the requests.

The tcp_send_an_echo() routine does the following:

- Fill each byte of the outgoing buffer with values incrementing from 0 to ECHODFTLEN - 1.
- If the system is congested, then it is possible that only part of the data will be sent with the first t_send(). The tcp_send_an_echo routine will loop until all bytes in the request are sent.
- If the t_send call blocks, we call TK_YIELD to let other tasks run. On return we continue in the send loop.
- If t_send return a positive value, then we add the return to the number of bytes sent.
- Once the full echo request has been sent, we increment the statistics variables and set the timer for when the next request should be sent.

The tcecho_loop does the following:

- goes into a loop. The "while (1)" expression means it will only exit when "return" is called.
- Calls t_select() to see if there are any responses ready to be read. It sets the timeout value for t_select to 4 ticks. This allows it to test to see if it is time to send another echo request.
- Calls t_recv() to read any available echo response.
- If the return from t_recv ("len") is < 0 it calls t_errno to determine the error number. It prints an error message if the return is anything other the EWOULDBLOCK.
- If "len" is greater than 0, then we read a echo response. Note if the system were congested, the full response may not be read in a single t_recv() call.
- Loops to validate each byte of the received data. The value of each byte is an incrementing number from 0 to 254. The variable "tec->nextrcvbyte" hold the value of the next expected byte. We tec->nextrcvbyte increments to 255, it is reset to 0. If we find an unexpected character, we print and error message and close the connection.
- After a successful read, "tec->replies", the number of replies is incremented and "len" is added to "tec->tot_rcvd", the cumulative total number of bytes read.
- When "tec->send_count", the number of requests sent, equals the configured number of requests to send, tcp echo has completed. It call tcp_client_close() to clean up and returns.
- Otherwise, if it is time to send another request, it cals tcp_send_an_echo.

When the TCP echo client is ready to close for any reason, it calls tcp_client_close() to clean up the connection. The tcp_client_close() routine does the following:

- If there is a valid socket, it closes it.
- If there is an allocated TCPCLIENT structure, it frees it.

Other TCP Examples

The tcp_examples directory contains 3 other examples:

- Example 2. A TCP echo server
- Example 3. A UDP echo client
- Example 4. A UDP echo server.

The directory for each example contains a `readme.txt` file describing the example in detail.

Debug vs Non-Debug Libraries

embTCP includes two versions of the library: Debug, which is intended for use during initial application development; and Non-Debug, which is appropriate for use in your final product. In addition to the presentation of messages and "debug printf()s", the Debug compilation provides two API functions: `dtrap()` and `panic()`. These are reduced to empty functions in the non-debug library. **Note: Be sure to link the appropriate library as you build your own "Debug" and "Release" products.**

An Important Note Regarding Stack Sizes

It is important to recognize that the task stack size requirements must be set appropriately for the unique requirements of your application and the requirements of your final product. Failure to properly tune the stacks will result in either wasted memory or nearly impossible to diagnose runtime errors.

The size of embTCP's task stacks are specified in the `tcpdata.c` file. Please refer to FreeRTOS.org for information regarding stack sizing and the debugging of stack overflow conditions.

Name

`dtrap ()`

Syntax

```
void dtrap (void);
```

Parameters

None.

Description

The dtrap() routine is called by the debug version of embTCP whenever it detects a situation that should not be occurring. The function prints the message "dtrap" and then goes into a forever loop. In general, you should eliminate or understand all causes of a call to dtrap before moving on to the release version of the code.

Returns

Nothing

Notes

- `dtrap()` resides in `embsrc/tcpdata.c` and should be modified to suit the needs of the application developer.
- In Release-mode (non-Debug), embTCP will not call dtrap().

Name

`panic()`

Syntax

```
void panic(char *msg)
```

Parameters

msg	printed on "stdout"

Description

The panic() routine is called if the software detects a fatal system error. It will print and error message, call dtrap() in the debug version, and then call exit(1). You may want to add code to this routine to attempt to restart the system.

Returns

Never

Configuration

Tailoring and tuning of the library is accomplished through the `tcpdata.c` file, provided in source code. It is in this file that you will be able to specify configuration variables for the various tasks created by the library, configure the width and depth of the packet buffer pools, and include or remove CLI commands from being linked into your final application. The paragraphs below discuss the various parts of this module in the order that they appear.

Local Mac Address

Each device on an Ethernet-based Local Area Network must have a unique 48-bit MAC address. Since these addresses are board specific, the application/porting engineer will have to determine how to read this value for his/her specific hardware and make it available through the `get_mac_address()` function, located in `embsrc/tcpdata.c`.

Modules Array

There are four modules specified in the `in_modules` array for embTCP. The `cli_module` can be removed if your system does not use menus. Similarly, the `console_module` can be removed if your system does not have a console. The `ipv6_module` is a 'stub' for embTCP and is fully available only in the embDUAL configuration.

If you have purchased additional InterNiche embedded libraries you will need to add their modules to this array, e.g., the telnet_module for embTELNET and the http_module and wbs_module for embHTTP.

max_user_tasks

`max_user_tasks` is the maximum number of application tasks that you will run. This value must be set in `embsrc/tcpdata.c`.

Buffer Queues Array

The `in_bufq[ ]` array defines the number and sizes of the buffers used to hold incoming and outgoing data packets. Each element specifies the number of packet buffers of a specific size that will be allocated at initialization time. While one could configure packet buffers of many different sizes, it normally works best to have pools of only 2 or 3 different sizes.

If there is no buffer large enough to hold a specific packet and there are enough smaller buffers, embTCP will chain buffers together in order to create a buffer of the required size. For this reason, having many smaller buffers (512 bytes or less) often makes more efficient use of buffer space than having fewer but larger buffers.

TCP Stack Configuration parameters.

The function embTCP_config() is used to configure the variables discussed below. Please see the file `tcpdata.c`. The formats of the structures are defined in stkdata.h, but you should not need to refer to these. The following table describes each variable configured by `embTCP_config()`:

Section	Name	Default Value	Description
uint16_t	netmain_stksize	3072	Default size in bytes of TCP/IP's OS stack
uint16_t	nettick_stksize	3072	Default size in bytes of the network timer's OS stack
uint16_t	console_stksize	3072	Default size in bytes of the embLib console's OS stack
DHCP Client	dhcp.max_tries	4	Maximum number of transmissions of DHCP packet.
	vendclass	0	This is an array that contains the vendor class identifier. This field can be upto DHCP_VENDCLASS_MAXLEN (32) bytes long. As its name implies, this field is vendor-specific.
	vendclass_len	1	The length (in bytes) of the vendor class identifier field.
DNS Server	dns.max_tries	4	Maximum times (including the first) that a specific DNS name resolution request will be sent before DNS gives up and returns an error.
	dns.retx_interval	6	Interval in seconds between retrying DNS requests for a specific name resolution.
	dns.max_entries	6	Number of entries in the DNS client's table. An entry can contain information from a resolved request or a name resolution request that is in progress.
	dns.servers[ ]	`{ "0.0.0.0", "0.0.0.0", "0.0.0.0" }`	DNS Server List: The IP address of DNS servers. Each address is entered as a standard IP address string in dotted notation. The maximum number of DNS server addresses (`MAXDNSSERVERS`) is `3`
ICMP	ping.enable_resp	TRUE	Enable/Disable the sending of a ping response when a ping request is received.
	ping.count	4	Default number of pings to send for one ping request
	ping.interval	20	Default interval in ticks between ping requests
	ping.length	64	Default data length in a ping request
	ping.quiet	0	Disable printing results of each ping (Default results are printed)
	ping.giveup_time	60	Default time (in ticks) to wait for last ping response before ending session
	ping.maxlength	1600	Maximum length of an outbound ping. This value plus the length of IP, ICMP and fragmentation headers must be less than `ipv4.reasm_max_mem` to ensure that the response can be received.
IPv4	ipv4.reasm_tmo	20	Maximum amount of time (in seconds) to wait for reassembly of a fragmented IPv4 datagram to complete.
	ipv4.reasm_max_mem	4096	Upper limit on the amount of memory that can be consumed by the IPv4 reassembly module. This includes memory consumed by the received fragments and the reassembly control data structures.
	ipv4.tos	0	Default value of Type of Service (TOS) field in IPv4 header of outgoing packets.
	ipv4.ttl	64	Default value of Time to Live (TTL) field in IPv4 header of outgoing packets.
IPv6	ipv6.hoplim	255	Default value of Hop Limit field in IPv6 header of outgoing unicast packets
	ipv6.ndcache_len	8	Maximum number of (resolved or unresolved) entries in the Neighbor Discovery cache

	ipv6.pfxlist_len	4	maximum number of entries in the IPv6 prefix list
	ipv6.reasm_max_pkts	4	Maximum number of IPv6 datagrams that can be simultaneously reassembled
	ipv6.reasm_tmo	60	Maximum amount of time (in seconds) to wait for reassembly of a fragmented IPv6 datagram to complete
	ipv6.sndq_max	3	Maximum number of packets queued waiting for IPv6 address resolution
TCP	tcp.conn_estab_tmo	75	The maximum time in seconds that embTCP will wait for a response to a connection request before it will return an error.
	tcp.msl	20	Maximum segment lifetime.
	tcp.enable_noport_rst	TRUE	Enables the sending of an RST when a connection request is received for a non-existent port. (No server listening on this port).
	tcp.recv_space	8760	The maximum amount of received data that can be queued for an application.
	tcp.send_space	8760	The maximum amount of data that can be queued in the send buffer.
	tcp.kal.idle_time	600	The length of time a session can be idle before embTCP will start sending keep-alive probe messages.
	tcp.kal.numprobes	8	The maximum number of probes that will be sent before the connection will be closed.
	tcp.kal.probe_interval	15	Time in seconds between keep-alive probes.
TCP/IP Stack	stk46.in_l3prots	(product dependent)	This variable controls which of the available Layer 3 protocols (IPv4 and/or IPv6) will be enabled in the system at run-time. For use with a dual-stack library, this parameter can be set to any one of the following values: IPV4_ENABLED, IPV6_ENABLED, or IPV46_ENABLED. For use with a library only supporting IPv4, this variable can only be set to IPV4_ENABLED.
UDP	udp.enable_dest_unreach	TRUE	Enables the sending of ICMP Destiniation Unreachable messages when a UDP connection request is received for a non-existent port. (No server listening on this port).
	udp.rcvbuf_size	4096	UDP receive buffer size. This is the maximum amount of data that can be queued waiting to be read. It also limits the maximum size of a UDP message that can be received.

FreeRTOS Task Parameters

These parameters are used to configure the tasks that are internal to embTCP, they are only read during embTCP initialization and should not be modified at runtime. Changing these values may affect the performance and stability of the embTCP product, and should only be undertaken by someone knowledgeable in the internals of the FreeRTOS operating system.

max_user_tasks	1	The maximum number of user tasks that will be created using TK_CREATE. Tasks that use the embTCP API must be created using TK_CREATE.
netmain_priority	5	Task priorities of the internal embTCP tasks. The priorities of these 3 tasks can be reassigned to accommodate other user tasks, but the relative priorities of these 3 tasks should be preserved.
nettick_priority	3	
console_priority	2	
netmain_stksize	1536	Task stack sizes in bytes. These sizes are based testing with the various example programs. The stack sizes may need to be increased, depending on the functions that are executed in the context of each task.
nettick_stksize	1536	
console_stksize	1536	
free_rtos_tickrate_hz		Must be set to the constant, configTICK_RATE_HZ, defined in FreeRTOSconfig.h.

Ethernet Device Driver Parameters

The ethernet device driver uses arrays of data descriptors to describe the network data packets being transmitted and received. As receive descriptors are filled with data they are removed from the descriptor array and combined into data packets. The receive descriptors are replaced with data packets from an internal data packet pool. The size of each data packet in the pool is specified by '`eth_segment_size`'. For best results, `eth_segment_size` should match the size of one of the buffer queues.

The number of transmit and receive descriptors are specified by `eth_tx_num` and `eth_rx_num`. The number of receive descriptors should be enough to contain several worst-case data packets. The number of transmit descriptors should be larger than the number of descriptors required to contain a worst-case data packet.

eth_segment_size	512	The size (in bytes) of each receive descriptor.
eth_threshhold	6*512	The target size (in bytes) of the internal data packet pool.
eth_rx_num	8	The number of received descriptors.
eth_tx_num	16	The number of transmit descriptors.

User Modifiable Functions

User Pre-Setup

The `user_pre_setup()` routine is located in `tcpdata.c`. The function is called early in the initialization process before the embTCP initializes it modules and the devices. Additional code may be added to this routine to do any needed system initialization that is not part the normal embTCP port. This could include initializing devices, such as USB or video displays which are part of the developer's product.

By default `user_pre_setup()` copies the configuration parameters defined above in the embTCP structures and uses the `ip_v4addrcfg()` function to set up the IP addresses for the interface. The following is the definition of `ip_v4addrcfg()`:

Name

`in_v4addrcfg ()`

Syntax

```
int in_v4addrcfg(char *name, char *addr, char *subnet_mask, char *gateway, uint32_t flags);
```

Parameters

name	Name of the Ethernet interface (e.g., "et1").
addr	IP address of the interface in dotted decimal notation.
subnet_mask	The network subnet mask in dotted decimal notation.
gateway	Address of the default router (gateway) in dotted decimal notation.
flags	Bitwise-OR of: • NETF_DHCPC • NETF_AUTOIP to enable DHCP client and Auto-IP assignment as implied by their names

Description

This function is used to provide addressing information for an interface. It is only expected to be invoked at startup time (from user_pre_setup() in tcpdata.c). If the system uses DHCP (as indicated via the 'flags' parameter), all of the address parameters can be specified as zero.

Returns

This function returns EFAILURE if the interface name is incorrect or if any of the address strings cannot be parsed successfully. Otherwise, it returns ESUCCESS.

User Post Setup

The `user_post_setup()` routine is called after the system, modules, and devices have been intialized, but before the system begins its main run loop. The user could add any needed application initialization code here.

UPNP Callback

Name

`upnp_callback()`

Syntax

```
void upnp_callback(char *name, int status);
```

Parameters

name	Interface name
status	The results of the IPv4 address acquisition process:

- UPNP_DHCP_ADDR: Address was obtained via DHCP
- UPNP_AUTOCONF_ADDR: Address was obtained via Auto IP
- UPNP_STATIC_ADDR: Configured Static address was used
- UPNP_ZERO_ADDR: No IPv4 address was obtained

Description

The upnp_callback() routine is the registered callback handler for the IPv4 address acquisition manager. It will be called to indicate the result of address acquisition. The default version of this routine prints a status message.

Returns

Nothing.

Header files in emb_h

NOTE: Do not make changes to any of these header files.

The following files in this directory simply provide required defines, structure definitions, extern function prototypes.

stkdata.h	Provides many defines needed by the embTCP. It defines the structures used by `embTCP_config()` and the component modules structures used for the configuration parameters described in the Stack Configuration Parameters section. It also defines the structures used to display embTCP statistics.
embcli.h	• Defines the error values that may be returned by calls to embTCP menus. • It defines the CLI data types (CLI_INT, CLI_STRING, etc.) • Defines structures for various elements of the CLI menu system These features are described in detail in the Menu System section.
tcpdata.h	Defines the parameters to the menu commands given in tcpdata.c and provides the prototypes for CLI command processing functions
embtcp.h	• Provides the function prototypes for the socket API. See the section Sockets API section for a detailed description of each of these functions. • Defines most of the TCP, sockets, IPv4 and IPv6 data structures • Defines the `fd_set` structure and a number of other defines needed for setting the file descriptors for the t_select() socket API. • Defines the option flags that may be used with the `t_getsockopt( )` and `t_setsockopt( )` sockets API • Defines the error codes used by embTCP. Those with positive values are the standard socket error codes returned by t_errno() when a socket call returns a failure indication. The negative error codes are error values that may directly appear in the returns from embTCP function calls. • Defines some helper functions that may be used to handle host to network byte-order conversion and for dealing with network addresses. • Time related defines and helper functions. • Prototypes for the embTCP ping and DHCP client applications • Task initialization and control structure and function definitions and externs for task variables and functions.
embdns.h	• Defines the `hostent` and `addrinfo` structures • Defines the flags that are used in a DNS query • Defines the flags that can be used with the getaddrinfo API • Defines the error codes that may be returned from the getaddrinfo API • Provides the function prototypes for the DNS client API

Memory Organization and Buffer Management

Socket data is maintained internally in packets. A packet consists of a linked list of packet buffers. A packet is contiguous if has only one packet buffer. A packet is chained if more than one packet buffer is used to contain the packet's headers and data. When data is written to a socket, the Stack allocates a packet large enough to hold the data, copies the data into the packet buffer(s), and appends the packet to the socket's "send" queue. When data is read from a socket, the packet is removed from the socket's receive queue, the data is copied into the application's buffer and the packet is freed. The application writer does not need to be familiar with the internal organization of a packet.

Unused packet buffers are kept in free queues, sorted by packet buffer size. When a packet allocation request is made, one or more packet buffers are removed from the queue(s) and linked together to form a chained packet of sufficient length. The number of free queues and the size of packet buffers in each queue is configured by the porting engineer. The minimum packet buffer size is 128 bytes. The maximum packet buffer size will depend upon available target memory, device driver requirements, and the protocols being used.

Memory Tracking

It is often useful to know how much dynamic memory is being used by an embedded application. Having this information allows developers to tune RAM memory usage or possibly identify memory leaks. Memory tracking is implemented as part of the embTCP memory management functions. The memory usage statistics can be displayed with a CLI command.

The embTCP memory functions are implemented in in_memory.c. Memory tracking is enabled at compile-time when the symbol `TRACE_MEMORY` is defined in `in_memory.c`. Memory tracking uses an array of (address, size) pairs to record all of the memory allocation, frees, and reallocations that occur through the embTCP dynamic memory interface. The size of the array is defined by the symbol `IN_MSIZE` in `in_memory.c`. The value of IN_MSIZE may need to be increased if the application defines a large number of dynamically allocated structures, such as packet buffers, sockets, tasks, etc.

Whenever a block of memory is allocated by a call to `npalloc()`, the address of the block and its size are recorded in the memory trace array. When to block of memory is freed, corresponding entry in the array is deleted. Statistics are maintained about the number of blocks allocated and their total size in bytes. This information is available as part of the CLI "`queues`" command output. A typical memory statistics display is:

```
npallocs: 100/33152; max = 116/36156; frees = 502/83960; realloc = 0
```

These numbers indicate that there are currently 100 blocks of memory allocated in the system. These 100 blocks consume 33152 bytes of dynamic memory (i.e. heap). The maximum number of blocks that were ever allocated was 116 blocks. The maximum number of bytes that were ever allocated was 36156 bytes. Note that the maximum number of allocated blocks and the maximum number of allocated bytes may have occurred at different times in the execution of the application. There have been 502 calls to npfree() to free a total of 83960 bytes of dynamic memory. The total number of calls to `npalloc( )` is 100+502=602. There have not been any calls to `nprealloc( )`.

When embTCP is started, tasks are created, packet buffers are allocated, and internal tables are allocated. These structures tend to exist for the lifetime of the application. The number and total size of these structures represent the majority of the "100/33152" statistics. As new socket connection are made, socket structures are allocated. When the socket connection is closed, these same structures are freed. These types of transient data structures account for the difference between the current allocation and the maximum allocation statistics. The "free" statistics should never decrease during the execution of an application.

NOTE: If the maximum number of allocations continues to increase during the execution of an application, this might be an indication of a memory leak. If the memory trace array becomes full, embTCP will panic with a "memory trace overflow" message on the Console. The size of the memory trace array can be increased by increasing the value of `IN_MSIZE` and recompiling `in_memory.c`

The file `in_memory.c` is provided in source form as part of the embTCP `embsrc` directory. Developers can modify the memory trace code to add additional trace capabilities.

Module Initialization

At each stage of the embTCP initialization, each module's function is called to perform its initialization. When all modules have successfully completed a stage, initialization progresses to the next stage.

There are several boolean variables that can be tested to monitor the stages of the initialization process. These variableS are set to FALSE when initialization begins:

iniche_init_done	Set to TRUE during post_task_setup() processing. Indicates that all NicheStack resources have been created and initialized.
iniche_net_ready	Set to TRUE when that network is up and that tasks can make calls into the Stack.
iniche_os_done	Set to TRUE within the TK_OS_START() macro. Used by NicheTask to inform the scheduler that task initialization is done and task scheduling may begin.

The ARM archicture stores bytes within a word or long word in "little-endian" host byte-order. Network data, on the other hand, is stored in "big-endian" network byte-order. The macros, htons() and ntohs(), can be used to convert 16-bit quantities between host byte-order and network byte-order. The macros, htonl() and ntohl(), can be used to convert 32-bit quantities between host byte-order and network byte-order.

```
#define htonl(l) ((((l >> 24) & 0x000000ff)) | \
                  (((l >>  8) & 0x0000ff00)) | \
                  (((h) & 0x0000ff00) <<  8) | \
                  (((h) & 0x000000ff) << 24))
#define ntohl(l) htonl(l)
#define htons(s) ((((s) >> 8) & 0xff) | \
                  (((s) << 8) & 0xff00))
#define ntohs(s) htons(s)
```

This product is delivered pre-integrated with the FreeRTOS Operating System (v7.x.x). If you are beginning with the OS as provided by InterNiche, the integration of your application with embTCP consists of the following:

1. Modify the configuration parameters in tcpdata.c as needed.
2. Link your application with the embedded libraries.

embTCP is initialized with a call to nichestack_init(), which has the following prototype:

```
int  nichestack_init(unsigned int flags);
```

Nichestack_init returns 0 on success and -1 if there was an error. Currently the flags value is not used. Nichestack_init should be called after the board and FreeRTOS have been initialized, but before the call to start the OS.

API

Name

`start_ping4()`

Syntax

```
int start_ping4(ip_addr dest, uint16_t count, int length, int delay);
```

Parameters

dest	IPv4 destination address (host byte order)
count	Number of echo requests to send)
length	Length of data to be sent with echo request
delay	Delay in ticks between each packet. Minimum = 2 ticks

Description

This function sends "count" ICMP/IPv4 Echo requests to the "dest" IP address with "delay" ticks between each request. For each request, it generates "length" data bytes to be included in the packet. The data consists of a series of bytes in a pattern that increments from 0 to 255. If length is greater than 255 bytes, the pattern repeats. The function also validates that each byte in the response is received in the correct order. It does not validate the length of the response, because some sites limit the length of their responses.

Returns

This function returns immediately with `ESUCCESS` or a negative error code indicating whether or not the ping request was successfully set up. The actual requests are sent in the backgound. A message is sent to the console when each reply is received. If a console is not available, you can verify success by looking at the ICMP statistics.

Name

`start_ping6()`

Syntax

```
int start_ping6(ip6_addr *dest, uint16_t count, int length, int delay);
```

Parameters

dest	Pointer to IP6_addr structure containing the IPv6 destination address in network byte order
count	Number of echo requests to send)
length	Length of data to be sent with echo request
delay	Delay in ticks between each packet. Minimum = 2 ticks

Description

This function sends "count" ICMP/IPv6 Echo requests to the "dest" IP address with "delay" ticks between each request. For each request, it generates "length" data bytes to be included in the packet. The data consists of a series of bytes in a pattern that increments from 0 to 255. If length is greater than 255 bytes, the pattern repeats. The function also validates that each byte in the response is received in the correct order. It does not validate the length of the response, because some sites limit the length of their responses.

Returns

This function returns immediately with `ESUCCESS` or a negative error code indicating whether or not the ping request was successfully set up. The actual requests are sent in the backgound. A message is sent to the console when each reply is received. If a console is not available, you can verify success by looking at the ICMP statistics.

Name

`dnc_init()`

Syntax

```
int dnc init(void);
```

Parameters

None

Description

This function initializes the DNS client module.

Returns

This function can return any one of the following values:

- ENP_PARAM, if the configuration provided is not correct
- ENP_RESOURCE, if the DNS client couldn't allocate memory for the DNS client table.
- ESUCCESS, if the DNS client was initialized successfully

API Name

`dns_update()`

Syntax

```
int dns_update(char *soa_mname, char *hname,
               ip_addr ipaddr, unsigned long ttl, void *pio)
```

Parameters

soa_mname	domain name
hname	Name of host to be affected by the packet
ipaddress	IPv4 address to be added or deleted
long ttl	Time to live value. Zero indicates a delete.
pio	This parameter should be set to NULL

Description

Sends a DNS UPDATE packet to the authoritative name server with the specified domain name. The API can be used to add or delete IPv4 addresses for a specified host or delete the host name and all addresses from the specified domain.

Returns

- 0 if successful
- Negative `ENP` error if internal error occurs (eg timeout)
- One of the `DNSRC_` errors from network (all positive).

Name

`gethostbyname()`

Description

This function has been deprecated in favor of `getaddrinfo( )`.

API Name

`getaddrinfo()`

Syntax

```
int getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints, struct addrinfo **res);
```

Parameters

nodename	Domain name or an IP address
servname	Service name or port number
hints	Structure defined in RFC 3943
res	Pointer to an array of one or more addrinfo structures.

Description

Translates a host name and/or a service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service. This API is defined by RFC 3493 and intended as a replacement for `gethostbyname()`. It is thread safe and very flexible. You can avoid the complexities of of the API by setting only the `nodename` parameter and leaving the last 3 parameters as NULL. Used in this manner, the API is almost as simple as `gethostbyname()`.

The "hints" parameter is an `addrinfo` structure as defined in `embdns.h`. On entry it contains a flags field, "`ai_flags`". The value in `ai_flags` is a hexadecimal `OR` of the desired "AI_" flags (`embdns.h`). The flags direct the operation of the command and may limit the returned information.

The port number returned for a specified service name is based on the `servtoportlist[ ]` array in `tcpdata.c`. Additional entries should be added as needed for a specific implementation.

The `getaddrinfo()` function returns a pointer to an array of `addrinfo` structures with one structure for each address returned. The application is responsible for calling `freeaddrinfo()` to free the array of structures.

Note

The `AI_V4MAPPED` flag is not currently supported, and the command does not currently support IPv6 scope IDs other than one.

Returns

0 or one of the `EAI` error code defined in RFC 3493 and `embdns.h`

API Name

`getnameinfo()`

Syntax

```
int getaddrinfo(const struct sockaddr *sa, int salen, char *node, int nodelen, char *service, int servicelen, int flags);
```

Parameters

sa	NULL or pointer to a socket address structure to be translated.
salen	Size of the socket address structure pointed to by sa.
node	NULL or pointer to a buffer that receives the node name.
nodelen	Length of node buffer.
node	NULL or pointer to a buffer that receives the service name.
nodelen	Length of service buffer.
flags	zero or the bitwise OR of one or more of the "NI_xxx" flags defined in embdns.h

Description

Used to translate the contents of a socket address structure to a node name and/or service name. This API is defined by RFC 3493. It is thread safe and very flexible. The complexities of the API can be avoided by setting only the `sa` and `salen` parameters and leaving the remaining parameters as NULL. When used in this this manner it becomes a relatively simple reverse lookup API.

If a buffer is provided for the service and the NI_NUMERICSERV flag is not set, the API translates the port number in the sa structure to a service name based on the `servtoportlist[ ]` array in `tcpdata.c`. Additional entries should be added to that array as needed for a specific implementation.

The value in `flags` field is the hexadecimal `OR` of the desired "NI_" flags as defined in embdns.h and RFC 3493. The hexidecimal number may be optionally proceded by a "0x" and it may optionally have a leading zero, e.g., 0x03, 0x3, or 3 are all valid

Returns

0 or one of the `EAI` error code defined in RFC 3493 and `embdns.h`

API Name

`freeaddrinfo()`

Syntax

```
void freeaddrinfo(struct addrinfo *ai);
```

Parameters

ai	Ptr to array of addrinfo structures returned by getaddrinfo()

Description

Frees the array of `addrinfo` structures returned by `getaddrinfo()`. It also frees the buffers within the structures that were used to hold names and addresses.

Returns

Nothing

Name

`gai_strerror()`

Syntax

```
CONST char *gai_strerror(int err);
```

Parameters

err	error code returned by the getaddrinfo API.

Description

Returns a text string (a single word) that represents the error code value that was returned by getaddrinfo() API. See RFC 3493 for a detailed description of the possible error values.

Returns

Returns a string for the error code or NULL if the error was not one of the defined error codes for getaddrinfo.

Name

`if_freenameindex()`

Syntax

```
void if_freenameindex(struct if_nameindex *ptr);
```

Parameters

ptr	Pointer to an array of struct if_nameindex structures

Description

This function frees a previously allocated array of struct if_nameindex structures (that was created via a call to if_nameindex()).

Returns

Nothing

Name

`if_indextoname()`

Syntax

```
char *if_indextoname(unsigned int ifindex, char *ifname);
```

Parameters

ifindex	Interface index (ones-based)
ifname	Pointer to storage for the name of interface (must be at least IF_NAMELEN bytes long)

Description

This function converts a ones-based interface index into the corresponding interface name.

Returns

This function returns its second parameter ('ifname'). It returns NULL in the event of an error.

Name

`if_nameindex()`

Syntax

```
struct if_nameindex *if_nameindex(void);
```

Parameters

None

Description

This function creates and returns a dynamically allocated array of struct if_nameindex structures representing the various interfaces present in the system.

Returns

Pointer to the first element in the dynamically allocated array of struct if_nameindex structures.

Name

`if_nametoindex`

Syntax

```
unsigned int if_nametoindex(const char *ifname);
```

Parameters

ifname	Name of interface

Description

This function converts an interface name (e.g., "et1") into the corresponding ones-based interface index.

Returns

This function returns the ones-based interface index. If the interface does not exist, it returns 0.

Name

`in_addrconf_acquire()`

Syntax

```
int in_addrconf_acquire(char *name, uint32_t flags);
```

Parameters

name	Name of interface
flags	Bitmask indicating protocols that are utilized in the address acquisition process (NETF_DHCPC (DHCP) and/or NETF_AUTOIP (auto-configuration))

Description

This function starts the address acquisition process for the specified interface using the protocols specified in the 'flags' parameter.

Returns

This function returns ESUCCESS if the address acquisition process was initiated successfully; otherwise, it returns EFAILURE.

Name

`in_addrconf_release()`

Syntax

```
int in_addrconf_release(char *name, uint32_t flags);
```

Parameters

name	Name of interface
flags	Bitmask indicating protocols that will no longer be utilized in the address acquisition process (NETF_DHCPC and/or NETF_AUTOIP)

Description

This function terminates the use of the specified protocols for address acquisition on the specified interface. The interface address configuration data structures may also be cleared out.

Returns

This function returns ESUCCESS if the address release process was initiated successfully; otherwise, it returns EFAILURE.

Name

`in_clrstats()`

Syntax

```
int in_clrstats(int module_id, void *parm);
```

Parameters

modid	Module identifier
parm	Optional parameter (currently only required to specify the interface identifier when clearing interface-specific statistics, and can be specified as NULL for all other cases)

Description

This function clears the statistics data structure for the specified module.

Returns

This function returns ENP_PARAM if the module identifier is not valid, or if the parameter pointer is NULL when clearing any interface-specific statistics for the ICMPv6, IPv6, and interface modules. It returns ENP_LOGIC if the caller requests clearing of queue- or socket-related statistics. Otherwise, it returns ESUCCESS.

Notes

- See function `in_getstats()` for discussion of table and parameter identifiers and example.

Name

`in_clrtab()`

Syntax

```
int in_clrtab(int modid, int tabid);
```

Parameters

modid	Module identifier
tabid	Table identifier

Description

This function is used to clear the ARP cache or the DNS client table in the TCP/IP stack.

Returns

This function returns ENP_PARAM if the module identifier or table identifier is not valid. Otherwise, it returns ESUCCESS.

Notes

- See function `in_getstats()` for discussion of table and parameter identifiers and example.

Name

`in_getparm()`

Syntax

```
int in_getparm(int modid, int parmid, void *valp);
```

Parameters

modid	Module identifier
parmid	Parameter identifier
valp	Pointer to storage for parameter being retrieved

Description

This function retrieves the value of the specified TCP/IP stack configuration parameter into the storage provided via 'valp'.

Returns

This function returns ENP_PARAM if any of the input parameters fails validation. Otherwise, it returns ESUCCESS.

Notes

- See function `in_getstats()` for discussion of table and parameter identifiers and example.

Name

`in_getstats()`

Syntax

```
int in_getstats(int modid, void *statp);
```

Parameters

modid	Module identifier
statp	Pointer to storage for the statistics data structure that is being retrieved

Description

This function retrieves the statistics data structure for the specified module.

Example:

```
int
ut_arpstats(void)
{
   struct arp_stats arpstats;
   int rc;

   rc = in_getstats(ARP_MODULE, &arpstats);
   printf("ut_arpstats: in_getstats() returned %d\n", rc);

   if (rc == ESUCCESS) {
      printf("arpReqsIn:           %lu\n", arpstats.arpReqsIn);
      printf("arpReqsOut:          %lu\n", arpstats.arpReqsOut);
      printf("arpRepsIn:           %lu\n", arpstats.arpRepsIn);
      printf("arpRepsOut:          %lu\n", arpstats.arpRepsOut);
      printf("arpGratReqsConflict: %lu\n", arpstats.arpGratReqsConflict);
      printf("arpGratRepsConflict: %lu\n", arpstats.arpGratRepsConflict);
   }

   return (rc);
}
```

Notes

- Each module in the system has an identifier associated with it. The complete list of module identifiers is available in stkdata.h.
- Each configurable parameter in the system has an identifier associated with it. The complete list of parameter identifiers is available in stkdata.h.
- Two tables (ARP cache and DNS client) in the system have table identifiers associated with each one of them, and can be cleared. The complete list of table identifiers is available in stkdata.h.

Returns

This function returns ENP_PARAM if the module identifier is not valid, or if the pointer to the statistics data structure is NULL, or if the socket type field is invalid when the caller requests socket-related statistics. Otherwise, it always returns ESUCCESS.

Name

`in_setparm()`

Syntax

```
int in_setparm(int modid, int parmid, void *valp);
```

Parameters

modid	Module identifier
parmid	Parameter identifier
valp	Pointer to storage for parameter being configured

Description

This function sets the value of the specified TCP/IP stack configuration parameter from data provided via 'valp'.

Returns

This function returns ENP_PARAM if any of the input parameters fails validation. Otherwise, it returns ESUCCESS.

Notes

- See function `in_getstats()` for discussion of table and parameter identifiers and example.

Name

`inet_ntop()`

Syntax

```
const char *inet_ntop(int af, const void *addr, char *str, size_t size);
```

Parameters

af	Address family (AF_INET or AF_INET6)
addr	Pointer to storage for IPv4 address ('ip_addr') or IPv6 address ('struct in6_addr') in network byte order
str	Pointer to storage for string that will contain IPv4 address in dotted decimal notation, or an IPv6 address in colon-separated notation (with the scope identifier specified as '%N', where N is the ones-based interface identifier) (e.g., FE80::211:11FF:FEBE:7F62%1)
size	Length of output buffer ('str')

Description

This functions converts a binary representation of an IPv4 address or IPv6 address (in network byte order) into a string in dotted decimal notation. The output buffer must be at least 16 (or 40) bytes long for an IPv4 (or IPv6) address.

Returns

This function returns NULL if it encountered an error; otherwise, it returns the third argument ('str').

Name

`inet_pton()`

Syntax

```
int inet_pton(int af, const char *src, void *dst);
```

Parameters

af	Address family (AF_INET or AF_INET6)
src	Pointer to string containing IPv4 address in dotted decimal notation, or an IPv6 address in colon-separated notation
dst	Pointer to storage for IPv4 address ('ip_addr') or IPv6 address ('struct in6_addr') where the results of the conversion will be stored (in network byte order)

Description

This functions converts a string containing an IPv4 or IPv6 address in printable format into its equivalent binary representation (in network byte order).

Returns

This function returns 0 if the conversion was successful. A non-zero return value indicates a failure.

Quick List for Sockets Prototypes and Details

```
extern long t_socket (int, int, int);
extern int t_bind (long, struct sockaddr *, int);
extern int t_listen (long, int);
extern long t_accept (long, struct sockaddr *, int *);
extern int t_connect (long, struct sockaddr *, int);
extern int t_getpeername (long, struct sockaddr *, int *);
extern int t_getsockname (long, struct sockaddr *, int *);
extern int t_setsockopt (long, int, int, void *, int);
extern int t_getsockopt (long, int, int, void *, int);
extern int t_recv (long, char *, int, int);
extern int t_recvfrom (long s, char * buf, int len, int flags, struct sockaddr *, int *);
extern int t_send (long, char *, int, int);
extern int t_sendto (long s, char * buf, int len, int flags, struct sockaddr *, int);
extern int t_shutdown (long, int);
extern int t_socketclose (long);
extern int t_errno (long s);
extern int t_select(fd_set * in, fd_set * out, fd_set * ev, long tmo_seconds);
```

Sockets API

Overview

This section is documentation for the embTCP Sockets layer. Sockets is an API, primarily used today for TCP programming, which was developed during the early 1980s at U.C. Berkeley for UNIX. Dozens of books and tutorials are available for Sockets programming (one of the compelling arguments for their use), so this section is devoted to functional descriptions of the Sockets subset as supported by embTCP. It is not a tutorial.

The calls documented in this section are compatible with those on UNIX systems insofar as TCP use goes. Example networking code from other Sockets-based systems should work here, and most of what is in the books and tutorials apply as well. We've tried to update the man-pages herein to reflect any differences there are.

One general difference is that all the function names in the embTCP package start with "`t_`", e.g. `socket()` is `t_socket()`. This is so that embedded systems which already use some of the socket names will not have a conflict at link time.

Another is the UNIX `errno` mechanism has been replaced by an error holder attached to each socket structure and assigned a value whenever an error occurs. Thus when a socket call indicates failure, such as `t_recv()` returning `-1`, you can examine this member or call `t_errno(long s)` to find out what went wrong. Possible values for Sockets errors are listed below. These are a subset of the standard Berkeley errors.

Sockets Errors

```
/* BSD Sockets errors */
#define    ENOBUFS         1
#define    ETIMEDOUT       2
#define    EISCONN         3
#define    EOPNOTSUPP      4
#define    ECONNABORTED    5
#define    EWOULDBLOCK     6
#define    ECONNREFUSED    7
#define    ECONNRESET      8
#define    ENOTCONN        9
#define    EALREADY        10
#define    EINVAL          11
#define    EMSGSIZE        12
#define    EPIPE           13
#define    EDESTADDRREQ    14
#define    ESHUTDOWN       15
#define    ENOPROTOOPT     16
#define    EHAVEOOB        17
#define    ENOMEM          18
#define    EADDRNOTAVAIL   19
#define    EADDRINUSE      20
#define    EAFNOSUPPORT    21
#define    EINPROGRESS     22
#define    ELOWER          23
#define    ENOTSOCK        24
#define    EIEIO           27
#define    ETOOMANYREFS    28
#define    EFAULT          29
#define    ENETUNREACH     30
```

Sockets API Calls Reference

Name

`t_socket()`

Syntax

```
long t_socket (int domain, int type, int protocol);
```

Parameters

domain	Communication domain (AF_INET or AF_INET6)
type	Socket type (SOCK_STREAM (TCP) or SOCK_DGRAM (UDP))
protocol	0

Description

`t_socket()` creates an endpoint for communication and returns a descriptor. The `domain` parameter specifies a communications domain within which communication will take place; this selects the `protocol` family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file `embtcp.h`.

A `SOCK_STREAM` type provides sequenced, reliable, two-way connection based byte streams. A `SOCK_DGRAM` socket provides connectinless, unreliable data transfer to an application.

Sockets of type `SOCK_STREAM` are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a `t_connect()` call. Once connected, data may be transferred using `t_send()` and `t_recv()` calls. When a session has been completed, a

t_socketclose() may be performed. Out-of-band data may also be transmitted as described in the t_send() page and received as described in t_recv().

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the return code from t_errno(). An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (such as five minutes).

SOCK_DGRAM sockets allow sending of datagrams to correspondents named in t_sendto() calls. Datagrams are generally received with t_recvfrom(), which returns the next datagram with its return address.

The operation of sockets is controlled by socket level options. These options are defined in the file embtcp.h. t_getsockopt() and t_setsockopt() are used to get and set options, respectively.

Please note that (AF_INET6, SOCK_STREAM) and (AF_INET6, SOCK_DGRAM) sockets can only be created with the embDUAL library.

Returns

A return value of -1 from t_socket() indicates failure. Any other return value indicates success.

See Also

t_accept, t_bind, t_connect, t_getsockname, t_getsockopt, t_listen, t_recv, t_select, t_send, t_shutdown

---

Name

t_listen()

Syntax

```
int t_listen(long s, int backlog);
```

Parameters

s	Socket identifier
backlog	Used to compute a limit on the maximum number of connections that can be pending in the completed (those for which the TCP three-way handshake has completed) and partially completed (those for which the TCP three-way handshake has started, but isn't complete) queues.

Description

To accept connections, a socket is first created with t_socket(), a backlog for incoming connections is specified with t_listen() and then the connections are accepted with t_accept(). The t_listen() call applies only to sockets of type SOCK_STREAM. The backlog parameter defines the maximum length for the queue of pending connections (not maximum open connections).

The computation is:

```
val = MIN(backlog, 5);
limit  =  3 * val / 2;
```

If a connection request arrives with the queue full the client will receive an error with an indication of ECONNREFUSED.

All incoming connection requests (such as those from a HTTP client (Web browser)) start off in the incomplete connection queue (when a TCP segment containing the SYN is received), and move to the completed connection queue after the successful completion of the three-way handshake. When the server application executes t_accept(), it will pick up the first available connection from the completed connection queue.

Returns

Returns 0 on success. On failure, it returns -1 and sets an internal t_errno to one of the errors listed in Sockets Errors to indicate the error. The t_errno can be retrieved by a call to t_errno(s).

See Also

t_accept, t_connect

Name

`t_connect()`

Syntax

```
int t_connect(long s, struct sockaddr *name, int namelen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in (or struct sockaddr_in6) structure containing addressing information for remote end (peer)
namelen	Length of sockaddr_in (or struct sockaddr_in6) structure (bytes)

Description

The parameter `s` is a socket. If it is of type `SOCK_DGRAM`, then this call specifies the peer with which the socket is to be associated; the address to which datagrams are sent and the only address from which datagrams are received. If it is of type `SOCK_STREAM`, then this call attempts to make a connection to another socket. The other socket is specified by `name` which is an address in the communications space of the socket. Each communications space interprets the `name` parameter in its own way.

Datagrams may use `t_connect()` multiple times to change their association. Datagram may also dissolve the association by connecting to an invalid address, such as a zero address.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_accept, t_connect, t_getsockname, t_select, t_socket

Name

`t_socketclose()`

Syntax

```
int t_socketclose(long s);
```

Note: this is just `close()` on traditional Sockets systems.

Parameters

s	Socket identifier

Description

The `t_socketclose()` call causes all of a full-duplex connection on the socket associated with `s` to be shut down. Once a socket is closed, no further socket calls should be made with it.

Returns

This returns `0` on success. On failure, it returns `-1`.

See Also

t_accept, t_socket

Name

`t_errno ()`

Syntax

```
int t_errno(long s);
```

Parameters

s	Socket identifier

Description

This function returns the error associated with the specified socket.

Returns

This function returns ENOTSOCK if the socket identifier is not valid. Otherwise, it returns the error associated with the specified socket.

Name

`t_select()`

Syntax

```
int t_select (fd_set * readfds, fd_set * writefds, fd_set * exceptfds, long tv);

void FD_SET (long so, fd_set * set)

void FD_CLR (long so, fd_set * set)

void FD_ISSET (long so, fd_set * set)

void FD_ZERO (fd_set * set)
```

Parameters

s	Socket identifier
readfds	Set of descriptors that an application will wait to become ready for reading
writefds	Set of descriptors that an application will wait to become ready for writing
exceptfds	Set of descriptors that an application will wait for occurence of an exceptional condition on
tv	Timeout duration (system ticks)

Description

`t_select()` examines the socket descriptor sets whose addresses are passed in `readfds`, `writefds`, and `exceptfds` to see if some of their descriptors are ready for reading, ready for writing or have an exception condition pending. On return, `t_select()` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned. Any of `readfds`, `writefds`, and `exceptfds` may be given as `NULL` pointers if no descriptors are of interest. Selecting true for reading on a socket descriptor upon which a `t_listen()` call has been performed indicates that a subsequent `t_accept()` call on that descriptor will not block.

In the standard Berkeley UNIX Sockets API, the descriptor sets are stored as bit fields in arrays of integers. This works in the UNIX environment because under UNIX socket descriptors are file system descriptors which are guaranteed to be small integers that can be used as indexes into the bit fields. In the embTCP stack, socket descriptor are pointers and thus a bit field representation of the descriptor sets is not feasible. Because of this, the embTCP Sockets API differs from the Berkeley standard in that the descriptor sets are represented as instances of the following structure:

```
typedef struct fd_set {      /* the select socket array manager */
   unsigned fd_count;       /* how many are SET? */
   long fd_array[FD_SETSIZE];   /* an array of SOCKETs */
} fd_set;
```

Instead of a socket descriptor being represented in a descriptor set via an indexed bit, an embTCP socket descriptor is represented in a descriptor set by its presence in the `fd_array` field of the associated `fd_set` structure. Despite this non-standard representation of the descriptor sets themselves, the following standard entry points are provided for manipulating such descriptor sets: `FD_ZERO(&fdset)` initializes a descriptor set `fdset` to the null set. `FD_SET(fd, &fdset)` includes a particular descriptor, `fd`, in `fdset`. `FD_CLR(fd, &fdset)` removes `fd` from `fdset`. `FD_ISSET(fd, &fdset)` is nonzero if `fd` is a member of `fdset`, zero otherwise. These entry points behave according to the standard Berkeley semantics.

The porting engineer should be aware that the value of `FD_SETSIZE` defines the maximum number of descriptors that can be represented in a single descriptor set. The default value of `FD_SETSIZE` of 12 is defined in `emb_h/embtcp.h`.

Another difference between the Berkeley and embTCP `t_select()` calls is the representation of the timeout parameter. Under Berkeley Sockets, the timeout parameter is represented by a pointer to a structure. Under embTCP Sockets, a timeout value is specified by the `tv parameter`, which defines the maximum number of system ticks that should elapse before the call to

t_select() returns. A tv parameter value of 0 implies that t_select() should return immediately (effectively a poll of the sockets in the descriptor set). The value INFINITE_DELAY is used to specify that t_select() block forever unless one of its descriptors becomes ready. The longest finite delay is 0x7FFFFFFE system ticks.

The final difference between the Berkeley and embTCP versions of t_select() is the absence in the embTCP version of the Berkeley width parameter. The width parameter is of use only when descriptor sets are represented as bit arrays and was thus deleted in the embTCP implementation.

Returns

t_select() returns a non-negative value on success. A positive value indicates the number of ready descriptors in the descriptor sets. 0 indicates that the time limit specified by tv expired.

See Also

t_accept, t_connect, t_listen, t_recv, t_send

Notes

Under rare circumstances, t_select() may indicate that a descriptor is ready for writing when in fact an attempt to write would block. This can happen if system resources necessary for a write are exhausted or otherwise unavailable. If an application deems it critical that writes to a file descriptor not block, it should set the descriptor for non-blocking I/O. See discussion of t_setsockopt.

Name

t_recv()

t_recvfrom()

Syntax

```
int t_recv(long s, char * buf, int len, int flags);

int t_recvfrom(long s, char *buf, int len, int flags, struct sockaddr *from, int *fromlen);
```

Parameters

s	Socket identifier
buf	Start address of buffer where received data will be copied into
len	Length of data to be sent
flags	Flags for receiving process (e.g., MSG_PEEK)
from	Pointer to struct sockaddr_in (or struct sockaddr_in6) structure that will be used to store addressing information for the remote end
fromlen	Pointer to storage for length of sockaddr_in (or struct sockaddr_in6) structure (bytes)

Description

s is a socket created with t_socket(). t_recv() and t_recvfrom() are used to receive messages from another socket. t_recv() may be used only on a connected socket (see t_connect), while t_recvfrom() may be used to receive data on a socket whether it is in a connected state or not.

If from is not a NULL pointer, the source address of the message is filled in. fromlen is a value-result parameter, initialized to the size of the buffer associated with from, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see t_socket).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see t_setsockopt) in which case -1 is returned with the external variable t_errno set to EWOULDBLOCK.

Note that t_recv() will return an EPIPE if an attempt is made to read from an unconnected socket.

The t_select() call may be used to determine when more data arrive.

The flags parameter is formed by OR-ing one or more of the following:

MSG_OOB	Read any "out-of-band" data present on the socket, rather than the regular "in-band" data.
MSG_PEEK	"Peek" at the data present on the socket; the data are returned, but not consumed, so that a subsequent receive operation will see the same data.

Returns

These calls return the number of bytes received, or `-1` if an error occurred. On failure, they set an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_connect,t_getsockopt,t_select, t_send, t_socket

Name

`t_send()`

`t_sendto()`

Syntax

```
int t_send(long s, char *buf, int len, int flags);

int t_sendto(long s, char *buf, int len, int flags, struct sockaddr *to, int tolen);
```

Parameters

s	Socket identifier
buf	Start address of data to be sent
len	Length of data to be sent
flags	Flags for sending process (e.g., MSG_OOB)
to	Pointer to struct sockaddr_in (or struct sockaddr_in6) structure containing addressing information for the remote end
tolen	Length of sockaddr_in (or struct sockaddr_in6) structure (bytes)

Description

`t_send()` and `t_sendto()` are used to transmit the message addressed by buf to another socket. `t_send()` may be used only when the socket is in a connected state, while `t_sendto()` may be used at any time, in which case the address of the target is given by the to parameter. The length of the message is given by `len`.

No indication of failure to deliver is implicit in a `t_send()`. Locally detected errors are indicated by a return value of `-1`.

If no messages space is available at the socket to hold the message to be transmitted, then `t_send()` normally blocks, unless the socket has been placed in non-blocking I/O mode. The `t_select()` call may be used to determine when it is possible to send more data.

The `flags` parameter may include one or more of the following:

```
#define MSG_OOB         0x1   /* process out-of-band data */
```

The flag `MSG_OOB` is used to send "out-of-band" data on sockets that support this notion (e.g. `SOCK_STREAM`); the underlying protocol must also support "out-of-band" data.

Returns

The call returns the number of characters sent, or `-1` if an error occurred. On failure, it sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_recv, t_select, t_getsockopt, t_socket

Name

`t_accept()`

Syntax

```
long t_accept(long s, struct sockaddr *addr, int *addrlen);
```

Parameters

s	Socket identifier
addr	Pointer to struct sockaddr_in (or struct sockaddr_in6) structure that will be used to store addressing information for the remote end in newly accepted connection
addrlen	Pointer to storage for length of struct sockaddr_in (or struct sockaddr_in6) structure (bytes)

Description

The argument `s` is a socket that has been created with `t_socket()`, bound to an address with `t_bind()` and is listening for connections after a `t_listen()`. `t_accept()` extracts the first connection on the queue of pending connections, creates a new socket with the same properties as `s` and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `t_accept()` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `t_accept()` returns an error as described below. The accepted socket is used to read and write data to and from the socket which connected to this one; it is not used to accept more connections. The original socket s remains open for accepting further connections.

The argument `addr` is a result parameter that is filled in with the address of the connecting entity as known to the communications layer, i.e. the exact format of the `addr` parameter is determined by the domain in which the communication is occurring. The `addrlen` is a value-result parameter. It should initially contain the amount of space pointed to by `addr`. On return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

It is possible to `t_select()` a socket for the purposes of doing an `t_accept()` by selecting it for read.

Returns

`t_accept()` returns a non-negative descriptor for the accepted socket on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_bind, t_connect, t_listen, t_select, t_socket

Name

`t_bind()`

Syntax

```
int t_bind(long s, struct sockaddr *name, int namelen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in (or struct sockaddr_in6) structure containing addressing information for local end
namelen	Length of struct sockaddr_in (or struct sockaddr_in6) structure (bytes)

Description

`t_bind()` sets the local endpoint address and port number for a socket. When a socket is created with `t_socket()` it exists in a name space (address family) but has no name assigned. `t_bind()` requests that the name pointed to by name be assigned to the socket.

Returns

`t_bind()` returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_connect, t_getsockname, t_listen, t_socket

Name

`t_shutdown()`

Syntax

```
int t_shutdown(long s, int how);
```

Parameters

s	Socket identifier
how	Type of shutdown (SHUT_RD, SHUT_WR, or SHUT_RDWR)

Description

The `t_shutdown()` call causes all or part of a full-duplex connection on the socket associated with `s` to be shut down. If `how` is `SHUT_RD`, then further receives will be disallowed. If how is `SHUT_WR`, then further sends will be disallowed. If how is `SHUT_RDWR`, then further sends and receives will be disallowed.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_connect, t_socket

Name

`t_getpeername()`

Syntax

```
int t_getpeername(long s, struct sockaddr *name, int * addrlen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in (or struct sockaddr_in6) structure that will be used to store addressing information for remote end
addrlen	Pointer to storage for length of struct sockaddr_in (or struct sockaddr_in6) structure (bytes)

Description

Fills in the passed `struct sockaddr` with the IP addressing information of the connected host.

The `addrlen` is a value-result parameter and should initially contain the amount of space pointed to by `name`.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_bind, t_socket

Name

`t_getsockname()`

Syntax

```
int t_getsockname(long s, struct sockaddr *name, int * addrlen);
```

Parameters

s	Socket identifier
name	Pointer to struct sockaddr_in (or struct sockaddr_in6) structure that will be used to store addressing information for local end
addrlen	Pointer to storage for length of struct sockaddr_in (or struct sockaddr_in6) structure (bytes)

Description

`t_getsockname()` returns the current name for the specified socket, in the passed `struct sockaddr`.

The `addrlen` is a value-result parameter and should initially contain the amount of space pointed to by `name`.

Returns

This returns `0` on success. On failure, it returns `-1` and sets an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_bind, t_getpeername, t_socket

Name

`t_getsockopt()`

`t_setsockopt()`

Syntax

```
int t_getsockopt(long s, int level, int optname, char *optval, int optlen);

int t_setsockopt(long s, int level, int optname, char *optval, int optlen);
```

Parameters

s	Socket identifier
level	Level of socket option (IP_OPTIONS or SOL_SOCKET)
optname	Name of socket option (e.g., SO_ERROR)
optval	Pointer to storage for socket option (for reading (get) or writing (set))
optlen	Size of storage pointed to by 'optval'

Description

`t_getsockopt()` and `t_setsockopt()` manipulate options associated with a socket. The `optname` parameter identifies an option that is to be set with `t_setsockopt()` or retrieved with `t_getsockopt()`.

The parameter `optval` is used to specify option values for `t_setsockopt()`. On calls to `t_setsockopt()` it generally contains a pointer to a variable or structure, the contents of which will define the value of the option to be set. On calls to `t_getsockopt()` it generally points to a variable or structure into which the value for the requested option is to be returned.

The following options are recognized by embTCP:

Option Name	Description
IPV6_JOIN_GROUP	Join a multicast IPv6 group on the specified (ones-based) interface
IPV6_LEAVE_GROUP	Leave a multicast IPv6 group on the specified (ones-based) interface
IPV6_MULTICAST_HOPS	Specify the Hop Limit value for use in IPv6 header of outgoing UDP/IPv6 multicast datagram
IPV6_MULTICAST_IF	Specify (ones-based) egress interface for outgoing UDP/IPv6 multicast datagrams
IPV6_MULTICAST_LOOP	Enable or disable loopback of outgoing UDP/IPv6 multicast datagrams
IPV6_UNICAST_HOPS	Value of Hop Limit field in IPv6 header of outgoing unicast packets
IP_ADD_MEMBERSHIP	Join the socket to the supplied UDP/IPv4 multicast group on the specified interface.

IP_DROP_MEMBERSHIP	Leaves the specified UDP/IPv4 multicast group from the specified interface.
IP_MULTICAST_IF	Specify egress interface for outgoing UDP/IPv4 multicast datagrams
IP_MULTICAST_LOOP	Enable or disable loopback of outgoing UDP/IPv4 multicast datagram
IP_MULTICAST_TTL	Specify the Time to Live value for use in IPv4 header of outgoing UDP/IPv4 multicast datagram
IP_SCOPEID	Scope identifier for outgoing IPv6 datagrams (to identify outbound interface)
IP_TOS	set TOS field in IPv4 header for outgoing IPv4 datagrams
IP_TTL_OPT	set TTL field in IPv4 header for outgoing IPv4 datagrams
SO_BIO	configure socket to use blocking I/O
SO_ERROR	retrieve and clear last error on the socket
SO_KEEPALIVE	enable or disable TCP keepalives
SO_LINGER	enable or disable linger on close if data present
SO_NBIO	configure socket to use non-blocking I/O
SO_NONBLOCK	configure socket to use blocking or non-blocking I/O
SO_OOBINLINE	enable or disable inline reception of TCP out-of-band data
SO_RCVBUF	maximum amount of data that can be queued in socket's receive buffer
SO_REUSEADDR	enable or disable local address reuse
SO_RXDATA	number of bytes queued in socket's receive buffer
SO_SNDBUF	maximum amount of data that can be queued in socket's send buffer
SO_TXDATA	number of bytes queued in socket's send buffer
SO_TYPE	retrieve the type of the socket (SOCK_STREAM or SOCK_DGRAM)
TCP_ACKDELAYTIME	Specify delayed ACK time (in units of milliseconds)
TCP_NODELAY	enable or disable Nagle algorithm

embTCP supports get/set operations for socket options as follows:

Option Name	level	type	supported operation
IPV6_JOIN_GROUP	IPPROTO_IP	struct ipv6_mreq	set
IPV6_LEAVE_GROUP	IPPROTO_IP	struct ipv6_mreq	set
IPV6_MULTICAST_HOPS	IPPROTO_IP	int	get, set
IPV6_MULTICAST_IF	IPPROTO_IP	unsigned int	get, set
IPV6_MULTICAST_LOOP	IPPROTO_IP	unsigned int	get, set
IPV6_UNICAST_HOPS	IP_OPTIONS	unsigned int	set
IP_ADD_MEMBERSHIP	IPPROTO_IP	struct ip_mreq	set
IP_DROP_MEMBERSHIP	IPPROTO_IP	struct ip_mreq	set
IP_MULTICAST_IF	IPPROTO_IP	ip_addr	get, set
IP_MULTICAST_LOOP	IPPROTO_IP	u_char	get, set
IP_MULTICAST_TTL	IPPROTO_IP	u_char	get, set
IP_SCOPEID	IP_OPTIONS	unsigned int	set
IP_TOS	IP_OPTIONS	int	get, set
IP_TTL_OPT	IP_OPTIONS	int	get, set
SO_BIO	SOL_SOCKET	<none>	set
SO_ERROR	SOL_SOCKET	int	get
SO_KEEPALIVE	SOL_SOCKET	int	get, set
SO_LINGER	SOL_SOCKET	struct linger	get, set

SO_NBIO	SOL_SOCKET	<none>	set
SO_NONBLOCK	SOL_SOCKET	int	get, set
SO_OOBINLINE	SOL_SOCKET	int	get, set
SO_RCVBUF	SOL_SOCKET	int	get, set
SO_REUSEADDR	SOL_SOCKET	int	get, set
SO_RXDATA	SOL_SOCKET	int	get
SO_SNDBUF	SOL_SOCKET	int	get, set
SO_TXDATA	SOL_SOCKET	int	get
SO_TYPE	SOL_SOCKET	int	get
TCP_ACKDELAYTIME	SOL_SOCKET	int	get, set
TCP_NODELAY	SOL_SOCKET	int	get, set

The include file `embtcp.h` contains definitions for option names, described below. Most options take a pointer to an int variable for `optval`. For `t_setsockopt()`, the variable addressed by the parameter should be non-zero to enable a Boolean option or zero if the option is to be disabled.

`SO_LINGER` uses a `struct linger` parameter defined in `embtcp.h`. This parameter specifies the desired state of the option and the linger interval (see below).

`SO_REUSEADDR` indicates that the rules used in validating addresses supplied in a `t_bind()` call should allow reuse of local addresses.

`SO_KEEPALIVE` enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken. If the process is waiting in `t_select()` when the connection is broken, `t_select()` returns true for any read or write events selected for the socket.

`SO_LINGER` controls the action taken when unsent messages are queued on socket and a `t_socketclose()` is performed. If the socket promises reliable delivery of data and `SO_LINGER` is set, the system will block the caller on the `t_socketclose()` attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the `linger interval`, is specified in the `t_setsockopt()` call when `SO_LINGER` is requested). If `SO_LINGER` is disabled and a `t_socketclose()` is issued, the system will process the close in a manner that allows the caller to continue as quickly as possible.

With protocols that support out-of-band data, the `SO_OOBINLINE` option requests that out-of-band data be placed in the normal data input queue as received. It will then be accessible with `t_recv()` calls without the `MSG_OOB` flag.

`SO_SNDBUF` and `SO_RCVBUF` are options to adjust the normal buffer sizes allocated for the output and input buffers respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

`SO_TYPE` and `SO_ERROR` are options used only with `t_getsockopt()`. `SO_TYPE` returns the type of the socket, for example `SOCK_STREAM`. `SO_ERROR` returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

The `TCP_NODELAY` disables the Nagle algorithm, and prevents attempts to coalesce small packets less than the `TCP_MSS`, while awaiting acknowledgement for data already sent.

The options `SO_NONBLOCK`, `SO_NBIO`, and `SO_BIO` are unique to the InterNiche stack (these options do not appear in the Berkeley Sockets API) and are used to control whether a socket uses blocking or non-blocking IO.

`SO_NBIO` is used to specify that a socket use non-blocking IO. `SO_BIO` is used to specify that a socket use blocking IO. The use of `t_setsockopt()` to set these options is different than that of the standard Boolean options in that the value in optval is not used. All that is necessary is to specify the appropriate option name in optname.

Returns

The value returned for the `SO_KEEPALIVE`, `SO_OOBINLINE`, and `SO_REUSEADDR` socket options in a `t_getsockopt()` call is the corresponding bitmask (e.g., `SO_KEEPALIVE (8)`, as defined in `embtcp.h`) if the option is enabled, and zero otherwise.

For all others, `0` is returned upon success. On failure, they return `-1` and set an internal `t_errno` to one of the errors listed in Sockets Errors to indicate the error. The `t_errno` can be retrieved by a call to `t_errno(s)`.

See Also

t_socket

Ethernet Device Driver API

Name

`get_mac_address()`

Syntax

```
int get_mac_address(int iface, uint8_t *emac);
```

Parameters

iface	Address of the array to contain the Ethernet address.
emac	Specifies the ethernet interface; the first interface is 0.

Description

The ethernet driver calls `get_mac_address()` to obtain the 48-bit MAC address of the hardware interface. The function obtains the 6-byte value from a user-defined location, such as non-volatile memory, and copies it into the array pointed to by the `emac` parameter.

Returns

The function returns 0 if the copy was successful, and -1 if there was a parameter error or the MAC address was unavailable.

Notes

For a chip with a single ethernet device, 0 is the only valid `iface` parameter value.

Name

`emac_phy_init()`

Syntax

```
int emac_phy_init(void);
```

Parameters

None.

Description

The ethernet driver calls `emac_phy_init()` to initialize the external PHY chip. In order to accomodate different PHY chips, the implementation of the initialization function is the responsibility of the porting engineer. A sample implementation for the DP83848C PHY chip is provided in phy_dp83848.c.

The initialization function should:

- reset the PHY chip
- configure the PHY chip
- bring up the Link layer of the network interface
- provide the link configuration information to the ethernet device driver.

If the link is successfully established, either through auto-negotiation or by programming the link speed and duplex mode, the `emac_phy_init()` function must call `eth_setlink()` to communicate the negotiated link characteristics to the ethernet device driver. Refer to the sample `emac_phy_init()` implementation for an example of the `eth_setlink()` call.

Returns

`emac_phy_init()` returns 0 if the PHY initialization was successful and a negative error code if the initialization failed. If the PHY initialization fails, the ethernet device initialization fails, and the ethernet interface is considered unavailable.

Notes

The PHY chip must be connected to the network in order to initialize the PHY using the auto-negotiation protocol.

The `usleep` function can be used to insert short delays into the initialization process. The delay is accomplished via a software loop. The loop parameters may need to be adjusted for the actual system clock speed.

CLI

tcpdata.c carries a table of CLI commands and the implementation of a demonstration command: user1_cli(). The CLI commands are described as man pages in the CLI Commands section. The implementation of user1_cli() was designed to illustrate the major aspects of the CLI Interface and its helper functions.

embTCP's Command Line Interface (CLI) provides access to InterNiche's embedded products commands and statistics function. Additional commands and menus can be added and these will be treated exactly like the pre-defined commands. Broadly speaking, the CLI mechanism parses "command line input", and uses the initial tokens to identify the desired "C" function. Any parameters that follow the initial tokens are parsed and passed to the menu routine in a standardized format.

CLI commands are parsed and processed by the CLI module. The command definitions are grouped within menus which are part of a NicheStack module. All command for the embTCP library are grouped within a single menu. However if you have purchased other InterNiche embedded libraries, they will each have separate menus.

Throughout the discussion below, the example command, "user1_cli()", located in tcpdata.c will be used to illustrate the various CLI structures and mechanisms.

Menu Structures

A menu is defined by the cli_menu structure in embcli.h

```
/* CLI menu description */
struct cli_menu {
   char  *name;           /* The name of the menu */
   int   ncmds;           /* Number of commands in the menu */
   struct cli_cmd *cmds;  /* Array of structures describing each command */
};
```

The following is the definition of the menu for the embTCP module (from tcpdata.c):

```
struct cli_menu netmain_nt = {
   "embtcp",
   sizeof(net_cmds)/sizeof(struct cli_cmd),
   &net_cmds[0],
};
```

Each command within a menu is defined by the cli_cmd structure:

```
/* CLI command element */
struct cli_cmd {
   char  *name;           /* command name */
   char  *desc;           /* command description displayed to user*/
   int   (*func)(void *);  /* pointer to the CLI command function  that
                           * handles the command
   int   nparms;          /* number of parameters */
   struct cli_parm *parms; /* pointer to the parameter array */
};
```

The following is the definition of the "user1" example menu command:

```
#if INICHE_CLICMD_USER1 == 1
   {
      "user1",                    /* Command name */
      "user-defined CLI command",  /* Command description seen by the user  */
      &user1_cli,                 /* Function that will handle the command  */
      sizeof(user1_parms)/sizeof(struct cli_parm),   /* Number of parameters  */
      &user1_parms[0],            /* Pointer to parameter array for this command */
   },
#endif
```

Note that the command is surrounded by #if INICHE_CLICMD_USER1 == 1. The definitions of the cli_cmd structures for each command in embTCP are surrounded by similar defines. This allows you to control exactly which commands will be included in the build.

Each command within a menu may take from zero to many parameters (options). Parameters are defined by the cli_parm structure:

```
/* CLI parameter definition */
struct cli_parm {
   char  option;     /* letter that identifies (selects) the option */
   char  opttype;    /* CLI data type */
};
```

Each parameter consists of a letter preceded by a hyphen and optionally followed by a value. The parameter letter and the parameter value are separated by "whitespace". The parameter letter is not case-sensitive. In order to simplify command processing for small embedded systems, multiple letter identifiers are not allowed and. options cannot be combined; that is, there must be whitespace between each option.

The type of each parameter is checked as the command is parsed. Supported parameter types are:

CLI_NONE	No parameter value. The command function can test for the presence or absence of the parameter.
CLI_INT	Signed 32-bit integer.
CLI_UINT	Unsigned 32-bit integer.
CLI_STRING	A character string delimited by "whitespace". Strings containing "whitespace" can be delimited by matching single quotes, double quotes, or parentheses: -a 'hello world!', -a "I don't know", or -a (a, b, c).
CLI_IPADDR	IPv4, IPv6 or MAC address.

The following is the list of options allowed with the "`usr1_cli()`" example in `tcpdata.c`

```
struct cli_parm user1_parms[] = {
   { 'a', CLI_IPADDR },  /* parameter of type IPv4 address */
   { 'b', CLI_NONE },    /* parameter with no <argument> */
   { 'i', CLI_INT },     /* parameter of type integer */
   { 's', CLI_STRING },  /* parameter of type string */
   { 'u', CLI_UINT },    /* parameter of type unsigned integer */
};
```

Command Line Parsing

The CLI module is responsible for parsing a command line and calling the command's execution function.

The command line parser scans the command line to find the command name. The command parser then searches the installed menus, looking for a match; names are not case-sensitive. Command names can be abbreviated to the first *'N'* characters of the name, as long as the name is still unique. Command names must be minimum of 3 characters.

If the command is found, the corresponding `cli_cmd` and `cli_parm` structures are used to parse and validate the command parameters. The parameter information is stored in the CLI Context. A parameter may appear at most once in a command line and must be of the correct type. If a parameter needs to support multiple types, for example a device name string or a device number, the `CLI_STRING` type can be used and parameter validation can be deferred to the command's execution function.

The command parser recognizes the special symbols, '?' and '-?' as "help" flags. Entering a "help" flag as part of a command line causes the command's description and syntax to be displayed on the output device. If the command was successfully parsed, the command's execution function will then be called. The function can test if "help" was entered and take appropriate action, such as displaying addition "help" text and/or not executing the command.

Note that **the command line is modified** during the command parsing process. Commands that are stored in read-only memory, they must be copied into a writable memory buffer before they can be parsed and executed.

Command Execution

The CLI command parser finds the command's "`cli_cmd`" structure in the installed menus. The structure includes a pointer to the function to be called to execute the command. The function prototype for any command's execution function is:

```
int commandname(void *ctx);
```

The macros below, defined in `embcli.h`, can be used to access the command's parameters. The `user1_cli()` function in `tcpdata.c` contains example code for the `CLI_HELP`, `CLI_DEFINED` and `CLI_VALUE` macros.

CLI_HELP(ctx)	Returns TRUE if either '?' or '-?' was entered as part of the command line. Otherwise, it returns FALSE.
CLI_COUNT(ctx)	Returns an integer value representing the number of parameters that were present in the command line. The "help" symbols are not included in the count.
CLI_DEFINED(ctx,c)	Return TRUE if the command line included a '-<c>' parameter. Otherwise, it returns FALSE.

CLI_VALUE(ctx,c)	Returns the value of the '-<c>' parameter. The value must be cast into the type of the parameter. For example,
	`if (CLI_DEFINED(ctx, 'p')`
	`  port = (uint32_t)(CLI_VALUE(ctx, 'p'));`
	`else`
	`  port = MY_DEFAULT_PORT_NUMBER;`

The type for each CLI parameter type was shown in the table in the Commands and Parameters section above.

The `CLI_DEFINED()` macro should be used to test for the presence of a parameter before using CLI_VALUE() to get the its value.

When a parameter of type `CLI_IPADDR` is parsed, the information is stored in a `cli_addr` structure defined by:

```
struct cli_addr {
   int type;            /* CLI_IPV4, CLI_IPV6, or CLI_MAC */
   uint8_t addr[16];    /* IPv4, IPv6, or MAC address (network byte order) */
   int scopeID;         /* IPv6 scopeID of address */
   int prefixLen;       /* IPv6 prefix length of the address */
}
```

The bytes of the IP address are stored in network byte-order. The "type" field of the structure can be tested to determine if an IPv4 address, IPv6 address or a MAC address was entered in the command line.

If the embedded system has a console then it can call `printf()` to output a message to the user.

When the command's execution function has completed its processing, it returns an error code indicating the success or failure of the command. This error code is returned to the caller of `cli_command()` CLI error codes are defined in `embcli.h`.

Interactive CLI Commands

help - display information about a command

Name

`help - display information about a command`

Syntax

`help [-m STRING] [-c STRING]`

Parameters

`-m`	Menu group name
`-c`	Command name

Description

This command provides helpful information about the CLI commands. If no parameters are entered, the list of available menu groups and the commands within each menu group is displayed. If a menu group is specified, information about the menu group and its commands are displayed. If a command is specified, information about the command is displayed. If the command name is a present in more than one menu group, the menu group name must also be specified to remove any ambiguity.

Notes/Status

- "`help -m foo -c bar`" is equivalent to "`foo bar ?`".
- Entering "`?`" is equivalent to entering "`help`".

arp - Display ARP table entries and statistics

---

arp

`arp - Display ARP table entries and statistics`

Syntax

`arp [-z <interface number or IPv4 address>]`

Parameters

`-z`	Delete all ARP table entries associated with the specified interface or IPv4 address.

Description

This command displays the ARP table and the ARP statistics. If '`-z STRING`' is specified, all ARP table entries associated with the specified interface or IPv4 address are deleted. The interface must be specified by its name.

debug - set the IP stack trace level

---

debug

`debug - set the IP stack trace level`

Syntax

`debug [-d | -e | -n <debug level>]`

Parameters

`-d`	disable IP stack tracing
`-e`	enable default IP stack trace levels
`-n`	specify the TCP/IP stack trace levels

Description

NicheStack includes code to trace the progress of network packets as they move through the various levels of the stack. This command controls which levels of the stack display trace information.

The '-n' parameter is a bitmask specifying which levels of the IP stack (i.e. protocol, transport, internet, application, etc.) will display trace information. If '-d' is specified, IP stack tracing is disabled (equivalent to '-n 0'). If '-e' is specified, IP stack tracing is enabled for the default IP stack trace level (equivalent to '-n 0x2314').

Notes/Status

- Only available if NPDEBUG is defined.
- The bitmask associated with each of the various trace levels are: INFOMSG (0x04), NETERR (0x08), PROTERR (0x10), TPTRACE (0x100), IPTRACE (0x200), UPCTRACE (0x400), IP6TRACE (0x2000).

dhcstat - Display DHCP/auto-configuration-related information

Name

dhcstat - Display DHCP/auto-configuration-related information

Syntax

dhcstat

Parameters

None.

Description

This command displays the status of the DHCP, auto-configuration, and UPNP state machines.

dnscstats - Display DNS client statistics.

Command Name

dnscstats - Display DNS client statistics.

Syntax

dnscstats [-c]

Parameters

-c	Display the DNS client cache

Description

Display DNS client statistics. If -c is specified it will also display the DNS client cache

Sample output when the -c options was specified:

```
DNS Servers:68.87.76.178, 65.106.1.196, 0.0.0.0
Number entries in DNS Client cache: 2
protocol/implementation runtime errors: 0
requests sent: 2
updates sent: 0
replies received: 2
usable replies: 2
total retries: 0
timeouts: 0

DNS cache:
name: www.something.com
11.22.33.44
  Age: 8 seconds, Expires: 82103 seconds
  trys: 1, ID:4661, rcode:0, err:0
name: www.something.com
  2001:1111:2222:3333::2
  Age: 84 seconds, Expires 77261 seconds
  trys: 1, ID:4660, rcode:0, err:0
```

getaddrinfo - Get list of IP addresses and/or port numbers for a hostname and service name

Command Name

`getaddrinfo - Get list of IP addresses and/or port numbers for a hostname and service name`

Syntax

`getaddrinfo [-a <host>] [-s <service>] [-f <flags>] [-p <protocol>] [-t <socktype>] [-v <version>]`

Parameters

`-a`	STRING: Either a domain name or, when used with the AI_NUMERICHOST flag, an IP address
`-s`	STRING: Either a service name or, when used with the AI_NUMERICSERV flag, a port number
`-f`	STRING: Hexadecimal string representing an OR of desired "AI_" flags (see embdns.h)
`-p`	STRING: "TCP" or "UDP". The returned port number must be valid for this protocol
`-t`	INT: 1 = SOCK_STREAM, 2 = SOCK_DGRAM. The returned port number must be valid for this socket type
`-v`	INT: 4 = IPv4, 6 = IPv6. Restrict responses to these address types

Description

This command is intended as an example/test for calls to the `getaddrinfo()` API. The `getaddrinfo()` function is defined in RFC 3493. It returns a list of IP addresses and/or port numbers for the specified hostname and/or service name. `getaddrinfo()` is a replacement for `gethostbyname()`. It is thread safe and very flexible. You can avoid the complexities of of the API by setting only the `nodename` parameter and leaving the last 3 parameters as NULL. Used in this manner, the command is almost as simple as `gethostbyname()`.

Notes/Status

- Either `-a` or `-s` or both must be specified.
- The determination of port number for the `-s` parameter is based on the `servtoportlist[]` in `tcpdata.c`. The default array is limited in size. Additional entries should be added as needed for a specific implementation.
- For the `-f` option, the hexadecimal string must be an OR of one of the `"AI_"` flags defined in `embdns.h`. The meaning of each flag is defined in RFC 3493. The hexadecimal number may be optionally proceded by a "0x" and it may optionally have a leading zero, e.g., 0x03, 0x3, or 3 are all valid
- The `-p` and `-t` parameters apply only to the service name returned. If both are used, they must correspond, e.g., an error will be returned if a protocol of TCP and a sock type of SOCK_DGRAM are specified.
- The `getaddrinfo()` function returns a pointer to an array of addrinfo structures (defined in `embdns.h`), with one structure for each address returned. Normally after a return from getaddrinfo(), the calling application would use the information in the structures as needed and then call `freeaddrinfo()` to free the array of structures. For this command, the information in the returned structures is displayed and then the array of structures is freed via `freeaddrinfo()`
- The AI_V4MAPPED flag is not supported.
- The `getaddrinfo()` command does not support IPv6 scope IDs other than one.

iface - Display network interface information

Name

`iface - Display network interface information`

Syntax

`iface [-i <interface number> [-m <MTU (IPv4)>]]`

Parameters

`-i`	Network Interface number.
`-m`	The maximum transmission unit of a Network Interface.

Description

This command displays information about the specified Network Interface. If `'-i'` is not specified, the list of available Network Interfaces is displayed.

The value of the `'-m'` is used to set the maximum transmission unit of the selected interface (for use by IPv4). Valid values are 128 to 65535. Care must be taken when setting the value to ensure that the value matches the capabilities of the underlying hardware.

Notes/Status

- The first Network Interface number is 1.

Display or configure IPv6 parameters in system

ip6cfg

```
Display or configure IPv6 parameters in system
```

Syntax

```
ip6cfg [{-i <interface id> {-a <0 | 1> | -m <MTU>}} | -n <duration>]
```

Parameters

`-a`	Disable (0) or enable (1) auto-configuration of addresses from received Router Advertisement on a particular interface
`-i`	Specify interface index (ones-based)
`-m`	Specify IPv6 MTU for a particular interface in byes (must be >=1280)
`-n`	Specify ND cache entry lifetime in STALE state (seconds)

Description

This command is used to configure or display the value of IPv6 parameters.

Notes/Status

- When no arguments are specified, this command displays the values of IPv6 parameters.

Perform IPv6 control operations

ip6ctl

```
Perform IPv6 control operations
```

Syntax

```
ip6ctl -r -n
```

Parameters

`-n`	Neighbor Discovery cache
`-r`	Perform a reset operation (on the structure specified by additional command-line parameters)

Description

This command is used to clear out the Neighbor Discovery cache, reset a particular IPv6 interface, or clear out globals for a particular IPv6 interface.

Notes/Status

- Example invocation sequences:
    1. Clear out Neighbor Discovery cache

```
ip6ctl -r -n
```

Display contents of IPv6 tables in system

ip6tbl

```
Display contents of IPv6 tables in system
```

Syntax

```
ip6tbl [-a | -n | -p | -r]
```

Parameters

`-a`	Display list of addresses associated with each interface
`-n`	Display contents of Neighbor Discovery cache
`-p`	Display list of prefixes associated with each interface
`-r`	Display contents of reassembly table

Description

This command displays the contents of the specified IPv6 tables.

Notes/Status

- When no arguments are specified, this command displays the contents of all IPv6 tables.

linkstats - Display link-specific information

Name

```
linkstats - Display link-specific information
```

Syntax

```
linkstats -i <interface number>
```

Parameters

`-i`	Network Interface number.

Description

This command displays Link information for the specified Network Interface. The Link information is device-specific, and is provided by the implementor of the Network Interface device driver.

Notes/Status

- The first Network Interface number is 1.

netstat - display operational statistics

Name

```
netstat - display operational statistics
```

Syntax

```
netstat {-c [-p <protocol>]} -m -q {-s [-p <protocol>]}
```

Parameters

`-c`	display status information about TCP or UDP sockets.
`-m`	display status information about mbufs.
`-p`	specify network protocol (ICMP ("icmp"), ICMPv6 ("icmp6"), IP ("ip"), IPv6 ("ip6"), TCP ("tcp" or "tcp6"), and UDP ("udp" or "udp6")) for which information is being requested.
`-q`	display status information about the TCP/IP stack's receive queue.
`-s`	display statistics for specified network protocol.

Description

This command displays status information for various network protocols or modules.

Notes/Status

- The protocol names can be specified in either lowercase or uppercase.
- The -p option is only intended for use with either the -c option or the -s option.
- When the -c option is used without a -p option, this command will display information about sockets for all protocols.
- When the -s option is used without a -p option, this command will display statistics for all protocols.

nslookup - Find the IP address of a domain name or the domain name for an IP address

Name

`nslookup - Find the IP address of a domain name or the domain name for an IP address`

Syntax

```
nslookup -a <name> [-r | -y]
```

Parameters

`-a`	Display the IPv4 address(es) of the domain name
`-r`	Perform a reverse lookup of the domain name
`-y`	Display the IPv6 records of the domain name

Description

This command calls DNS servers or relies on the stack's cache to retrieve the IP address records or TXT records for a domain name, such as "www.iniche.com." If no record type is specified, the `A records` (IPv4 addresses) are displayed. If `'-y'` is specified, the `AAAA records` (IPv6 addresses) are displayed. If `'-r'` is specified, a reverse nslookup of an IPv4 address is performed.

Notes/Status

- `'-r'and '-y'` are mutually exclusive.
- Reverse lookup is not supported for IPv6 addresses.
- The command depends on the DNS server addresses set in tcpdata.c or by the `setdnssrv` command.

ping - Send ICMP echo requests

Name

`ping - Send ICMP echo requests`

Syntax

```
ping (-a <IP addr> | -h <host name>) [-l <length of packets>] [-n <number of pings>] [-q] [-t <ticks between pings>] [-v <version>]

ping -k <session ID>

ping -s
```

Parameters

`-a`	IPv4 or IPv6 address
`-h`	Host name
`-l`	Length of packets. Default = 64 bytes.
`-n`	Number of pings. Default = 4.
`-q`	Enable/Disable the printing of results for each ping. Default = Enabled.
`-t`	Interval between pings in ticks. Default = 20 ticks. Minimum = 2 ticks.
`-v`	IP address type ('4' or '6') that should be requested with the `-h` option. Default = 4.

-k	Kill ping request specified by Session ID.
-s	Print cumulative ping statistics.

Description

This command sends an ICMP echo requests in either IPv4 or IPv6 format. It verifies that the data in the responses exactly matches the data in the request and reports any mismatches. Multiple ping requests can be entered while the earlier requests are in progress. The ping request will return either a negative error code or a positive session ID number. The session ID is intended for use with the -k option.

Notes/Status

- The -a and -h options are mutually exclusive.
- The -v may be used to specify whether the nslookup that occurs as a result of the -h option should request an IPv4 or an IPv6 address.
- The -k and -s options must not be used in conjunction with any other option.
- The session ID argument for the -k option is printed when the ping request is started.
- Some echo servers have a maximum length for responses and will truncate any responses longer than the maximum.
- The system must be configured with DNS server information (via 'setdnssrv') prior to pinging a device via its hostname.

queues - Display NicheStack packet queues

---

queues

```
queues - Display NicheStack packet queues
```

Syntax

```
queues
```

Parameters

-z	(no parameter). Resets all queues' min and max threshholds to their current values.

Description

This command displays the current status of various resource queues. These queues include the receive packet queue (rcvdq), free packet queues (cb-#), the free mbuf queue (mfreeq) and the 'inuse' mbuf queue (mbufq).

Sample Output:

```
-> queues
Packet buffer free queues, total packets:120 (99840 bytes)
cb-1 Q0128: size:7680, head:00A5A9FC, tail:00A5A96C, len:60, min:57, max:60
cb-2 Q1536: size:92160, head:00A5E784, tail:00A5E73C, len:60, min:52, max:60
mfreeq: head:00A5D82C, tail:00A5D804, len:120, min:118, max:120
mbufq: head:00000000, tail:00000000, len:0, min:0, max:2
rcvdq: head:00000000, tail:00000000, len:0, min:0, max:3
```

setip - manually set IPv4 or IPv6 address information

Command Name

```
setip - manually set IPv4 or IPv6 address information
```

Syntax

```
setip -i <interface number> [-a <IP address>] [-s <IPv4 subnet mask>] [-g <IPv4 gateway address>] [-d <IPv6 address>] [[-o | -r] -p <protocol>]
```

Parameters

-a	Network interface's IPv4 or IPv6 address.
-d	Network interface's IPv6 address.
-g	Network interface's IPv4 gateway address.
-i	Network interface number.
-o	Obtain an IPv4 address using the specified protocols (e.g., DHCP ("dhcpc"), auto-configuration ("autoip")).

-p	Set of address protocols to be used on the link (e.g., "dhcpc" (DHCP only), "autoip" (auto-configuration only), "dhcpc,autoip" (DHCP or auto-configuration)).
-r	Terminate the use of the specified address protocols on the link, and relinquish address obtained thru' them.
-s	Network Interface's IPv4 subnet mask.

Description

This command configures a network interface. The '-i' parameter selects the network interface to configure. The user can set the IPv4 address, IPv4 subnet mask, and gateway IPv4 address. An IP address is specified in the following format: NNN.NNN.NNN.NNN, where NNN can range from 0 to 255. The format: NNN.NNN, is shorthand for NNN.0.0.NNN.

This command can also be used to add or delete IPv6 addresses from a specified interface. The '-a' parameter will also accept an IPv6 address in the form XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX. The setip command may be used repeatedly to add more than one IPv6 unicast address. An interface can have only one link-local IPv6 address, and upto three global IPv6 addresses. The '-d' parameter will accept an IPv6 address to delete from the interface.

Changes to the Network Interface configuration are performed immediately; open connections are not flushed or closed. If no parameters are specified, the current IPv4 configuration settings are displayed.

Notes/Status

- When both DHCP and auto-configuration are specified on a link, the system first attempts to obtain an address via DHCP.
- Setting incorrect values may cause the network interface to become unusable.
- Example invocation sequences:
    1. configure a static IPv4 address, subnet mask, and gateway address

    ```
    setip -i 1 -a 10.0.0.112 -s 255.255.255.0 -g 10.0.0.1
    ```

    2. configure a global IPv6 address on interface #1

    ```
    setip -i 1 -a 3FFE:501:FFFF::211:11FF:FEBE:7F62
    ```

    3. delete a global IPv6 address on interface #1

    ```
    setip -i 1 -d 3FFE:501:FFFF::211:11FF:FEBE:7F62
    ```

    4. configure interface #1 to obtain an IPv4 address via DHCP

    ```
    setip -i 1 -o -p dhcpc
    ```

    5. configure interface #1 to obtain an IPv4 address via auto-configuration

    ```
    setip -i 1 -o -p autoip
    ```

    6. configure interface #1 to obtain an IPv4 address via DHCP or auto-configuration in this case, the system first attempts to obtain an address via DHCP. If that fails, it auto-configures itself with an address in the range from 169.254.1.0 to 169.254.254.255.

    ```
    setip -i 1 -o -p dhcpc,autoip
    ```

    7. terminate the use of DHCP on interface #1, and release address obtained via that protocol. If auto-configuration was previously configured, the system will attempt to obtain an address via that protocol. Otherwise, it will revert to the previously configured static IPv4 address.

    ```
    setip -i 1 -r -p dhcpc
    ```

setdnssrv - Add, delete, or display IP addresses from the table of DNS name servers.

Command Name

setdnssrv - Add, delete, or display IP addresses from the table of DNS name servers.

Syntax

```
setdnssrv

setdnssrv -a <IP address> [-i <server index>]

setdnssrv -d -i <server index>
```

Parameters

-a	ASCII string providing the IPv4 or IPv6 address of a DNS server
-d	No parameters
-i	Ones-based index of a name server to add or delete

Description

Add, delete, or display IP addresses from the table of DNS name servers

Notes/Status

- '-a' and '-d' arguments are mutually exclusive.
- The '-i' argument is required with '-d'
- Command without arguments displays the current state of the DNS server table
- The DNS server table can contain a maximum of three (IPv4 and/or IPv6) addresses.

Sample output, when there is already a nameserver at index 1:

```
-> setdnssrv -i 2 -a 10.0.0.1
DNS servers: 10.0.0.226, 10.0.0.1, Invalid addr,
```

status - display system status

Name

```
status - display system status
```

Syntax

```
status [-i] [-m] [-q] [-s]
```

Parameters

-i	Display IP MIB information.
-m	Display MBuf information.
-q	Display packet queue information.
-s	Display system status information.

Description

This command displays information about various embTCP components.

osinfo - display FreeRTOS status information

---

debug

```
osinfo - display FreeRTOS status information
```

Syntax

```
osinfo
```

Parameters

none

Description

This command dumps status information (tick count, task status, etc.) for the FreeRTOS operating system.

Related Products

This product was derived from a portable, flexible and more full-featured product available from InterNiche Technologies, Inc. For more information about this **SOURCE CODE PRODUCT**, please visit www.iNiche.com or email Sales@iNiche.com.

For Additional Information ...

- InterNiche Support Site
- FreeRTOS web site
- Unix Network Programming, Volume 1 by Richard Stevens.