# CrossWorks TCP/IP Library

**Version: 3.2**

Copyright 2013 Rowley Associates Limited

# Contents

CrossWorks TCP/IP Library

# CrossWorks TCP/IP Library

The *CrossWorks TCP/IP Library* is a collection of functions and device drivers that add TCP/IP networking to your application. We have primarily designed the TCP/IP Library to work well on reduced-memory real-time embedded systems that require network connectivity, but you can equally well use the library on faster processors with more memory.

The TCP/IP Library is designed to run exclusively in the CrossWorks tasking environment; if your application doesn't use tasking and you wish to use this product then you must convert your application to run in a tasking environment which is simple enough to do. If you are using some other real time operating system, then using the TCP/IP Library is not viable and should seek a product that integrates well with your existing RTOS—or ditch that RTOS and use our excellent CTL tasking environment instead.

As you would expect, the TCP/IP Library integrates with other components in the CrossWorks Target Library. For instance, the TCP/IP Library uses the CrossWorks Mass Storage Library to store and retrieve files using FTP, or serve web pages from files in the file system. The file system and the TCP/IP Library both integrate with the CrossWorks Streams framework.

# Object Code Evaluation License

If you are evaluating the TCP/IP Library for use in your product, the following terms apply.

## General terms

The source files and object code files in this package are not public domain and are not open source. They represent a substantial investment undertaken by Rowley Associates to assist CrossWorks customers in developing solutions using well-written, tested code.

## Library Evaluation License

Rowley Associates grants you a license to the Object Code provided in this package solely to evaluate the performance and suitability of this library for inclusion into your products. You are prohibited from extracting, disassembling, and reverse engineering the Object Code in this package.

# Object Code Commercial License

If you have paid to use the TCP/IP Library in your product, the following terms apply.

## General terms

The source files and object code files in this package are not public domain and are not open source. They represent a substantial investment undertaken by Rowley Associates to assist CrossWorks customers in developing solutions using well-written, tested code.

## Object Code Commercial License

If you hold a paid-for Object Code Commercial License for this product, you are free to incorporate the object code in your own products without royalties and without additional license fees. This Library is licensed to you PER DEVELOPER and is associated with a CrossWorks Product Key which, when combined, forms the entitlement to use this library. You must not provide the library to other developers to link against: each developer that links with this Library requires their own individual license.

# Prerequisites

## What's in the box?

As delivered the TCP/IP Library provides the following core TCP/IP protocols in object form:

- ARP, UDP, TCP, DHCP, NTP, and DNS

The stack also provides examples of application-level protocols in source form that you can customize:

- FTP, HTTP, SMTP

You can extend the capabilities of the TCP/IP Library by writing your own functions to implement other application-level UDP and TCP protocols just as we have implemented the existing application-level protocols using core protocols.

## What we assume you know

This user manual is a user manual for our network stack. You don't know anything about our stack or how it works, so this manual teaches you how to use it. This user manual is not a 'Dummies Guide to TCP/IP', because you don't find that in the title: we expect you to know what you want to do but not how to achieve it using our software.

You need a good understanding of how TCP/IP and Ethernet work and the underlying concepts. If you know nothing to very little about TCP/IP, don't know what a datagram is or the difference between a TCP segment and grapefruit segment, you're not really ready to swim with sharks just yet—then check out the following books to expand your horizons:

- *TCP/IP Illustrated, Volume 1: The Protocols*, W. Richard Stevens, ISBN 978-0201633467.
- *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*, W. Richard Stevens, ISBN 978-0201634952.

In fact, the books above are a great reference for anybody that needs to use TCP/IP on a daily basis, so go and buy them.

If all you want to do is send an e-mail from the network stack, this manual alone is enough to construct a solution for that. If you want to write your own TCP and datagram protocol handler, this manual shows you the mechanisms to achieve that goal. What this manual does not do is tell you how to make UDP protocols 'reliable' or how to design your own protocols—that's all up to you, we just provide the necessary parts kit for you to assemble your application.

This manual tells you how you can use some of the TCP/IP Library's built-in features that help when you're debugging your code. It doesn't tell you how to go about debugging your application or how to use a network analyzer to track down rogue packets, how to figure out which rabbit hole a particular packet disappeared down, or how to tune out noise and dig deep into packets scuttling across the network—you need to acquire

those skills yourself, it's called ‘being an software developer.’ When you're a true network warrior, buy yourself a celebration beer and T-shirt. Nobody said this profession was easy.

## What's not in the box?

I must also take the opportunity to tell you what is not included in your purchase. No, we do not include your favorite bizarre protocol for controlling a network coffee pot. There are so many protocols built upon TCP and UDP that it is impossible to offer implementations for them all, so we offer the useful few. It just means you need to implement the Coffee Pot Control Protocol yourself for that must-have network-attached Espresso machine, or find somebody who has the necessary experience and has done so already. Start your search by Googling ‘RFC 2324’.

## Product support and questions

If you ask us for support about things that you should really know yourself, don't be surprised or offended when we tell you that product support doesn't include hand-holding, nursemaid duties, or writing your application for you, no matter how nicely you ask.

If you ask us a question that can be answered by reading the manual, don't be surprised if you receive a short, to-the-point reply. I am writing this documentation for a reason: if I have taken the time to write it, you really should take the time to read it, or at least search it. Impending product deadlines do not excuse you from using our support service as an on-demand oracle.

And as a final request, *never* end you e-mails with ‘Please advise’ because that really ticks me off.

With all that understood, let's begin.

# Before you begin

## Simplify your life

Your intention is to deploy, or evaluate, the *CrossWorks TCP/IP Library* for use in your product. Before you begin, there is something very important that I must ask you to do: *run on known-good hardware with tested software!* You don't want to make your life complicated to begin with. You don't want to port the TCP/IP Library to an untested piece of hardware, as well as learn about the TCP/IP Library and, maybe, even learn CrossWorks at the same time. So, do yourself a favor and spend a little money getting a piece of hardware that is fully tested and that we know runs the TCP/IP Library well.

## Purchase a SolderCore

Suggestions? Well, the TCP/IP Library is primarily developed using the SolderCore, and as Rowley Associates manufactures the SolderCore we would recommend most highly that you purchase one, or more, of these to start learning how the TCP/IP Library works. You'll feel so much better running networking examples straight away, and then you can progress to other hardware and see how it works out for you.

## Tested examples

This manual is written using the TCP/IP Library examples that come included with boards that are pre-configured, ready to run networking, as part of a CrossWorks Board Support Package. Not all Board Support Packages contain networking examples—they may not have them because we haven't supported the embedded or external network controller, or because they are too limited to run networking.

If you are familiar with TCP/IP networking, CrossWorks, and are comfortable skipping the manual and diving straight into code with a reference manual, that's great, go right ahead and try out some of the examples…

## What you need to know

To try out the networking examples, there's very little that you need to know about CrossWorks and the Platform Library. All you need is a board that we ship examples for and a way to program it. If you want to start delving a little deeper into the examples, you will need to refer to the *Platform Library* user manual as the examples use Platform Library facilities to make the code portable over all the boards we support.

There are many examples that you can extract code from: inheritance by text editor is a tried and tested method of program development! Because all the support code is provided in source form, you can copy that into your application to get it working.

# Get on the network

## Your first TCP/IP Library application

So you have a board, you have a network, and you're ready to attach your device to the network. The first thing to do is establish that basic Ethernet communication works between your PC and your evaluation board.

## Install board support

Install the board support package for the evaluation board that you have purchased. From now on we will assume this is the SolderCore, but you can substitute your own board as required. So, install the *SolderCore Board Support Package* into CrossWorks using the package manager, **Tools** > **Package Manager**.

## Load the board examples

The easiest way to load the examples for the board is to open up the **Contents** window, navigate to **Board Support**, expand the **SolderCore Board Support Package** item, and click the **SolderCore Samples Solution**.

## Select and build the project

In the examples for your board, you'll find a **Networking Projects** solution, and within that a **Minimal Network with Ping (Fixed IP address)** project. Double-click that project to make it active and press **F7** to build. This will compile cleanly: we've tested this before release. If it doesn't build cleanly, that usually means that you're missing one of the packages that the board support package requires, or you've edited something within a support package—if this is the case, you'll need to figure out what you've done or get in touch with us.

## Find a spare network address

As the example we are going to run uses a fixed IP address, you need to find a free one to assign to the evaluation board. On Windows, you can use `ipconfig` to view your network parameters:

```
> ipconfig
Windows IP Configuration

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : rowley.co.uk
   Link-local IPv6 Address . . . . . : fe80::9c2d:e057:8641:2281%10
   IPv4 Address. . . . . . . . . . . : 10.0.0.58
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.0.0.3

> _
```

Here we see that the subnet mask is `255.255.255.0` and the PC's IP address is `10.0.0.58`. So, let's try a random IP address, by changing the last number, to see if it's free:

```
> ping 10.0.0.32

Pinging 10.0.0.32 with 32 bytes of data:
Reply from 10.0.0.32: bytes=32 time<1ms TTL=64
Reply from 10.0.0.32: bytes=32 time<1ms TTL=64
```

```
Reply from 10.0.0.32: bytes=32 time<1ms TTL=64
Reply from 10.0.0.32: bytes=32 time<1ms TTL=64

Ping statistics for 10.0.0.32:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

> _
```

Ahh, that one's in use. Let's try another:

```
> ping 10.0.0.44

Pinging 10.0.0.44 with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 10.0.0.44:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

> _
```

OK, that one seems free as the host is not reachable on the network.

You might see a variation on the above:

```
> ping 10.0.0.44

Pinging 10.0.0.44 with 32 bytes of data:
Reply from 10.0.0.58: Destination host unreachable.
Reply from 10.0.0.58: Destination host unreachable.
Reply from 10.0.0.58: Destination host unreachable.
Reply from 10.0.0.58: Destination host unreachable.

Ping statistics for 10.0.0.44:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),

> _
```

This indicates that the ping request was answered, in this case by 10.0.0.58, with a response that says that the IP address 10.0.0.44 cannot be reached.

### Configure the board's network

Double-click the file `example_minimal_ping_fixed_ipaddr.c` in the **Source Files** folder and it will open in the code editor:

```
// Set up network using a fixed IP address.

#include "libnet/ctl_net_private.h"
#include "libplatform/platform.h"
#include "libplatform/platform_network.h"
#include "example_support.h"
#include <string.h>
```

```
// TODO: You must alter these to match your network!
#define FIXED_IP_ADDRESS   "10.0.0.44"
#define FIXED_NETMASK      "255.255.255.0"

// Assign a fixed MAC address to the NIC.  Normally this will be blown into
// OTP or some other nonvolatile medium when the device is personalized as
// part of production.
#define FIXED_MAC_ADDRESS  "bc-28-d6-ff-ff-ff"

// Thread Priority
#define NET_TASK_PRIORITY  200

// Network interface,
static CTL_NET_INTERFACE_t nic;

static void
bring_up_network(void)
{
  CTL_IP_CONFIG_t ip_config;

  // Clear network IP configuration for population.
  memset(&ip_config, 0, sizeof(CTL_IP_CONFIG_t));

  // Assign fixed IP address and subnet mask.
  ip_config.ip_addr = ctl_net_scan_dot_decimal_ip_addr(FIXED_IP_ADDRESS);
  ip_config.subnet_mask = ctl_net_scan_dot_decimal_ip_addr(FIXED_NETMASK);

  // Assign a fixed MAC address to the NIC.
  example_check_status(ctl_net_scan_mac_addr(&nic.mac.mac_addr, FIXED_MAC_ADDRESS));

  // Bring up network.
  example_check_status(ctl_mac_init(&nic));

  // Bring up the IP network.
  example_check_status(ctl_net_init(NET_TASK_PRIORITY, &ip_config));

  // Bring up only ICMP to respond to pings.
  example_check_status(ctl_icmp_init());
}

int
main(void)
{
  char dot_ipaddr[16], dot_netmask[16];

  // Initialize platform.
  platform_initialize();

  // Configure the NIC for this platform.
  example_check_status(platform_configure_network(&nic));

  // Start network.
  bring_up_network();

  // Idle away, the network task responds to pings.
  for (;;)
    {
      // Dump message inviting a ping.
      printf("IP address is %s and subnet mask is %s\n",
             ctl_ip_sprint_addr(dot_ipaddr, ctl_net_get_ip_address()),
             ctl_ip_sprint_addr(dot_netmask, ctl_net_get_subnet_mask()));
```

```
      // Don't be too enthusiastic with messages.
      ctl_delay(1000);
    }
}
```

Modify the definition of `FIXED_IP_ADDRESS` to match your selected IP address and `FIXED_NETMASK` to match your subnet mask.

Power up and attach a network cable to your evaluation board, and press **F5** to run your code. The application downloads and, if CrossWorks is configured to stop at `main`, press **F5** again to continue running the code.

In the CrossWorks **Debug Terminal** you should see something similar to the following, but with your selected IP address and subnet mask:

```
IP address is 10.0.0.44 and subnet mask is 255.255.255.0
IP address is 10.0.0.44 and subnet mask is 255.255.255.0
IP address is 10.0.0.44 and subnet mask is 255.255.255.0
```

This is inviting you to ping the board. So, do it:

```
> ping 10.0.0.44

Pinging 10.0.0.44 with 32 bytes of data:
Reply from 10.0.0.44: bytes=32 time<1ms TTL=64
Reply from 10.0.0.44: bytes=32 time<1ms TTL=64
Reply from 10.0.0.44: bytes=32 time<1ms TTL=64
Reply from 10.0.0.44: bytes=32 time<1ms TTL=64

Ping statistics for 10.0.0.44:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

> _
```

### That's it!

So, you now have a functioning Ethernet connection between your PC and your target board!

# Don't break it...

### Your second TCP/IP Library application

It's not that common to use a fixed IP address for a network-attached device. Modern networks use dynamically-assigned IP addresses and a *DHCP server* to manage assignment: when a device powers on, it broadcasts a request to the network asking for a DHCP server to assign it an IP address. Using a DHCP server is now a necessity with so many devices attached to a LAN, there is no way that a human can possibly manage a large network without error.

### Select and build the project

In the examples for your board, you'll find a **Networking Projects** solution, and within that a **Minimal Network with Ping (DHCP IP address)** project. Double-click that project to make it active and press **F7** to build.

Double-click the file `example_minimal_ping_fixed_ipaddr.c` in the **Source Files** folder and it will open in the code editor:

```
// Set up network using a DHCP-assigned IP address.

#include "libnet/ctl_net_private.h"
#include "libplatform/platform.h"
#include "libplatform/platform_network.h"
#include "example_support.h"

// Assign a fixed MAC address to the NIC.  Normally this will be blown into
// OTP or some other nonvolatile medium when the device is personalized as
// part of production.
#define FIXED_MAC_ADDRESS        "bc-28-d6-ff-ff-ff"

// Network task thread priority
#define NET_TASK_PRIORITY        200

// Network interface,
static CTL_NET_INTERFACE_t nic;

static void
bring_up_network(void)
{
  // Assign a fixed MAC address to the NIC.
  example_check_status(ctl_net_scan_mac_addr(&nic.mac.mac_addr, FIXED_MAC_ADDRESS));

  // Initialize MAC.
  example_check_status(ctl_mac_init(&nic));

  // Bring up network task and use DHCP to assign an IP address.
  example_check_status(ctl_net_init(NET_TASK_PRIORITY, 0));

  // Bring up UDP and ICMP: DHCP requires UDP, and ICMP will respond to pings.
  example_check_status(ctl_udp_init(0));
  example_check_status(ctl_icmp_init());

  // Start DHCP to assign us an IP address.
  example_check_status(ctl_dhcp_init());
```

```
}

int
main(void)
{
  char dot_ipaddr[16], dot_netmask[16];

  // Initialize platform.
  platform_initialize();

  // Initialize NIC for this platform.
  example_check_status(platform_configure_network(&nic));

  // Start network.
  bring_up_network();

  // Idle away; when we're configured, dump our network.
  for (;;)
    {
      // See if we've acquired an IP address yet...
      if (ctl_net_get_ip_address())
        {
          // Dump message inviting a ping.
          printf("DHCP: IP address is %s and subnet mask is %s\n",
                 ctl_ip_sprint_addr(dot_ipaddr, ctl_net_get_ip_address()),
                 ctl_ip_sprint_addr(dot_netmask, ctl_net_get_subnet_mask()));
        }
      else
        {
          // Can't ping me yet.
          printf("DHCP: awaiting IP address assignment\n");
        }

      // Don't be too enthusiastic with messages.
      ctl_delay(1000);
    }
}
```

There's no fixed IP address in this, but there is an option to start up the DHCP client subsystem to manage acquisition of DHCP-assigned IP addresses.

### See if it works

Power up and attach a network cable to your evaluation board, and press **F5** to run your code. The application downloads and, if CrossWorks is configured to stop at `main`, press **F5** again to continue running the code.

In the CrossWorks **Debug Terminal** you should see something similar to the following, but with your DHCP-assigned IP address and subnet mask:

```
DHCP: awaiting IP address assignment
DHCP: awaiting IP address assignment
DHCP: awaiting IP address assignment
DHCP: IP address is 10.0.0.44 and subnet mask is 255.255.255.0
DHCP: IP address is 10.0.0.44 and subnet mask is 255.255.255.0
DHCP: IP address is 10.0.0.44 and subnet mask is 255.255.255.0
```

You can ping the device to make sure that it does indeed work.

### Job done!

You now have a functioning Ethernet connection between your PC and your target board, using a dynamically-assigned IP address. However, it's a bit of a bore to type in IP addresses each time, and as the IP address may change, how do you know which IP address to use?

# Ping by name

## Your third TCP/IP Library application

What would be great is if your evaluation board had a name, rather than an address, so we can simply ping the name of the board. Well, there is a way, and that is to register a name using DHCP.

## Select and build the project

In the examples for your board, you'll find a **Networking Projects** solution, and within that a **Ping by Name** project. Double-click that project to make it active and press **F7** to build.

Double-click the file `example_ping_by_name.c` in the **Source Files** folder and it will open in the code editor. In `main` you'll find a call to **ctl_net_set_host_name**, before the network is brought up, to set the host name of the evaluation board:

```
// Set our host name.
ctl_net_set_host_name("crossworks");
```

This registers the name of the host with the DHCP server and means that you can ping the board using a friendly name, whatever the assigned IP address is.

## See if it works

Power up the board and run the code. In the CrossWorks **Debug Terminal** you should see the same as before:

```
DHCP: awaiting IP address assignment
DHCP: awaiting IP address assignment
DHCP: awaiting IP address assignment
DHCP: IP address is 10.0.0.44 and subnet mask is 255.255.255.0
DHCP: IP address is 10.0.0.44 and subnet mask is 255.255.255.0
DHCP: IP address is 10.0.0.44 and subnet mask is 255.255.255.0
```

Now you can ping the device by its assigned name, `crossworks`:

```
> ping crossworks

Pinging crossworks.rowley.co.uk [10.0.0.44] with 32 bytes of data:
Reply from 10.0.0.44: bytes=32 time<1ms TTL=64
Reply from 10.0.0.44: bytes=32 time<1ms TTL=64
Reply from 10.0.0.44: bytes=32 time<1ms TTL=64
Reply from 10.0.0.44: bytes=32 time<1ms TTL=64

Ping statistics for 10.0.0.44:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

>_
```

Notice that the full name of the host is `crossworks.rowley.co.uk`. This is because the LAN that the board is connected to has a domain name suffix. You might have noticed this in the output from `ipconfig` in the first example:

```
> ipconfig
Windows IP Configuration

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : rowley.co.uk
   Link-local IPv6 Address . . . . . : fe80::9c2d:e057:8641:2281%10
   IPv4 Address. . . . . . . . . . . : 10.0.0.58
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.0.0.3

> _
```

The full host name is the name that we assigned to the node, `crossworks`, with the suffix assigned by the network, `rowley.co.uk`.

### Job done!

You now have a functioning Ethernet connection between your PC and your target board, using a dynamically-assigned IP address, and with a friendly name to contact the board.

### The code

```
// Set up network using a DHCP-assigned IP address.

#include "libnet/ctl_net_private.h"
#include "libplatform/platform.h"
#include "libplatform/platform_network.h"
#include "example_support.h"

// Assign a fixed MAC address to the NIC.  Normally this will be blown into
// OTP or some other nonvolatile medium when the device is personalized as
// part of production.
#define FIXED_MAC_ADDRESS        "bc-28-d6-ff-ff-ff"

// Network task thread priority
#define NET_TASK_PRIORITY        200

// Network interface,
static CTL_NET_INTERFACE_t nic;

static void
bring_up_network(void)
{
  // Assign a fixed MAC address to the NIC.
  example_check_status(ctl_net_scan_mac_addr(&nic.mac.mac_addr, FIXED_MAC_ADDRESS));

  // Initialize MAC.
  example_check_status(ctl_mac_init(&nic));

  // Bring up network task and use DHCP to assign an IP address.
  example_check_status(ctl_net_init(NET_TASK_PRIORITY, 0));

  // Bring up UDP and ICMP: DHCP requires UDP, and ICMP will respond to pings.
  example_check_status(ctl_udp_init(0));
  example_check_status(ctl_icmp_init());

  // Start DHCP to assign us an IP address.
```

```
  example_check_status(ctl_dhcp_init());
}

int
main(void)
{
  char dot_ipaddr[16], dot_netmask[16];

  // Initialize platform.
  platform_initialize();

  // Initialize NIC for this platform.
  example_check_status(platform_configure_network(&nic));

  // Set our host name.
  ctl_net_set_host_name("crossworks");

  // Start network.
  bring_up_network();

  // Idle away; when we're configured, dump our network.
  for (;;)
    {
      // See if we've acquired an IP address yet...
      if (ctl_net_get_ip_address())
        {
          // Dump message inviting a ping.
          printf("DHCP: IP address is %s and subnet mask is %s\n",
                 ctl_ip_sprint_addr(dot_ipaddr, ctl_net_get_ip_address()),
                 ctl_ip_sprint_addr(dot_netmask, ctl_net_get_subnet_mask()));
        }
      else
        {
          // Can't ping me yet.
          printf("DHCP: awaiting IP address assignment\n");
        }

      // Don't be too enthusiastic with messages.
      ctl_delay(1000);
    }
}
```

## See Also

ctl_net_set_host_name

# Resolving host names

## Finding IP addresses

You've seen how to get your board registered with a name on the LAN. Now it's time to step outside and get onto the Internet. This example is how to resolve the IP address of the Rowley Associates web server, `www.rowley.co.uk`.

## Select and build the project

In the examples for your board, you'll find a **Networking Projects** solution, and within that a **Resolve Domain Name** project. Double-click that project to make it active and press **F7** to build.

## Hiding some details

Rather than repeat all the boilerplate code that brings up the network and waits for an IP address, that code is moved into `example_network_support.c`. This example, and all following examples, assume that `example_network_support.c` is included in the project.

Double-click the file `example_network_support.c` in the **Source Files** folder and it will open in the code editor. Because this example needs to resolve a domain name, it initializes the *Domain Name System* component of the TCP/IP Library:

```
// Start DNS for domain name lookup.
stat = ctl_dns_init();
if (stat < CTL_NO_ERROR)
  return stat;
```

Initializing the DNS part of the TCP/IP Library enables you to resolve human-readable domain names, such as `www.rowley.co.uk` into an IP address you can communicate with.

## About DNS

In order to resolve a domain name to an IP address, you must have already set the domain name server (or servers) that the TCP/IP Library communicates with to resolve the domain name. If you are using DHCP to configure the TCP/IP Library, which we assume from here on, the domain name servers are automatically set as part of IP address assignment with DHCP.

If you are using a static IP address then you must configure the DNS servers the stack uses by passing in the IP addresses of the primary and (optional) secondary server when initializing the network (see **ctl_net_init** and **CTL_IP_CONFIG_t**).

## Client code

Double-click the file `example_resolve_domain_name.c` in the **Source Files** folder and it will open in the code editor. The example is now much smaller:

```
// Resolve a domain name.

#include "libnet/ctl_net_api.h"
#include "libplatform/platform.h"
#include "libplatform/platform_network.h"
#include "example_support.h"

int
main(void)
{
  CTL_STATUS_t stat;
  CTL_NET_IPv4_ADDR_t addr;
  char dot_ipaddr[16];

  // Initialize platform.
  platform_initialize();

  // Start networking, wait for an IP address.
  example_check_status(example_bring_up_full_networking());
  example_check_status(example_await_assigned_ip_address());

  // Dump the primary domain name server, for reference.
  printf("Using DNS server %s\n",
         ctl_ip_sprint_addr(dot_ipaddr, ctl_dns_primary_server_addr()));

  // Try to resolve www.rowley.co.uk.  Wait a maximum of two
  // seconds for an answer.
  stat = ctl_dns_get_host_by_name("www.rowley.co.uk", &addr, 2000);

  // Did this resolve?
  if (stat < CTL_NO_ERROR)
    {
      // No.
      printf("Could not resolve www.rowley.co.uk!\n");
    }
  else
    {
      // Yes, print the resolved IP address.
      printf("www.rowley.co.uk resolved to %s\n",
             ctl_ip_sprint_addr(dot_ipaddr, addr));
    }

  // Done.
  return example_finish();
}
```

The part of interest is:

```
stat = ctl_dns_get_host_by_name("www.rowley.co.uk", &addr, 2000);
```

This sends a request to the DNS server to resolve the domain name `www.rowley.c.uk` and deliver the result to **addr**. The third parameter, `2000`, indicates the maximum duration we're prepare to wait for—in this case, two seconds.

### See if it works

Power up the board and run the code. In the CrossWorks **Debug Terminal** you will see something similar to this:

```
DHCP: awaiting IP address assignment
```

```
DHCP: awaiting IP address assignment
DHCP: awaiting IP address assignment
DHCP: IP address is 10.0.0.44 and subnet mask is 255.255.255.0
Using DNS server 10.0.0.8
www.rowley.co.uk resolved to 178.236.4.60
Finished.
```

## See Also

ctl_dns_get_host_by_name

# Retrieving a web page

### Start of a web browser...

We build upon the capabilities of previous examples by showing how to grab the contents of a web page from the Internet. This example shows how to dump the HTML data of the Rowley Associates home page at `www.rowley.co.uk/index.htm`.

### Select and build the project

In the examples for your board, you'll find a **Networking Projects** solution, and within that a **Retrieve Web Page** project. Double-click that project to make it active and press **F7** to build.

### Sockets

Double-click the file `example_retrieve_web_page.c` in the **Source Files** folder and it will open in the code editor. This example is longer than before, but then it does much more than previous examples.

Once the domain name is resolved, the example needs to communicate with the web server in order to download the web page. It does this by creating a socket and connecting the socket to the server:

```
// Open a socket to the host on port 80.
s = ctl_tcp_socket();
if (s < CTL_NO_ERROR)
  example_terminate("No sockets available\n");
example_check_status(ctl_tcp_connect(s, addr, HTONS(80), 1000));
```

**ctl_tcp_socket** creates a new socket and **ctl_tcp_connect** connects that socket to the server. The parameters to `ctl_tcp_connect` are:

- The socket, created by **ctl_tcp_socket**.
- The IP address of the server to connect to. The IP address in this example is resolved using DNS using **ctl_dns_get_host_by_name**.
- The TCP/IP port to connect to. HTTP connections use port 80, and HTONS converts the port number from host byte order to network byte order as required by the TCP/IP Library.
- The maximum time to wait for the connection to be made. In this example we are prepared to wait one second for the connection to be established.

### Sending the request

Once the socket is established, you start to communicate with the server using a GET request. For reference, the HTTP protocol is fully described in RFC2616.

The GET request consists of the command, the headers, and a blank line to terminate the headers:

```
ctl_tcp_printf(s, "GET http://%s/%s HTTP/1.0\r\n", host, name);
ctl_tcp_printf(s, "Accept: text/plain\r\n");
ctl_tcp_printf(s, "Host: %s\r\n", host);
ctl_tcp_printf(s, "\r\n");
```

```
ctl_tcp_push(s);
```

The application sends:

- The GET command specifying the URL and the protocol to use. In this case, the URL is composed of the host name and the page we are interested in. Following that is the protocol to use, `HTTP/1.0`, which simplifies the following code somewhat.
- The headers. This tells the server the MIME type of the response and the host we are addressing.
- A blank line which indicates that the headers are complete.

### An important difference

One thing to notice is the call to **ctl_tcp_push**: this ensures that the data sent to the socket gets pushed to the network and sent out on the wire. The TCP/IP Library buffers data on a socket until a TCP segment is full, when it is pushed to the network—to flush a partially-filled segment, call **ctl_tcp_flush**. This makes the TCP/IP Library different from classic TCP stacks which will typically flush a partially-filled segment to the network after a short timeout.

### Reading the response

Once the headers are sent, the example reads the response from the server using repeated calls to `tcp_read_line`. We specified `HTTP/1.0` which requests the server to close the connection after sending all its data, and take advantage of the fact that when a socket is closed, we'll receive an error if we try to read more from it, and we exit the loop:

```
// Process response.
for (;;)
  {
    // Try to read a whole line from the web server.
    stat = ctl_tcp_read_line(s, line_buffer, sizeof(line_buffer)-1);

    // Ensure the buffer is terminated.
    line_buffer[sizeof(line_buffer)-1] = 0;

    // Process return status.
    if (stat == CTL_NET_ERR_TIMEDOUT)
      {
        // Didn't get anything, loiter...
      }
    else if (stat < CTL_NO_ERROR)
      {
        // Error reading the socket or the socket closed?
        break;
      }
    else
      {
        // Dump response.
        printf("%s\n", line_buffer);
      }
  }

// Make sure socket is closed.
ctl_tcp_shutdown(s);
```

Before exiting, we close the socket. If the socket is already closed because the server closed it, closing it a second time makes no difference.

### See if it works

Power up the board and run the code. In the CrossWorks **Debug Terminal** you will see something similar to this:

```
DHCP: awaiting IP address assignment
DHCP: awaiting IP address assignment
DHCP: awaiting IP address assignment
DHCP: IP address is 10.0.0.44 and subnet mask is 255.255.255.0
Using DNS server 10.0.0.8
www.rowley.co.uk resolved to 178.236.4.60
Connecting to www.rowley.co.uk (178.236.4.60)...
Requesting ...
HTTP/1.1 200 OK
Date: Mon, 09 Sep 2013 13:17:33 GMT
Last-Modified: Thu, 29 Aug 2013 08:37:15 GMT
Content-Type: text/html
Content-Length: 13841
Connection: keep-alive
Server: AmazonS3

<!DOCTYPE HTML>
<html>
…
```

### Job done!

You now have a way to communicate with an HTTP server. You'll find that many servers will communicate in much the same way: a command, some headers, a blank line, and read the response, so you have a starting point at least.

### The code

```
// Retrieve a web page.

#include "libnet/ctl_net_api.h"
#include "libplatform/platform.h"
#include "libplatform/platform_network.h"
#include "example_support.h"

// Static data.
static char line_buffer[512];

static void
example_retrieve_web_page(const char *host, const char *name)
{
  CTL_NET_IPv4_ADDR_t addr;
  CTL_SOCKET_t s;
  CTL_STATUS_t stat;
  char str[16];

  // Try to resolve host.
  stat = ctl_dns_get_host_by_name(host, &addr, 2000);

  // Did this resolve?
  if (stat < CTL_NO_ERROR)
```

```
    {
      // No.
      example_terminate("Could not resolve www.rowley.co.uk!\n");
    }
  else
    {
      // Yes, print the resolved IP address.
      printf("%s resolved to %s\n",
             host,
             ctl_ip_sprint_addr(str, addr));
    }

  // User needs to know...
  printf("Connecting to %s (%s)...\n",
         host,
         ctl_ip_sprint_addr(str, addr));

  // Open a socket to the host on port 80.
  s = ctl_tcp_socket();
  if (s < CTL_NO_ERROR)
    example_terminate("No sockets available\n");
  example_check_status(ctl_tcp_connect(s, addr, HTONS(80), 1000));

  // Send the request
  printf("Requesting %s...\n", name);
  ctl_tcp_printf(s, "GET http://%s/%s HTTP/1.0\r\n", host, name);
  ctl_tcp_printf(s, "Accept: text/plain\r\n");
  ctl_tcp_printf(s, "Host: %s\r\n", host);
  ctl_tcp_printf(s, "\r\n");
  ctl_tcp_push(s);

  // Process response.
  for (;;)
    {
      // Try to read a whole line from the web server.
      stat = ctl_tcp_read_line(s,
                               line_buffer, sizeof(line_buffer)-1,
                               CTL_TIMEOUT_DELAY, 4000);

      // Ensure the buffer is terminated.
      line_buffer[sizeof(line_buffer)-1] = 0;

      // Process return status.
      if (stat == CTL_NET_ERR_TIMEDOUT)
        {
          // Didn't get anything, loiter...
        }
      else if (stat < CTL_NO_ERROR)
        {
          // Error reading the socket or the socket closed?
          break;
        }
      else
        {
          // Dump response.
          printf("%s\n", line_buffer);
        }
    }

  // Make sure socket is closed.
  ctl_tcp_shutdown(s);
}
```

```
int
main(void)
{
  // Initialize platform.
  platform_initialize();

  // Start networking, wait for an IP address.
  example_check_status(example_bring_up_full_networking());
  example_check_status(example_await_assigned_ip_address());

  // Send headers, read web page.
  example_retrieve_web_page("www.rowley.co.uk", "");

  // Done.
  return example_finish();
}
```

# Sending e-mail

### Send a mail...

As a more complex example of interacting with a server, here is an example of how to send e-mail using an open relay. You can send e-mail

### Select and build the project

In the examples for your board, you'll find a **Networking Projects** solution, and within that a **Send E-mail** project. Double-click that project to make it active and press **F7** to build.

### SMTP server

You need to configure the SMTP server for this example to work. In the example you will need to configure SMTP_SERVER with the domain name or dotted-decimal IP address of your SMTP server, and also set USER_EMAIL_ADDRESS to the e-mail address of the recipient.

### The code

There is nothing new in this example, it's just a little longer than retrieving a web page in the previous example.

```
/* Copyright (c) 2004-2013 Rowley Associates Limited.
*/

#include <string.h>
#include "example_support.h"
#include "libnet/ctl_net_api.h"
#include "libnet/extras/ctl_smtp_client.h"
#include "libplatform/platform.h"
#include "libplatform/platform_network.h"

// TODO: Example SMTP server address.  Replace with yours, either
// dotted-decimal or DNS name.
#define SMTP_SERVER \
  "your.mailserver.here"

// TODO: Example e-mail delivery address.  Replace with yours.
#define USER_EMAIL_ADDRESS  \
  "somebody@home.com"

// Resolved SMTP server.
static CTL_NET_IPv4_ADDR_t smtp_server_addr;

int
main(void)
{
  CTL_STATUS_t stat;
  char dot_ipaddr[16];

  // Initialize platform.
  platform_initialize();
  example_initialize();
```

```
  // Start networking, wait for an IP address.
  example_check_status(example_bring_up_full_networking());
  example_check_status(example_await_assigned_ip_address());

  // Wait 5s to see if we can resolve our mail server.  If you
  // use a dotted-decimal IPv4 address, there is no name lookup
  // and this completes immediately.
  printf("DNS: Resolving %s, maximum wait for DNS reply is 5 seconds.\n",
         SMTP_SERVER);
  example_check_status(ctl_dns_get_host_by_name(SMTP_SERVER,
                                                &smtp_server_addr,
                                                5000));
  printf("DNS: Resolved %s to %s\n",
        SMTP_SERVER,
        ctl_ip_sprint_addr(dot_ipaddr, smtp_server_addr));

  // Attempt to send some mail.
  stat = ctl_smtp_client_send_mail(smtp_server_addr,
                                   USER_EMAIL_ADDRESS,
                                   "crossworks@rowley.co.uk",  // fake
                                   "Hello from the CrossWorks TCP/IP Library!",
                                   0,
                                   "Hello!\n\nThis is the CrossWorks TCP/IP Library"
                                     "sending an e-mail to you.\n\n"
                                     "Regards,\n\n-- The CrossWorks Team.");

  // Say whether it worked.
  if (stat < CTL_NO_ERROR)
    example_terminate("SMTP: Didn't send that e-mail correctly.  Sorry.");
  else
    printf("SMTP: E-mail sent OK!\n");

  // Done.
  return example_finish();
}
```

# <ctl_net_api.h>

## Overview

TCP/IP Library public interface.

## API Summary

| | |
|---|---|
| **Network** | |
| CTL_NET_ERROR_t | Network Library errors |
| CTL_NET_PORT_t | A network port |
| ctl_net_get_host_name | Get host name |
| ctl_net_init | Initialize network library |
| ctl_net_interface | Network interface |
| ctl_net_set_host_name | Set host name |
| **Sockets** | |
| CTL_SOCKET_t | A TCP socket |
| CTL_TCP_ACCEPT_FN_t | Accept callback |
| CTL_TCP_GEN_ISS_FN_t | Initial send segment generation |
| CTL_TCP_GET_SOCKETS_FLAG_t | Flags for enumerating sockets |
| CTL_TCP_PORT_OPTIONS_t | TCP port options |
| CTL_TCP_SEND_FLAG_t | Socket send options |
| CTL_TCP_SOCKET_CLOSE_TYPE_t | Socket close options |
| CTL_TCP_SOCKET_CONNECTION_STATE_t | Logical socket state |
| CTL_TCP_SOCKET_OPTIONS_t | Socket options |
| ctl_soc_use_callback | Assign a server callback function on a per-socket basis |
| ctl_soc_use_event | Assign event group to socket |
| ctl_tcp_accept | Register an accept callback for a port |
| ctl_tcp_bind | Reserve TCP listener for TCP port |
| ctl_tcp_close_socket | Close a socket |
| ctl_tcp_connect | Connect a socket to port on a remote host |
| ctl_tcp_get_local_ip_addr | Get socket's local IP address |
| ctl_tcp_get_local_port | Get socket's local port |
| ctl_tcp_get_port_options | Get options for a TCP port |
| ctl_tcp_get_remote_ip_addr | Get socket's local IP address |
| ctl_tcp_get_remote_port | Get socket's remote port |

| | |
|---|---|
| ctl_tcp_get_socket_connection_state | Get socket state |
| ctl_tcp_get_socket_error | Get socket error |
| ctl_tcp_get_socket_options | Get socket options |
| ctl_tcp_get_sockets | Enumerate sockets for port |
| ctl_tcp_init | Initialize TCP layer |
| ctl_tcp_look_ahead | Look ahead in socket data |
| ctl_tcp_push | Push data on socket to network |
| ctl_tcp_read_line | Read a line of text from a socket |
| ctl_tcp_recv | Receive from socket |
| ctl_tcp_send | Send data to socket |
| ctl_tcp_set_port_options | Set options for a TCP port |
| ctl_tcp_set_socket_options | Set socket options |
| ctl_tcp_shutdown | Shut down a socket |
| ctl_tcp_socket | Fetch a TCP socket from the pool of unused sockets |
| ctl_tcp_unbind | Releases TCP listener resources for a TCP port |
| ctl_tcp_use_callback | Assign a server callback function to a bound TCP port |
| ctl_tcp_use_event | Assign a server event |
| **UDP** | |
| CTL_UDP_CONFIGURATION_t | UDP layer configuration parameters |
| CTL_UDP_INFO_t | UDP packet information |
| ctl_udp_bind | Register a UDP port callback |
| ctl_udp_init | Initialize UDP layer |
| ctl_udp_sendto | Send a UDP datagram |
| ctl_udp_unbind | Release a UDP port callback |
| **DHCP** | |
| ctl_dhcp_init | Initialize DHCP client subsystem |
| ctl_dhcp_lease_expire_time | Get lease renewal time |
| ctl_dhcp_lease_rebind_time | Get lease rebind time |
| ctl_dhcp_lease_renew_time | Get lease renewal time |
| ctl_net_domain_name_suffix | Return assigned domain name suffix |
| **DNS** | |
| ctl_dns_get_host_by_name | Look up a host name |
| ctl_dns_get_server | Get DNS server address |
| ctl_dns_init | Initialize DNS Client subsystem |
| ctl_dns_primary_server_addr | Get primary DNS server address |

| | |
|---|---|
| ctl_dns_print_cache | Display the DNS cache |
| ctl_dns_purge_cache | Purge the DNS cache |
| ctl_dns_secondary_server_addr | Get secondary DNS server address |
| ctl_dns_set_max_ttl | Set DNS time to live |
| ctl_dns_set_memory_allocator | Set DNS request memory allocator |
| ctl_dns_set_primary_server_addr | Set primary DNS server IP address |
| ctl_dns_set_secondary_server_addr | Set secondary DNS server IP address |
| ctl_dns_set_server | Set DNS server list entry |
| **NTP** | |
| ctl_ntp_init | Initialize NTP subsystem |
| ctl_ntp_server_addr | Return IPv4 address of NTP server |
| ctl_ntp_set_time_server | Set IPv4 address of NTP server |
| **ICMP** | |
| ctl_icmp_init | Initialize ICMP |
| **IP** | |
| CTL_IP_CONFIG_t | IP configuration structure |
| CTL_NET_IPv4_ADDR_t | IPv4 network address |
| CTL_NET_IPv4_LOCAL_BROADCAST_ADDR | IP local network broadcast address |
| ctl_net_get_gateway_address | Get gateway IP address |
| ctl_net_get_ip_address | Get system IP address |
| ctl_net_get_subnet_mask | Get system subnet mask |
| ctl_net_is_autoip_address | Is an IP address a link-local Auto-IP address? |
| ctl_net_is_local_broadcast_address | Is an IP address a local broadcast IP address? |
| ctl_net_is_local_ip_address | Is an IP address on the local subnet? |
| ctl_net_is_multicast_ip_address | Is an IP address a multicast IP address? |
| ctl_net_is_private_ip_address | Is an IP address a private address? |
| ctl_net_is_subnet_broadcast_address | Is an IP address a subnet broadcast IP address? |
| ctl_net_scan_dot_decimal_ip_addr | Scan a dotted-decimal IPv4 address |
| **ARP** | |
| ctl_arp_cache_entry | Replace ARP cache entry |
| ctl_arp_clear_entry | Clear ARP cache entry |
| ctl_arp_get_entry | Get ARP cache entry |
| ctl_arp_get_ttl | Get the ARP time to live |
| ctl_arp_print_cache | Display the ARP cache |
| ctl_arp_purge_cache | Purge ARP cache |

| | |
|---|---|
| ctl_arp_request_entry | Generate an ARP request for an IP address |
| ctl_arp_set_cache_size | Set maximum ARP cache size |
| ctl_arp_set_memory_allocator | Set ARP cache memory allocator |
| ctl_arp_set_ttl | Set ARP time to live |
| **MAC** | |
| CTL_NET_MAC_ADDR_t | Ethernet MAC address |
| ctl_eth_get_mac_addr | Return interface's MAC address |
| ctl_mac_addr_is_broadcast | Is this MAC address a broadcast address? |
| ctl_mac_addr_is_null_or_empty | Is this MAC address a null address? |
| ctl_net_scan_mac_addr | Scan a textual MAC address |
| **Memory** | |
| ctl_net_mem_alloc_data | Allocate network memory |
| ctl_net_mem_alloc_xmit | Allocate network memory |
| ctl_net_mem_free | Deallocate network memory |
| ctl_net_mem_trim | Trim allocated network memory |
| **Utility** | |
| ctl_ip_sprint_addr | Convert IPv4 address to dotted decimal string |
| ctl_mac_sprint_addr | Convert Ethernet MAC address to string |
| ctl_net_register_error_decoder | Register network error decoder |

# CTL_IP_CONFIG_t

## Synopsis

```
typedef struct {
  CTL_NET_IPv4_ADDR_t ip_addr;
  CTL_NET_IPv4_ADDR_t subnet_mask;
  CTL_NET_IPv4_ADDR_t gateway;
  CTL_NET_IPv4_ADDR_t dns_primary_server;
  CTL_NET_IPv4_ADDR_t dns_secondary_server;
  unsigned char ttl;
} CTL_IP_CONFIG_t;
```

## Description

**CTL_IP_CONFIG_t** contains the values needed to configure the IPv4 layer of the network library. If DHCP is not used, the application code must supply one of these structures to **ctl_net_init** during startup.

### ip_addr

Our IP address in network byte order.

### subnet_mask

Our subnet mask in network byte order.

### gateway

Local router (gateway) address in network byte order. This can be zero if packets never leave the LAN.

### dns_primary_server

Primary DNS server IP address in network byte order. This can be zero if DNS is not used.

### dns_secondary_server

Secondary DNS server IP address in network byte order. This can be zero if DNS is not used or there is no secondary DNS server.

### ttl

Time to live for outgoing IP packets. Generally a ‘don't care’ for use on a LAN.

## See Also

ctl_net_init

# CTL_NET_ERROR_t

## Synopsis

```
typedef enum {
  CTL_NET_CONFIGURATION_ERROR,
  CTL_NET_NOT_UP,
  CTL_NET_UNREACHABLE,
  CTL_DNS_HOST_NAME_ERROR,
  CTL_DNS_RESOLVE_IN_PROGRESS,
  CTL_DNS_OUT_OF_MEMORY,
  CTL_DNS_NAME_UNKNOWN,
  CTL_DNS_NO_DNS_SERVER,
  CTL_UDP_TOO_MANY_PORTS,
  CTL_UDP_PORT_IN_USE,
  CTL_TCP_PORT_ACTIVE,
  CTL_UDP_BAD_PORT,
  CTL_TCP_TOO_MANY_PORTS,
  CTL_TCP_BAD_PORT,
  CTL_TCP_PORT_NOT_BOUND,
  CTL_TCP_PORT_IN_USE,
  CTL_TCP_BAD_SOCKET,
  CTL_TCP_TOO_MANY_OPEN_SOCKETS,
  CTL_TCP_SOCKET_CLOSED,
  CTL_NET_ERR_WOULD_BLOCK,
  CTL_NET_ERR_ALREADY,
  CTL_NET_ERR_NOTSOCK,
  CTL_NET_ERR_OPNOTSUPP,
  CTL_NET_ERR_NETDOWN,
  CTL_NET_ERR_NETUNREACH,
  CTL_NET_ERR_CONNABORTED,
  CTL_NET_ERR_CONNRESET,
  CTL_NET_ERR_NOTCONN,
  CTL_NET_ERR_TIMEDOUT,
  CTL_NET_ERR_CONNREFUSED,
  CTL_NET_ERR_HOSTUNREACH,
  CTL_NET_ERR_NOTEMPTY,
  CTL_NET_ERR_DISCON
} CTL_NET_ERROR_t;
```

## Description

**CTL_NET_ERROR_t** enumerates the errors that the TCP/IP Library generates.

### CTL_NET_CONFIGURATION_ERROR

Indicates that the network library is not configured correctly. As delivered, the CTL network library is correctly configured and tested, so this error should not be seen by users. If you do see this error, please check your configuration.

### CTL_NET_NOT_UP

Indicates that a call to **ctl_dns_get_host_by_name** timed out without the network stack coming up.

**CTL_NET_UNREACHABLE**

Indicates that a network packet needs to be delivered to an IP address that has no route. This can happen, for instance, when the packet has a non-local IP address which must be delivered to the gateway and no gateway has been configured either statically or by DHCP.

**CTL_DNS_HOST_NAME_ERROR**

Indicates that a host name is invalid, for instance it has a trailing period (`foo.bar.` is invalid), or the host name has more than 47 characters.

**CTL_DNS_RESOLVE_IN_PROGRESS**

Indicates that the requested host name is already being resolved. Typically, this status is returned by **ctl_dns_get_host_by_name** when a non-zero timeout is specified and the name did not resolve before the timeout.

**CTL_DNS_OUT_OF_MEMORY**

Indicates that the DNS resolver could not allocate memory using the DNS memory allocator when queuing a DNS request.

**CTL_DNS_NAME_UNKNOWN**

Indicates that the DNS resolver could not resolve the host name.

**CTL_DNS_NO_DNS_SERVER**

Indicates that no DNS server has been defined in order to resolve requests.

**CTL_UDP_TOO_MANY_PORTS**

Indicates that all UDP ports are bound and no unused port exists when using **ctl_udp_bind**.

**CTL_UDP_PORT_IN_USE**

Indicates that the client tried to bind a port using **ctl_udp_bind** but that port has already been bound.

**CTL_UDP_BAD_PORT**

Indicates that the port passed to **ctl_udp_unbind** is invalid or is not currently bound.

**CTL_TCP_TOO_MANY_PORTS**

Indicates that you have requested to bind a TCP ports using **ctl_tcp_bind** but there are no TCP ports left in the TCP port pool. You will need to increase the number of ports when calling **ctl_tcp_init** to initialize the TCP subsystem.

**CTL_TCP_PORT_ACTIVE**

Indicates that there are active, open sockets associated with a port when the port is unbound with **ctl_tcp_unbind**.

**CTL_TCP_BAD_PORT**

Indicates that an invalid TCP port has been provided as a parameter. Port numbers in API calls must be in network byte order and must specify a valid TCP port number, usually between 1 and 65535.

**CTL_TCP_PORT_NOT_BOUND**

Indicates that an unbound port parameter has been passed to an API call that requires a bound TCP port. Many API calls require bound ports, and you try to operate on a port that has not been bound using **ctl_tcp_bind** you will receive this error.

**CTL_TCP_PORT_IN_USE**

Indicates that a call to **ctl_tcp_bind** failed because the port provided is already being listened to. In order to specify a different listener for a port, the port must be first be unbound using **ctl_tcp_unbind**.

**CTL_TCP_BAD_SOCKET**

Indicates that the socket provided to a network API call is invalid because the socket has been closed (either by the client or by the network library), or has never been open.

**CTL_TCP_TOO_MANY_SOCKETS**

Indicates that an API call could not allocate a socket using **ctl_tcp_socket**. The number of sockets that the application can open is determined by the number of streams that the CTL library supports—one socket requires one stream, and other components, such as the mass storage library, will consume shared streams when you use them.

**CTL_TCP_SOCKET_CLOSED**

Indicates that the other TCP closed the socket whilst the client was waiting for data from the socket. In some cases the network library will return **CTL_TCP_BAD_SOCKET** for the same conditions if, on entry to the API call, the socket is already closed.

For socket-related errors, see **ctl_tcp_get_socket_error**.

# CTL_NET_IPv4_ADDR_t

## Synopsis

```
typedef unsigned long CTL_NET_IPv4_ADDR_t;
```

## Description

**CTL_NET_IPv4_ADDR_t** contains a 4-octet IPv4 address held in in network byte order.

# CTL_NET_IPv4_LOCAL_BROADCAST_ADDR

## Synopsis

```
#define CTL_NET_IPv4_LOCAL_BROADCAST_ADDR   0xFFFFFFFF
```

## Description

**CTL_NET_IPv4_LOCAL_BROADCAST_ADDR** is the IP broadcast address 255.255.255.255. It is the broadcast address of the zero network (0.0.0.0/0), which in IP standards stands for *this* network, i.e. the local network. Transmission to this address is limited *by definition*—it is never forwarded by routers that connect the local network to the Internet.

# CTL_NET_MAC_ADDR_t

## Synopsis

```
typedef struct {
  unsigned char octet[];
} CTL_NET_MAC_ADDR_t;
```

## Description

**CTL_NET_MAC_ADDR_t** points to an object that contains the Ethernet MAC address in network byte order.

CrossWorks TCP/IP Library

# CTL_NET_PORT_t

## Synopsis

```
typedef unsigned short CTL_NET_PORT_t;
```

## Description

**CTL_NET_PORT_t** is a network port.

## Note

Ports are *always* specified in network byte order.

# CTL_SOCKET_t

## Synopsis

```
typedef CTL_STREAM_t CTL_SOCKET_t;
```

## Description

**CTL_SOCKET_t** is the type for TCP sockets. You can treat the TCP socket as a simple stream of bytes and read from and write to the socket using standard CTL stream functions.

# CTL_TCP_ACCEPT_FN_t

## Synopsis

```
typedef unsigned (*CTL_TCP_ACCEPT_FN_t)(CTL_SOCKET_t);
```

## Description

The Accept callback performs two functions:

- Decide whether or not the network library will accept an incoming connection request.
- Setup the "process socket" callback or CTL task trigger event, i.e. call **ctl_tcp_use_callback** or **ctl_tcp_use_event**.

When a SYN (synchronize, or "connect") packet arrives for a bound port, a check is first made to determine if there is a free socket and that the number of open sockets for the port is less than the **max_connections** value for that port.

If that check is passes, a socket is allocated and the port's accept callback is invoked, to make the final pass/fail judgment.

For example:

```
unsigned tcpAcceptCallbackFn(SOCKET s)
{
  // SOCKET s is not yet readable or writable, but does have
  // valid endpoint information.  You may choose to accept or
  // reject the connection based upon the remote TCP's IP
  // address, for example.

  // If the connection is accepted, ctl_tcp_use_callback() or
  // ctl_tcp_use_event() should be called to set up processing
  // of the TCP data.

  // Now is the time to adjust per-socket memory limits using
  // ctl_tcp_set_socket_options(), before the response is made
  // to the remote TCP's synchroization packet.

  if (we accept connection)
    return 1;
  else
    return 0;
}
```

# CTL_TCP_GEN_ISS_FN_t

## Synopsis

```
typedef unsigned long (*CTL_TCP_GEN_ISS_FN_t)(void);
```

## Description

**CTL_TCP_GEN_ISS_FN_t** describes a callback function to generate TCP *initial send segment* numbers. The application must supply an instance of this which must generate unpredictable numbers.

# CTL_TCP_GET_SOCKETS_FLAG_t

## Synopsis

```
typedef enum {
  CTL_TCP_GET_SOCKETS_CONNECTING,
  CTL_TCP_GET_SOCKETS_CONNECTED,
  CTL_TCP_GET_SOCKETS_READABLE,
  CTL_TCP_GET_SOCKETS_TRIGGERED,
  CTL_TCP_GET_SOCKETS_CLOSED
} CTL_TCP_GET_SOCKETS_FLAG_t;
```

## Description

**CTL_TCP_GET_SOCKETS_FLAG_t** defines a set of flags for enumerating sockets using **ctl_tcp_get_sockets**.

### CTL_TCP_GET_SOCKETS_CONNECTING

Enumerate sockets that have not completed the synchronization handshake.

### CTL_TCP_GET_SOCKETS_CONNECTED

Enumerate sockets with an established connection.

### CTL_TCP_GET_SOCKETS_READABLE

Sockets with an established connection that also have queued bytes available.

### CTL_TCP_GET_SOCKETS_TRIGGERED

Sockets with an established connection that have a "push" packet in the receive queue and all sent "push" packets have been acknowledged by the remote.

### CTL_TCP_GET_SOCKETS_CLOSED

Sockets that are to be reclaimed soon, typically within 100 milliseconds.

## See Also

ctl_tcp_get_sockets

# CTL_TCP_PORT_OPTIONS_t

## Synopsis

```
typedef struct {
  unsigned max_connections;
  CTL_TCP_SOCKET_OPTIONS_t defaults;
} CTL_TCP_PORT_OPTIONS_t;
```

**CTL_TCP_PORT_OPTIONS_t** contains settings for server sockets, on a port-by-port basis.

### `max_connections`

Maximum number of sockets that can be 'owned' by the server.

### `defaults`

Default options for a socket created by the server. When a new TCP connect request is received for the port registered to the server, a socket is created and its options are initialized with these values before the 'accept' callback is invoked.

## See Also

CTL_TCP_SOCKET_OPTIONS_t, ctl_tcp_get_port_options, ctl_tcp_set_port_options, ctl_tcp_get_socket_options, ctl_tcp_set_socket_options

# CTL_TCP_SEND_FLAG_t

## Synopsis

```
typedef enum {
  CTL_TCP_SEND_PUSH,
  CTL_TCP_SEND_URGENT,
  CTL_TCP_SEND_NOCOPY,
  CTL_TCP_SEND_FREE
} CTL_TCP_SEND_FLAG_t;
```

See ctl_tcp_send for a full description of the flags.

### CTL_TCP_SEND_PUSH

Push buffered data to network.

### CTL_TCP_SEND_URGENT

Send out-of-band data. This is not implemented.

### CTL_TCP_SEND_NOCOPY

Perform a zero-copy send of static data.

### CTL_TCP_SEND_FREE

Perform a zero-copy send of dynamic data.

# CTL_TCP_SOCKET_CLOSE_TYPE_t

## Synopsis

```
typedef enum {
  CTL_TCP_CLOSE_LINGER,
  CTL_TCP_CLOSE_DONTLINGER
} CTL_TCP_SOCKET_CLOSE_TYPE_t;
```

## Description

**CTL_TCP_SOCKET_CLOSE_TYPE_t** indicates how a socket should be closed.

| linger | timeout | Type of close | Wait for close? |
|---|---|---|---|
| CTL_TCP_CLOSE_DONTLIN( | Don't care | Graceful | No |
| CTL_TCP_CLOSE_LINGER | Zero | Hard | No |
| CTL_TCP_CLOSE_LINGER | Nonzero | Graceful | Yes |

## See Also

ctl_tcp_close_socket

# CTL_TCP_SOCKET_CONNECTION_STATE_t

## Synopsis

```
typedef enum {
  CTL_TCP_SOCKET_STATE_CLOSED,
  CTL_TCP_SOCKET_STATE_CONNECTING,
  CTL_TCP_SOCKET_STATE_CONNECTED,
  CTL_TCP_SOCKET_STATE_CLOSING
} CTL_TCP_SOCKET_CONNECTION_STATE_t;
```

## Description

**CTL_TCP_SOCKET_CONNECTION_STATE_t** is a condensed version of the complete set of states defined by RFC793. Whilst this should be self-explanatory we document the states anyway:

### CTL_TCP_SOCKET_STATE_CLOSED

Socket has never been open, is invalid, or has been closed.

### CTL_TCP_SOCKET_STATE_CONNECTING

Socket is connecting.

### CTL_TCP_SOCKET_STATE_CONNECTED

Socket has completed three-way handshake and is ready for business.

### CTL_TCP_SOCKET_STATE_CLOSING

Socket is closing.

# CTL_TCP_SOCKET_OPTIONS_t

## Synopsis

```
typedef struct {
  size_t max_receive_segment_size;
  size_t max_owned_receive_bytes;
  size_t max_send_segment_size;
  size_t max_owned_send_bytes;
  unsigned long idle_socket_shutdown;
  char autoPush;
} CTL_TCP_SOCKET_OPTIONS_t;
```

**CTL_TCP_SOCKET_OPTIONS_t** contains configuration information for a socket.

In lieu of the classic sockets **getsockopt** and **setsockopt** functions, the TCP layer presents and receives its options in a single structure.

A client socket should set these options before calling **ctl_tcp_connect**.

A server socket's only chance at legally manipulating this its options would be during the `CTL_TCP_ACCEPT_FN_t` callback, but all sockets for a given port are initialized with the `CTL_TCP_SOCKET_OPTIONS_t` contained in the `CTL_TCP_PORT_OPTIONS_t` for that port. In general, calling `ctl_tcp_set_socket_options` for an individual server socket is not required.

The structure has the following members:

### `max_receive_segment_size`

Maximum size of a receive segment. This cannot be greater than 1460 for Ethernet transports.

### `max_owned_receive_bytes`

Used to calculate the *receive window* and slow down the remote TCP, if required. For maximum efficiency, it should be a multiple of **max_receive_segment_size**.

### `max_send_segment_size`

Maximum size of a sense segment. This cannot be greater than 1460 for Ethernet transports. When sending a segment for this socket, the network library will allocate the minimum of this value and what the remote advertises during the connect handshake.

### `max_owned_send_bytes`

Used to slow down application code, if required. This value does not include big external buffers that are passed during blocking **ctl_tcp_send**. For maximum efficiency, this should be a multiple of the **max_send_segment_size**.

### `idle_socket_shutdown`

In whole seconds. Set this to zero if an idle socket should be kept alive forever. Otherwise, when a socket is idle for longer than this value, the network library will gracefully close the socket and recover its resources by initiating a FIN handshake with the remote TCP.

### Note

This structure should be set *prior* to a connection being established with a remote TCP. For a client socket, it means that the application layer may only set a socket's options between the calls to **ctl_tcp_socket** and **ctl_tcp_connect**. For a server socket, it means that the only place to modify the socket options is within the `CTL_TCP_ACCEPT_FN_t` callback function.

### See Also

**ctl_tcp_get_socket_options**, **ctl_tcp_set_socket_options**, **ctl_tcp_connect**

# CTL_UDP_CONFIGURATION_t

## Synopsis

```
typedef struct {
  CTL_NET_PORT_t min_ephemeral_port;
  CTL_NET_PORT_t max_ephemeral_port;
  int max_bound_ports;
} CTL_UDP_CONFIGURATION_t;
```

## Description

**CTL_UDP_CONFIGURATION_t** contains the initialization parameters for the UDP layer. Please refer to **ctl_udp_init** for a description of these members.

## See Also

ctl_udp_init

# CTL_UDP_INFO_t

## Synopsis

```
typedef struct {
  CTL_NET_PORT_t this_port;
  CTL_NET_PORT_t other_port;
  CTL_NET_IPv4_ADDR_t other_ip_addr;
  void *metadata;
} CTL_UDP_INFO_t;
```

## Description

pointer to an instance of **CTL_UDP_INFO_t** is passed into user code during a UDP receive callback and out of user code when calling **ctl_udp_sendto**.

Note the use of 'this' and 'other' semantics rather than 'src' and 'dst'.

In a UDP server, the same **CTL_UDP_INFO_t** pointer received in the **CTL_UDP_RECV_FN_t** may be passed unmodified to **ctl_udp_sendto** as in the following example: The simple semantic change of using "this" and "other" avoids having to do a parameter swap in the callback.

```
void myUdpReceiveFn(unsigned long *rcvData,
                    unsigned rcvByteCount,
                    const CTL_UDP_INFO_t *info)
{
  // \em{process the rcvData...}
  // \em{and then call ctl_udp_sendto}
  ctl_udp_sendto(sendData, sendByteCount, info, 0);
}
```

You can use the **metaData** member to store endpoint information for any application-specific data set by the MAC-layer driver.

## Note

The **metaData** member is intended to be used by IEEE 1588 (Precision Time Protocol)-compliant MAC layers to provide a packet timestamp (or at least a pointer to a packet timestamp), but the field may be used for any information that needs to be transmitted from the MAC layer to the application layer as part of a UDP datagram.

## See Also

CTL_UDP_RECV_FN_t, ctl_udp_sendto

# ctl_arp_cache_entry

## Synopsis

```
void ctl_arp_cache_entry(CTL_NET_IPv4_ADDR_t ip_addr,
                         const CTL_NET_MAC_ADDR_t *mac_addr);
```

## Description

**ctl_arp_cache_entry** updates the ARP cache to associate the IP address **ip_addr** with the MAC address **mac_addr**. Broadcast MAC addresses are rejected and not entered into the cache.

You would not usually need to call **ctl_arp_cache_entry** as ARP management is handled transparently by the network library.

## Thread Safety

**ctl_arp_cache_entry** is thread-safe.

## See Also

ctl_arp_clear_entry, ctl_arp_purge_cache

# ctl_arp_clear_entry

## Synopsis

```
void ctl_arp_clear_entry(CTL_NET_IPv4_ADDR_t ip_addr);
```

## Description

**ctl_arp_clear_entry** removes the entry for the IP address **ip_addr** in the ARP cache.

## Thread Safety

**ctl_arp_clear_entry** is thread-safe.

## See Also

ctl_arp_purge_cache

# ctl_arp_get_entry

## Synopsis

```
unsigned ctl_arp_get_entry(CTL_NET_MAC_ADDR_t *dst,
                           CTL_NET_IPv4_ADDR_t ip_addr);
```

## Description

**ctl_arp_get_entry** queries the ARP cache for the MAC address corresponding to the IP address **ip_addr**.

If the IP address is found in the ARP cache, the found MAC address is copied into the MAC address pointed to by **mac_addr** and a non-zero result is returned.

If the IP address is not found in the ARP cache, the MAC address pointed to by **mac_addr** is zeroed and **ctl_arp_get_entry** returns zero.

**mac_addr** can be null to query the presence of an IP-to-MAC mapping without returning the MAC address of the entry.

## Note

**ctl_arp_get_entry** only queries the cache and does not send an ARP request top the network if the IP address is not found in the cache.

# ctl_arp_get_ttl

## Synopsis

```
unsigned long ctl_arp_get_ttl(void);
```

## Description

**ctl_arp_get_ttl** returns the currently-set time-to-live for entries in the ARP cache. The default time to live is 10 minutes.

## Thread Safety

**ctl_arp_get_ttl** is thread-safe.

## See Also

ctl_arp_set_ttl

# ctl_arp_print_cache

## Synopsis

```
void ctl_arp_print_cache(CTL_STREAM_t s);
```

## Description

**ctl_arp_print_cache** displays the contents of the ARP cache to the stream **s**.

# ctl_arp_purge_cache

## Synopsis

```
void ctl_arp_purge_cache(void);
```

## Description

**ctl_arp_purge_cache** clears the entire ARP cache.

## Thread Safety

**ctl_arp_purge_cache** is thread-safe.

## See Also

ctl_arp_clear_entry

# ctl_arp_request_entry

## Synopsis

```
void ctl_arp_request_entry(CTL_NET_IPv4_ADDR_t ip_addr);
```

## Description

**ctl_arp_request_entry** generates an ARP request for the MAC address corresponding to the IP address **ip_addr**.

# ctl_arp_set_cache_size

## Synopsis

```
void ctl_arp_set_cache_size(unsigned size);
```

## Description

**ctl_arp_set_cache_size** sets the maximum number of entries held in the ARP cache to **size**.
**ctl_arp_set_cache_size** restricts the range of **size** to be between 4 and 256 entries.

**ctl_arp_set_cache_size** does not clear the ARP cache when it is resized, but if the cache is contracted, entries in the cache cache are discarded in reverse age order, from oldest to youngest.

## Thread Safety

**ctl_arp_set_cache_size** is thread-safe.

# ctl_arp_set_memory_allocator

## Synopsis

```
void ctl_arp_set_memory_allocator(CTL_MEMORY_ALLOCATOR_t *allocator);
```

## Description

**ctl_arp_set_memory_allocator** sets ARP memory allocator to **allocator**. If **allocator** is zero, the ARP cache uses the system memory allocator **ctl_system_memory_allocator**.

## Thread Safety

**ctl_arp_set_memory_allocator** is thread-safe.

# ctl_arp_set_ttl

## Synopsis

```
void ctl_arp_set_ttl(unsigned long ttl);
```

## Description

**ctl_arp_set_ttl** sets the timeout before an entry is deleted from the ARP cache to **ttl** seconds. The default time to live is 10 minutes.

## Thread Safety

**ctl_arp_set_ttl** is thread-safe.

## See Also

ctl_arp_get_ttl

CrossWorks TCP/IP Library

# ctl_dhcp_init

## Synopsis

```
CTL_STATUS_t ctl_dhcp_init(void);
```

## Description

**ctl_dhcp_init** initializes the DHCP client subsystem and registers it with the IP layer. DHCP counts as one of your bound UDP ports.

## See Also

ctl_net_init, ctl_udp_init

# ctl_dhcp_lease_expire_time

## Synopsis

```
CTL_TIME_t ctl_dhcp_lease_expire_time(void);
```

## Description

**ctl_dhcp_lease_expire_time** returns the time that the DHCP lease expires.

## Note

This is provided as a convenience so the application can print DHCP information; the DHCP client code in the Network Library manages all aspects of the IP lease.

## See Also

ctl_dhcp_lease_renew_time, ctl_dhcp_lease_rebind_time

# ctl_dhcp_lease_rebind_time

## Synopsis

```
CTL_TIME_t ctl_dhcp_lease_rebind_time(void);
```

## Description

**ctl_dhcp_lease_rebind_time** returns the time that the DHCP client will attempt a rebind as the lease has not been renewed by a DHCP server.

## Note

This is provided as a convenience so the application can print DHCP information; the DHCP client code in the Network Library manages all aspects of the IP lease.

## See Also

ctl_dhcp_lease_renew_time, ctl_dhcp_lease_expire_time

# ctl_dhcp_lease_renew_time

## Synopsis

```
CTL_TIME_t ctl_dhcp_lease_renew_time(void);
```

## Description

**ctl_dhcp_lease_renew_time** returns the time that the DHCP client initiates renewal to extend the lease of the assigned IP address.

## Note

This is provided as a convenience so the application can print DHCP information; the DHCP client code in the Network Library manages all aspects of the IP lease.

## See Also

ctl_dhcp_lease_rebind_time, ctl_dhcp_lease_expire_time

# ctl_dns_get_host_by_name

## Synopsis

```
CTL_STATUS_t ctl_dns_get_host_by_name(const char *hostname,
                                      CTL_NET_IPv4_ADDR_t *addr,
                                      CTL_TIME_t timeout);
```

## Description

**ctl_dns_get_host_by_name** writes the IP address of the host **hostname** into the address pointed to by **ip_addr**. If **ms** is zero this is a non-blocking lookup otherwise it is a blocking lookup.

The host name is validated and, if invalid, **ctl_dns_get_host_by_name** returns **CTL_DNS_HOST_NAME_ERROR**. If the network is not yet up (for instance, the network library has not received an IP address from a static configuration or by DHCP), **ctl_dns_get_host_by_name** returns **CTL_NET_NOT_UP**.

If the host address is in the DNS cache maintained by the network library, the address is written to **ip_addr** immediately and **ctl_dns_get_host_by_name** returns **CTL_NO_ERROR**.

If the host address is not in the DNS cache, the network library queues a DNS lookup. If this is a non-blocking call (i.e. **ms** is zero) then **ctl_dns_get_host_by_name** immediately returns the non-fatal status **CTL_DNS_RESOLVE_IN_PROGRESS**.

If this is a blocking call, **ctl_dns_get_host_by_name** waits for a response. If no response is received from a DNS server within **timeout** milliseconds, or all DNS servers are queried and time out, **ctl_dns_get_host_by_name** returns **CTL_DNS_NAME_UNKNOWN**.

## Return Value

**ctl_dns_get_host_by_name** returns a standard status code.

## Thread Safety

**ctl_dns_get_host_by_name** is thread-safe.

# ctl_dns_get_server

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_dns_get_server(unsigned index);
```

## Description

**ctl_dns_get_server** returns the IP address of the DNS server with index **index**. If **index** is invalid, **ctl_dns_get_server** returns an all-zero IP address.

If IP addresses are assigned by DHCP, **ctl_dns_get_server** returns all-zero IP address whilst IP negotiation is in progress.

# ctl_dns_init

## Synopsis

```
CTL_STATUS_t ctl_dns_init(void);
```

## Description

**ctl_dns_init** initializes the DNS client subsystem and registers it with the IP layer. DNS counts as one of your bound UDP ports.

# ctl_dns_primary_server_addr

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_dns_primary_server_addr(void);
```

## Description

**ctl_dns_primary_server_addr** returns the primary DNS server as set in the **CTL_IP_CONFIG_t** passed to **ctl_net_init** or or retrieved from a DHCP server.

If IP addresses are assigned by DHCP, **ctl_dns_primary_server_addr** will returns an all-zero IP address whilst IP negotiation is in progress.

## See Also

CTL_IP_CONFIG_t, ctl_net_init

# ctl_dns_print_cache

## Synopsis

```
void ctl_dns_print_cache(CTL_STREAM_t s);
```

## Description

**ctl_dns_print_cache** prints the contents of the DNS cache to the stream **s**.

## Thread Safety

**ctl_dns_print_cache** is thread-safe if writing to stream **s** is thread-safe.

# ctl_dns_purge_cache

## Synopsis

```
void ctl_dns_purge_cache(void);
```

## Description

**ctl_dns_purge_cache** purges the DNS cache throwing away all cache entries and canceling all outstanding resolves.

## Thread Safety

**ctl_dns_purge_cache** is thread-safe.

# ctl_dns_secondary_server_addr

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_dns_secondary_server_addr(void);
```

## Description

**ctl_dns_secondary_server_addr** returns the secondary DNS server as set in the **CTL_IP_CONFIG_t** passed to **ctl_net_init** or or retrieved from a DHCP server.

If IP addresses are assigned by DHCP, **ctl_dns_secondary_server_addr** will returns an all-zero IP address whilst IP negotiation is in progress.

## See Also

CTL_IP_CONFIG_t, ctl_net_init

# ctl_dns_set_max_ttl

## Synopsis

```
void ctl_dns_set_max_ttl(unsigned long ttl);
```

## Description

**ctl_dns_set_max_ttl** sets the maximum timeout before an entry is deleted from the DNS cache to **ttl** seconds. The default time to live is 24 hours.

The DNS cache entry for a DNS record is set to the earliest of the time to live set by **ctl_dns_set_max_ttl** and the time to live returned by the server.

## Thread Safety

**ctl_dns_set_max_ttl** is thread-safe.

# ctl_dns_set_memory_allocator

## Synopsis

```
void ctl_dns_set_memory_allocator(CTL_MEMORY_ALLOCATOR_t *allocator);
```

## Description

**ctl_dns_set_memory_allocator** sets DNS memory allocator to **allocator**. If **allocator** is zero, the DNS cache uses the system memory allocator **ctl_system_memory_allocator**.

## Note

Setting the memory allocator automatically clears the DNS cache and cancels any outstanding DNS resolves. We recommend that you set the DNS allocator before starting the DNS revolver.

## Thread Safety

**ctl_dns_set_memory_allocator** is thread-safe.

# ctl_dns_set_primary_server_addr

## Synopsis

```
void ctl_dns_set_primary_server_addr(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_dns_set_primary_server_addr** sets the primary DNS server IP address to **addr**.

## Note

Other parts of the network library may overwrite the address set by this function, for instance when DHCP negotiation is complete.

**ctl_dns_set_primary_server_addr** and **ctl_dns_set_secondary_server_addr** are decoupled from the rest of the DNS resolver so that you can use DHCP with assigned DNS server addresses set without automatically pulling in the resolver code.

# ctl_dns_set_secondary_server_addr

## Synopsis

```
void ctl_dns_set_secondary_server_addr(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_dns_set_secondary_server_addr** sets the secondary DNS server IP address to **addr**.

## Note

Other parts of the network library may overwrite the address set by this function, for instance when DHCP negotiation is complete.

**ctl_dns_set_primary_server_addr** and **ctl_dns_set_secondary_server_addr** are decoupled from the rest of the DNS resolver so that you can use DHCP with assigned DNS server addresses set without automatically pulling in the resolver code.

# ctl_dns_set_server

## Synopsis

```
void ctl_dns_set_server(int index,
                        CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_dns_set_server** sets index entry **index** of the DNS server list to **addr**. Index 0 is the primary DNS server, 1 is the secondary server, and so on.

# ctl_eth_get_mac_addr

## Synopsis

```
CTL_NET_MAC_ADDR_t *ctl_eth_get_mac_addr(void);
```

**ctl_eth_get_mac_addr** returns the MAC address set when registering the MAC driver using **ctl_net_set_mac_driver**.

## See Also

ctl_net_set_mac_driver

# ctl_icmp_init

## Synopsis

```
CTL_STATUS_t ctl_icmp_init(void);
```

## Description

**ctl_icmp_init** initializes the ICMP subsystem. Only the Echo Request (ping) ICMP type code is supported by the network library, all other type codes fail silently.

**ctl_icmp_init** returns **CTL_NO_ERROR** on success; i.e. the ICMP subsystem is registered with the IP layer.

# ctl_ip_sprint_addr

## Synopsis

```
char *ctl_ip_sprint_addr(char *dst,
                         CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_ip_sprint_addr** converts the address **addr** to dotted decimal notation and writes the result to the object pointed to by **dst**. **dst** must be 16 characters or more for three dotted decimal octets plus a terminating zero.

## Return Value

**ctl_ip_sprint_addr** returns **dst**.

# ctl_mac_addr_is_broadcast

## Description

**ctl_mac_addr_is_broadcast** returns true if the address **addr** is a broadcast address. A MAC address with every bit set to one is a broadcast address, i.e. address FF:FF:FF:FF:FF:FF.

## Thread Safety

**ctl_mac_addr_is_broadcast** is thread-safe.

# ctl_mac_addr_is_null_or_empty

## Description

**ctl_mac_addr_is_null_or_empty** returns true if the address **addr** is null or the address pointed to is an all-zero address. A MAC address with every bit set to zero is a null address, i.e. address 00:00:00:00:00:00.

## Thread Safety

**ctl_mac_addr_is_null_or_empty** is thread-safe.

# ctl_mac_sprint_addr

## Synopsis

```
char *ctl_mac_sprint_addr(char *dst,
                          const CTL_NET_MAC_ADDR_t *addr,
                          char sep);
```

## Description

**ctl_mac_sprint_addr** converts the address **addr** to hexadecimal notation, using **sep** to separate each octet, and writes the result to the object pointed to by **dst**. **dst** must be 18 characters or more for six hexadecimal octets, separators, and a terminating zero.

## Return Value

**ctl_mac_sprint_addr** returns **dst**.

# ctl_net_domain_name_suffix

## Synopsis

```
char *ctl_net_domain_name_suffix(void);
```

## Description

**ctl_net_domain_name_suffix** returns domain name suffix provided by the DHCP server when an IP address is assigned. If no domain name suffix is set by the DHCP server, or no address has been assigned by the DHCP server, **ctl_net_domain_name_suffix** returns zero.

## Thread Safety

**ctl_net_domain_name_suffix** is thread-safe.

# ctl_net_get_gateway_address

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_net_get_gateway_address(void);
```

## Description

**ctl_net_get_gateway_address** returns returns the gateway (local router's) IP address as set in the **CTL_IP_CONFIG_t** configuration passed to **ctl_net_init** or retrieved from a DHCP server.

If IP addresses are assigned by DHCP, **ctl_net_get_gateway_address** will returns an all-zero IP address whilst IP negotiation is in progress.

## Thread Safety

**ctl_net_get_gateway_address** is thread-safe.

# ctl_net_get_host_name

## Synopsis

```
char *ctl_net_get_host_name(void);
```

## Description

**ctl_net_get_host_name** returns a pointer to a null-terminated read-only string that contains the host name set by **ctl_net_set_host_name**. If no host name has been set, the host name is empty.

## See Also

ctl_net_get_host_name

# ctl_net_get_ip_address

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_net_get_ip_address(void);
```

## Description

**ctl_net_get_ip_address** returns the system's IP address as set in the **CTL_IP_CONFIG_t** configuration passed to **ctl_net_init** or retrieved from a DHCP server.

If IP addresses are assigned by DHCP, **ctl_net_get_ip_address** will return an all-zero IP address whilst IP negotiation is in progress.

## Thread Safety

**ctl_net_get_ip_address** is thread-safe.

# ctl_net_get_subnet_mask

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_net_get_subnet_mask(void);
```

## Description

**ctl_net_get_subnet_mask** returns the system's subnet mask as set in the **CTL_IP_CONFIG_t** configuration passed to **ctl_net_init** or retrieved from a DHCP server.

If IP addresses are assigned by DHCP, **ctl_net_get_subnet_mask** will returns an all-zero IP address whilst IP negotiation is in progress.

## Thread Safety

**ctl_net_get_subnet_mask** is thread-safe.

# ctl_net_init

## Synopsis

```
CTL_STATUS_t ctl_net_init(unsigned taskPriority,
                          const CTL_IP_CONFIG_t *ipInit);
```

**ctl_net_init** initializes the network library core, which consists of the IP and ARP layers. The network task is created using a task priority **priority**. The initial IP configuration is pointed to by and this may be null if DHCP is used to configure the host settings.

## See Also

CTL_IP_CONFIG_t

# ctl_net_interface

## Synopsis

```
CTL_NET_INTERFACE_t *ctl_net_interface;
```

## Description

**ctl_net_interface** holds a pointer to the network interface initialized by **ctl_mac_init**. If **ctl_net_interface** is zero, the MAC has not been initialized.

The TCP/IP library supports a single MAC at this time.

## See Also

ctl_mac_init.

# ctl_net_is_autoip_address

## Synopsis

```
unsigned ctl_net_is_autoip_address(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_net_is_autoip_address** determines whether **addr** is a IPv4 link-local Auto-IP address on the local subnet. **ctl_net_is_autoip_address** returns non-zero if **addr** is an Auto-IP address on the local subnet and zero if not.

Auto-IP addresses are defined by RFC 3927 to be the range 169.254.0.0—169.254.255.255 (169.254/16 prefix) with subnet mask 255.255.0.0.

## Thread Safety

**ctl_net_is_autoip_address** is thread-safe.

# ctl_net_is_local_broadcast_address

## Synopsis

```
unsigned ctl_net_is_local_broadcast_address(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_net_is_local_broadcast_address** determines whether **addr** is an local subnet broadcast address, that is the address **addr** is either the limited subnet broadcast address 255.255.255.255 or the subnet broadcast address.

## Thread Safety

**ctl_net_is_local_broadcast_address** is thread-safe.

## See Also

ctl_net_is_subnet_broadcast_address

# ctl_net_is_local_ip_address

## Synopsis

```
unsigned ctl_net_is_local_ip_address(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_net_is_local_ip_address** determines whether **addr** is an IP address on the local subnet.
**ctl_net_is_local_ip_address** returns non-zero if **addr** is known to be on the local subnet and zero if not.

## Thread Safety

**ctl_net_is_local_ip_address** is thread-safe.

# ctl_net_is_multicast_ip_address

## Synopsis

```
unsigned ctl_net_is_multicast_ip_address(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_net_is_multicast_ip_address** determines whether **addr** is an IP multicast address.
**ctl_net_is_multicast_ip_address** returns non-zero if **addr** is known to be a multicast address zero if not.

## Thread Safety

**ctl_net_is_multicast_ip_address** is thread-safe.

# ctl_net_is_private_ip_address

## Synopsis

```
unsigned ctl_net_is_private_ip_address(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_net_is_private_ip_address** determines whether **addr** is a private IPv4 address on the local subnet. **ctl_net_is_private_ip_address** returns non-zero if **addr** is a private address zero if not.

The private address ranges are 10.0.0.0—10.255.255.255 (10/8 prefix), 172.16.0.0—172.31.255.255 (172.16/12 prefix), and 192.168.0.0—192.168.255.255 (192.168/16 prefix).

## Thread Safety

**ctl_net_is_private_ip_address** is thread-safe.

# ctl_net_is_subnet_broadcast_address

## Synopsis

```
unsigned ctl_net_is_subnet_broadcast_address(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_net_is_subnet_broadcast_address** determines whether **addr** is an IP subnet broadcast address; the limited broadcast address 255.255.255.255 is not considered a subnet broadcast address by **ctl_net_is_subnet_broadcast_address**; if you need to know whether an IP address is a local subnet broadcast address or a limited broadcast address, use **ctl_net_is_local_broadcast_address**.

## Thread Safety

**ctl_net_is_subnet_broadcast_address** is thread-safe.

## See Also

ctl_net_is_local_broadcast_address

# ctl_net_mem_alloc_data

## Synopsis

```
void *ctl_net_mem_alloc_data(size_t byteSize,
                             CTL_TIME_t toTicks);
```

## Description

**ctl_net_mem_alloc_data** is a wrapper around the **alloc_data** member of the of the network memory manager (see **CTL_NET_MEM_DRIVER_t** and **ctl_net_mem_alloc_fn_t**).

The network memory manager will not use its entire heap for this request. Instead, a kilobyte or so is held in reserve for future **ctl_net_mem_alloc_xmit** requests and this routine will fail before dipping into that reserve. Buffers allocated with this routine should be freed using **ctl_net_mem_free**.

The network memory manager driver will return a word-aligned buffer of at least **byteSize** bytes if successful, null for fail. If **toTicks** is non-zero and the allocation initially fails, the routine will block in the hope that another task or ISR will call **ctl_net_mem_free** in the interim, giving the network memory manager adequate resources to perform the allocation.

## Thread Safety

Even with **toTicks** set to zero, **ctl_net_mem_alloc_data** routine is not safe to call from an ISR or a zero-priority main CTL task.

## See Also

CTL_NET_MEM_DRIVER_t, ctl_net_mem_alloc_xmit, ctl_net_mem_free

# ctl_net_mem_alloc_xmit

## Synopsis

```
void *ctl_net_mem_alloc_xmit(size_t byteSize,
                             CTL_TIME_t toTicks);
```

## Description

**ctl_net_mem_alloc_xmit** is a wrapper around the **alloc_xmit** member of the singleton instance of the network memory manager (see **CTL_NET_MEM_DRIVER_t** and **ctl_net_mem_alloc_fn_t**).

The network library memory manager attempts to use its entire heap to satisfy this request. Buffers allocated with this routine should be freed using **ctl_net_mem_free**.

The network memory manager driver returns a word-aligned buffer of at least **byteSize** bytes when successful, null for fail. If **toTicks** is non-zero and the allocation initially fails, **ctl_net_mem_alloc_xmit** blocks in the hope that another task or ISR will call **ctl_net_mem_free** in the interim, giving the network memory manager adequate resources to perform the allocation.

## Thread Safety

Even with **toTicks** set to zero, **ctl_net_mem_alloc_xmit** is not safe to call from an interrupt service routine.

## See Also

ctl_net_mem_alloc_data, ctl_net_mem_free, CTL_NET_MEM_DRIVER_t

# ctl_net_mem_free

## Synopsis

```
void ctl_net_mem_free(void *p);
```

## Description

**ctl_net_mem_free** frees the object pointed to by **p**; if **p** is a null pointer, **ctl_net_mem_free** does nothing.

**ctl_net_mem_free** is a wrapper around the **free_fn** member of the singleton instance of the network memory manager (see **CTL_NET_MEM_DRIVER_t**).

**ctl_net_mem_free** should only be used on buffers allocated with **ctl_net_mem_alloc_xmit** or **ctl_net_mem_alloc_data**. **ctl_net_mem_free** is safe to call from an interrupt service routine or the zero-priority main task.

## See Also

CTL_NET_MEM_FREE_FN_t, CTL_NET_MEM_DRIVER_t, ctl_net_mem_alloc_xmit, ctl_net_mem_alloc_data

# ctl_net_mem_trim

## Synopsis

```
void ctl_net_mem_trim(void *p,
                      size_t byteSize);
```

## Description

**ctl_net_mem_trim** is a wrapper around the **trim** member of the of the network memory manager (see **CTL_NET_MEM_DRIVER_t** and **ctl_net_mem_alloc_fn_t**).

This is a request to reduce the memory allocated and pointed to by **p** to **byteSize** bytes. It is guaranteed that **byteSize** is less than the currently allocated size for **p**. The network memory allocator is not required to trim its memory allocation, this call is an indication to the memory allocator that the extra memory will not be used by the network library and the allocator can recover it. It is acceptable for the implementation of the underlying trim function to do nothing.

# ctl_net_register_error_decoder

## Synopsis

```
void ctl_net_register_error_decoder(void);
```

## Description

**ctl_net_register_error_decoder** registers an error decoder with the CrossWorks runtime to decode errors generated by the TCP/IP Library.

# ctl_net_scan_dot_decimal_ip_addr

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_net_scan_dot_decimal_ip_addr(const char *str);
```

## Description

**ctl_net_scan_dot_decimal_ip_addr** parses the string pointed to by **str** as a a dotted-decimal IPv4 address and returns that address. If the string does not contain a valid IPv4 address, **ctl_net_scan_dot_decimal_ip_addr** returns an all-zero IP address.

## See Also

ctl_ip_sprint_addr.

# ctl_net_scan_mac_addr

## Synopsis

```
CTL_STATUS_t ctl_net_scan_mac_addr(CTL_NET_MAC_ADDR_t *dst,
                                   const char *text);
```

## Description

**ctl_net_scan_mac_addr** converts the zero-terminated string **text** into a MAC address in **dst**. The textual string is in the form "0A 1B 2C 4D F7 78"; the spaces between the octets can be any character, allowing use of both ':' and '-' as separators.

## Return Value

**ctl_net_scan_mac_addr** returns a standard status code.

## See Also

ctl_mac_sprint_addr.

# ctl_net_set_host_name

## Synopsis

```
void ctl_net_set_host_name(const char *name);
```

## Description

**ctl_net_set_host_name** sets the host name to the null-terminated string pointed to by **name**.
**ctl_net_set_host_name** makes a local copy of the host name which is truncated to 15 characters.

## See Also

ctl_net_get_host_name

# ctl_ntp_init

## Synopsis

```
CTL_STATUS_t ctl_ntp_init(void);
```

## Description

**ctl_ntp_init** initializes the NTP subsystem ready for use.

**ctl_ntp_init** returns **CTL_NO_ERROR** if the call was successful; i.e. the NTP callbacks were successfully registered with the UDP layer.

## Note

You must call call this after initializing the UDP subsystem with **ctl_udp_init**. NTP counts as one of your bound UDP ports.

## See Also

ctl_udp_init

# ctl_ntp_server_addr

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_ntp_server_addr(void);
```

## Description

**ctl_ntp_server_addr** returns the IPv4 address of the NTP server. If no NTP server is has been configured using **ctl_ntp_init**, **ctl_ntp_server_addr** returns an all-zero IP address.

# ctl_ntp_set_time_server

## Synopsis

```
CTL_STATUS_t ctl_ntp_set_time_server(CTL_NET_IPv4_ADDR_t addr);
```

## Description

**ctl_ntp_set_time_server** sets the address to use for the NTP time server to **addr**.

## Return Value

**ctl_ntp_set_time_server** returns a standard status code.

# ctl_soc_use_callback

## Synopsis

```
CTL_STATUS_t ctl_soc_use_callback(CTL_SOCKET_t s,
                                  CTL_TCP_SERVER_FN_t serverFn);
```

## Description

**ctl_soc_use_callback** assign the server callback function **serverFn** to the socket **s**. This function should only be called in the accept callback (see CTL_TCP_ACCEPT_FN_t).

'Callback' and 'Event' TCP server models are mutually exclusive—invoking this function will nullify the behavior set in a previous call to **ctl_soc_use_event**, **ctl_tcp_use_callback**, **ctl_tcp_use_event**.

## See Also

ctl_tcp_accept, ctl_tcp_bind, ctl_tcp_use_callback, ctl_soc_use_event, ctl_tcp_use_event

# ctl_soc_use_event

## Synopsis

```
CTL_STATUS_t ctl_soc_use_event(CTL_SOCKET_t s,
                               CTL_EVENT_SET_t *wakeEvent,
                               CTL_EVENT_SET_t wakeValue);
```

## Description

**ctl_soc_use_event** is a TCP server function to assign the wake event pointer and wake event value used for thread synchronization on a per-socket basis. **ctl_soc_use_event** This function should be called from the accept callback function (see CTL_TCP_ACCEPT_FN_t).

'Callback' and 'Event' TCP server models are mutually exclusive—invoking **ctl_soc_use_event** will nullify the behavior set in a previous call to **ctl_soc_use_callback**, **ctl_tcp_use_callback**, or **ctl_tcp_use_event**.

## See Also

ctl_soc_use_callback, ctl_tcp_use_event, ctl_tcp_accept, ctl_tcp_bind, ctl_tcp_use_callback

# ctl_tcp_accept

## Synopsis

```
CTL_STATUS_t ctl_tcp_accept(CTL_NET_PORT_t port,
                            CTL_TCP_ACCEPT_FN_t acceptFn);
```

## Description

**ctl_tcp_accept** registers the function **acceptFn** as the accept callback for port **port**. **port** is specified in network byte order.

**acceptFn** may be null, in which case all incoming connection requests are accepted provided that the number of open sockets is less than the allowed limit.

## See Also

CTL_TCP_ACCEPT_FN_t, ctl_tcp_bind

# ctl_tcp_bind

## Synopsis

```
CTL_STATUS_t ctl_tcp_bind(CTL_NET_PORT_t port);
```

## Description

**ctl_tcp_bind** reserves a listener for the TCP port **port**. **port** is specified in network byte order.

In the case of this library, "Bind" means "set aside one of the allocated server port slots for this port". "Unbind" means to free up the resource.

## See Also

ctl_tcp_unbind, ctl_tcp_accept, ctl_tcp_init

# ctl_tcp_close_socket

## Synopsis

```
void ctl_tcp_close_socket(CTL_SOCKET_t soc,
                          CTL_TCP_SOCKET_CLOSE_TYPE_t linger,
                          CTL_TIME_t timeout);
```

## Description

**ctl_tcp_close_socket** closes the socket **soc**. Closing can be either *graceful* or *hard*. A graceful shutdown involves invoking the three-way FIN handshake with the remote TCP after all outgoing data has been sent. A hard shutdown merely closes socket **soc** at the local end—any further packets from the socket's remote partner are NAKed with a reset response.

| linger | timeout | Type of close | Wait for close? |
|---|---|---|---|
| CTL_TCP_CLOSE_DONTLIN( | Don't care | Graceful | No |
| CTL_TCP_CLOSE_LINGER | Zero | Hard | No |
| CTL_TCP_CLOSE_LINGER | Nonzero | Graceful | Yes |

**ctl_tcp_close_socket** should not be invoked from the network task with **CTL_TCP_CLOSE_LINGER** and a non-zero timeout value. In other words, do not use the blocking version of this function in a UDP or TCP callback.

## See Also

ctl_tcp_shutdown

# ctl_tcp_connect

## Synopsis

```
CTL_STATUS_t ctl_tcp_connect(CTL_SOCKET_t s,
                             CTL_NET_IPv4_ADDR_t remoteIpAddr,
                             CTL_NET_PORT_t remotePort,
                             CTL_TIME_t timeout);
```

## Description

**ctl_tcp_connect** connects socket **s** to port **remotePort** of remote host **remoteIpAddr**. The socket should have been previously allocated with **ctl_tcp_socket**.

Returns **CTL_NO_ERROR** if successful or an error code for fail (i.e. no sockets were available).

There is no non-blocking version of this function. If **timeout** is non-zero **ctl_tcp_connect** will block until the connection is made or it times out. If **timeout** is zero, **ctl_tcp_connect** will block for a few microseconds until the network task signals that it has started the connect process.

If you call **ctl_tcp_connect** with **timeout** set to zero, you can poll the connection state using **ctl_tcp_get_socket_state** to determine when the connect (or fail timeout) occurs.

## Note

**ctl_tcp_connect** must not be called in the zero-priority main task nor should it be called from the network task (in a UDP or TCP receive callback).

## See Also

ctl_tcp_socket, ctl_tcp_get_socket_state

# ctl_tcp_get_local_ip_addr

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_tcp_get_local_ip_addr(CTL_SOCKET_t s);
```

## Description

**ctl_tcp_get_local_ip_addr** returns the IP address of the TCP partner of socket **soc** or zero if there is an error. The local IPv4 address is returned in network byte order.

# ctl_tcp_get_local_port

## Synopsis

```
CTL_NET_PORT_t ctl_tcp_get_local_port(CTL_SOCKET_t soc);
```

## Description

**ctl_tcp_get_local_port** returns the local port number for socket **soc** or zero if there is an error. The local remote port is returned in network byte order.

## See Also

ctl_tcp_get_remote_port, ctl_tcp_get_remote_ip_addr

# ctl_tcp_get_port_options

## Synopsis

```
CTL_STATUS_t ctl_tcp_get_port_options(CTL_NET_PORT_t port,
                                      CTL_TCP_PORT_OPTIONS_t *options);
```

## Description

**ctl_tcp_get_port_options** copies the port options used by a server on TCP port **port** to the buffer pointed to by **options**. **port** is specified in network byte order.

## See Also

CTL_TCP_PORT_OPTIONS_t, ctl_tcp_set_port_options, CTL_TCP_SOCKET_OPTIONS_t, ctl_tcp_get_socket_options, ctl_tcp_set_socket_options

CrossWorks TCP/IP Library

# ctl_tcp_get_remote_ip_addr

## Synopsis

```
CTL_NET_IPv4_ADDR_t ctl_tcp_get_remote_ip_addr(CTL_SOCKET_t soc);
```

## Description

**ctl_tcp_get_remote_ip_addr** returns the IP address for socket **soc** or zero if there is an error. The IP address is returned in network byte order.

## See Also

ctl_tcp_get_local_port

# ctl_tcp_get_remote_port

## Synopsis

```
CTL_NET_PORT_t ctl_tcp_get_remote_port(CTL_SOCKET_t soc);
```

## Description

**ctl_tcp_get_remote_port** returns the port number for the TCP partner of socket **soc** or zero if there is an error. The remote port is returned in network byte order.

## See Also

ctl_tcp_get_local_port, ctl_tcp_get_remote_ip_addr

# ctl_tcp_get_socket_connection_state

## Synopsis

```
CTL_TCP_SOCKET_CONNECTION_STATE_t ctl_tcp_get_socket_connection_state(CTL_SOCKET_t soc);
```

## Description

**ctl_tcp_get_socket_connection_state** returns the connection state of socket **soc**. If **soc** does not identify a socket, **ctl_tcp_get_socket_connection_state** returns **CTL_TCP_SOCKET_STATE_CLOSED**.

## See Also

CTL_TCP_SOCKET_CONNECTION_STATE_t

# ctl_tcp_get_socket_error

## Synopsis

```
CTL_STATUS_t ctl_tcp_get_socket_error(CTL_SOCKET_t soc);
```

## Description

**ctl_tcp_get_socket_error** returns the error state of the socket **soc**. If there is no error on the socket, **ctl_tcp_get_socket_error** returns **CTL_NO_ERROR**, otherwise one of the following error codes:

**CTL_NET_ERR_WOULDBLOCK**
: The operation cannot be completed without blocking and the application-layer software requested non-blocking operation.

**CTL_NET_ERR_ALREADY**
: The requested operation has already been performed.

**CTL_NET_ERR_NOTSOCK**
: Invalid socket descriptor.

**CTL_NET_ERR_OPNOTSUPP**
: Option not supported.

**CTL_NET_ERR_NETDOWN**
: Network interface is not configured or has a problem at the MAC level.

**CTL_NET_ERR_NETUNREACH**
: Network interface is not connected.

**CTL_NET_ERR_CONNABORTED**
: TCP connection aborted.

**CTL_NET_ERR_CONNRESET**
: TCP connection reset.

**CTL_NET_ERR_NOTCONN**
: Not connected.

**CTL_NET_ERR_TIMEDOUT**
: Timed out.

**CTL_NET_ERR_CONNREFUSED**
: The remote TCP refused our connection attempt.

**CTL_NET_ERR_HOSTUNREACH**
: The remote host does not respond.

**CTL_NET_ERR_NOTEMPTY**

A TCP connect call was made on a socket is already connected.

**CTL_NET_ERR_DISCON**

The socket was disconnected: no further communication is possible.

# ctl_tcp_get_socket_options

## Synopsis

```
CTL_STATUS_t ctl_tcp_get_socket_options(CTL_SOCKET_t s,
                                        CTL_TCP_SOCKET_OPTIONS_t *options);
```

## Description

**ctl_tcp_get_socket_options** copies the socket options used by socket **soc** into the buffer pointed to by **options**.

## See Also

CTL_TCP_SOCKET_OPTIONS_t, ctl_tcp_set_socket_options

# ctl_tcp_get_sockets

## Synopsis

```
unsigned ctl_tcp_get_sockets(CTL_NET_PORT_t port,
                             unsigned flags,
                             CTL_SOCKET_t *sockets,
                             unsigned max_socket_count);
```

## Description

**ctl_tcp_get_sockets** enumerates the sockets for the port **port** that match the conditions specified in **flags**. **port** is specified in network byte order. When a TCP server thread is woken up, before it can do anything useful it must first fetch a list of active sockets (on a per-port basis) using **ctl_tcp_get_sockets**.

**flags** is the bitwise-or of one or more of the flags in **CTL_TCP_GET_SOCKETS_FLAG_t**.

The sockets matching the combination of **flags** are written into the array pointed to by **sockets** which must have at least **max_socket_count** elements.

## Description

**ctl_tcp_get_sockets** returns the number of sockets that matched and were written into the **sockets** array.

## See Also

CTL_TCP_GET_SOCKETS_FLAG_t

# ctl_tcp_init

## Synopsis

```
CTL_STATUS_t ctl_tcp_init(unsigned socket_count,
                          unsigned listener_count,
                          CTL_TCP_GEN_ISS_FN_t issGenCallback);
```

## Description

**ctl_tcp_init** will attempt to allocate a buffer for its state data from the general heap and then register itself with the IP layer. The buffer allocation will be approximately:

(160 bytes * **socket_count**) + (48 bytes * **listener_count**)

Registration with the IP layer requires a small allocation as well. **ctl_tcp_init** must be invoked during initialization, prior to calling any other function in the TCP group.

A pseudo-random number generating routine, **issGenCallback**, must be provided to make 'initial send segments', **CTL_TCP_GEN_ISS_FN_t**.

In general, the free-running accumulator from the hardware timer that drives **ctl_get_current_time** is used for this purpose so the network library can provide a one-size-fits-all solution.

## See Also

CTL_TCP_GEN_ISS_FN_t

# ctl_tcp_look_ahead

## Synopsis

```
size_t ctl_tcp_look_ahead(CTL_SOCKET_t soc,
                          char ch);
```

## Description

**ctl_tcp_look_ahead** looks ahead to find the character **ch** in the received (but as yet unread) data for the socket **soc**.

**ctl_tcp_look_ahead** returns the number of characters that can be read from the socket such that the data on the socket is exhausted or the character **ch** is the terminating character read.

You can use **ctl_tcp_look_ahead**, for instance, to search for specific characters in the stream.

## See Also

ctl_tcp_read_line

# ctl_tcp_push

## Synopsis

```
CTL_STATUS_t ctl_tcp_push(CTL_SOCKET_t s);
```

## Description

**ctl_tcp_push** sends any data queued on socket **s** to the network layer for transmission. Socket **s** must first be in the connected state, **CTL_TCP_SOCKET_STATE_CONNECTED**, or **ctl_tcp_push** fails.

**ctl_tcp_push** is equivalent to calling **ctl_tcp_send** with no data and the push flag set.

# ctl_tcp_read_line

## Synopsis

```
CTL_STATUS_t ctl_tcp_read_line(CTL_SOCKET_t s,
                               char *str,
                               size_t size,
                               CTL_TIMEOUT_t type,
                               CTL_TIME_t timeout);
```

## Description

**ctl_tcp_read_line** reads a whole line up to and including the CR and optional LF from the socket **s**. **size** is the size of the string that the line is returned in.

If the whole string cannot be placed into **str**, characters beyond the end of the string, up to the end of the line, are discarded.

**ctl_tcp_read_line** returns the number of characters that have been consumed from the socket **s** which may be greater than the length of the returned string or **size**.

## See Also

ctl_tcp_look_ahead

# ctl_tcp_recv

## Synopsis

```
CTL_STATUS_t ctl_tcp_recv(CTL_SOCKET_t s,
                          void *buf,
                          size_t bufLen,
                          CTL_TIMEOUT_t type,
                          CTL_TIME_t timeout);
```

## Description

**ctl_tcp_recv** receives up to **bufLen** bytes into **buf** from socket **soc**.

Socket **soc** must first be in the connected state, **CTL_TCP_SOCKET_STATE_CONNECTED**, or **ctl_tcp_recv** will fail.

**buf** may be null, in which case up to **bufLen** bytes are discarded from the input stream.

The timeout value **timeout** can be zero to indicate a non-blocking call. If that is the case, this routine will retrieve as much data as it can (up to **bufLen**) from the socket and immediately return. In a blocking call, multiple passes across the task synchronization between the network task and the calling task may be required before the entire **bufLen** is received.

**ctl_tcp_recv** returns the count actually received for success or a standard CTL error code if the socket failed. A non-blocking call that received at least one byte but fewer than **bufLen** bytes is considered 'successful'.

## Note

**ctl_tcp_recv** must not be called from the network task with a non-zero **timeout**. In other words, do not use the blocking version of this function in a TCP server callback.

# ctl_tcp_send

## Synopsis

```
CTL_STATUS_t ctl_tcp_send(CTL_SOCKET_t s,
                          const void *buf,
                          size_t len,
                          CTL_TIMEOUT_t type,
                          CTL_TIME_t timeout,
                          unsigned flags);
```

## Description

**ctl_tcp_send** sends **len** bytes from **buf** to socket **s**. Socket **s** must first be in the connected state, **CTL_TCP_SOCKET_STATE_CONNECTED**, or **ctl_tcp_send** fails.

Setting the timeout value **ms** to zero indicates a non-blocking call. In this case, as much data as possible will be passed to the MAC before returning. In a blocking call, multiple passes across the task synchronization between the network task and the calling task may be required before the entire **len** is sent.

Parameter **flags** may be zero or a bitwise combination of the following:

**CTL_TCP_SEND_PUSH**
: Indicates the end of the current query or response to the remote TCP. In other words, this is the final call to **ctl_tcp_send** in a message.

**CTL_TCP_SEND_URGENT**
: Send out-of-band data.

**CTL_TCP_SEND_NOCOPY**
: Perform a zero-copy send of static data. This flag indicates that **buf** meets the target CPU's requirement for network DMA memory (if any) and that **buf** will remain in scope indefinitely. Buffer pointer **buf** will be passed through the stack directly to the MAC layer instead of copying its data to network memory first.

**CTL_TCP_SEND_FREE**
: Perform a zero-copy send of dynamic data. This flag indicates that **buf** has been allocated by application code using **ctl_net_mem_alloc_data** and that the network library is to use **ctl_net_mem_free** to free it after the remote TCP acknowledges receipt.

## Notes

Using the network library, the application layer has complete control over packet send coalescing. If the **CTL_TCP_SEND_PUSH** flag is not set, then an outgoing packet is only sent when a complete TCP segment has been built up. The **CTL_TCP_SEND_PUSH** flag will cause the **buf** and any previous queued send data to be sent to the remote TCP.

If **flags** is **CTL_TCP_SEND_PUSH**, **buf** may be null or **len** may be zero; in that case all previous queued data is sent on its way to the remote TCP.

### Return Value

**ctl_tcp_send** returns the count of bytes actually sent for success or a negative value for fail. A non-blocking call (**timeout** is zero) that sent at least one byte but less than **len** bytes is considered successful.

**ctl_tcp_send** should not be invoked from the network task with a non-zero **timeout** value. In other words, do not use the blocking version of this function in a TCP server callback.

### See Also

ctl_net_mem_alloc_data, ctl_net_mem_free, ctl_tcp_get_socket_state, ctl_tcp_get_socket_error

# ctl_tcp_set_port_options

## Synopsis

```
CTL_STATUS_t ctl_tcp_set_port_options(CTL_NET_PORT_t port,
                                      const CTL_TCP_PORT_OPTIONS_t *options);
```

## Description

**ctl_tcp_set_port_options** sets the server options for 'bound' TCP port **port**. The **socDefault** member of the port options only will be applied for newly-created sockets.

**ctl_tcp_set_port_options** returns **CTL_NO_ERROR** if the call was successful otherwise an error code if **port** is not a bound port.

## See Also

CTL_TCP_PORT_OPTIONS_t, ctl_tcp_get_port_options

# ctl_tcp_set_socket_options

## Synopsis

```
CTL_STATUS_t ctl_tcp_set_socket_options(CTL_SOCKET_t s,
                                        const CTL_TCP_SOCKET_OPTIONS_t *options);
```

## Description

**ctl_tcp_set_socket_options** copy the data pointed to by **options** to the set of values used by socket**soc**. This call should be made *prior* to a connection being established with a remote TCP. For a client socket, it means that the application layer should only use this function between the calls to **ctl_tcp_socket** and **ctl_tcp_connect**. For a server socket, it means that the appropriate place to call **ctl_tcp_set_socket_options** is in the 'accept' callback function.

## See Also

CTL_TCP_SOCKET_OPTIONS_t, ctl_tcp_get_socket_options

# ctl_tcp_shutdown

## Synopsis

```
void ctl_tcp_shutdown(CTL_SOCKET_t s);
```

## Description

**ctl_tcp_shutdown** begins the three-way shutdown handshake on socket **soc** after all outgoing data has been sent. Socket **soc**'s remote TCP partner is sent a FIN packet, indicating end-of-stream. Half-open connections are not supported—the classic socket's 'how' parameter is always **SD_BOTH**.

## See Also

ctl_tcp_socket, ctl_tcp_connect, ctl_tcp_send, ctl_tcp_recv, ctl_tcp_close_socket

# ctl_tcp_socket

## Synopsis

```
CTL_SOCKET_t ctl_tcp_socket(void);
```

## Description

**ctl_tcp_socket** fetches a TCP socket from the pool of unused sockets. **ctl_tcp_socket** returns a socket index if successful or zero for fail (i.e. no sockets were available for use).

Once a socket is allocated, application code must make a call to **ctl_tcp_connect** within 100 **CTL_TIME_t** units or the socket will be reclaimed by the network library.

## See Also

CTL_SOCKET_t, ctl_tcp_connect, ctl_tcp_send, ctl_tcp_recv, ctl_tcp_shutdown, ctl_tcp_close_socket

# ctl_tcp_unbind

## Synopsis

```
CTL_STATUS_t ctl_tcp_unbind(CTL_NET_PORT_t port);
```

## Description

**ctl_tcp_unbind** tells the TCP layer to stop accepting connections on TCP port **port**. **port** is specified in network byte order.

To resume accepting connections, call **ctl_tcp_bind** followed by **ctl_tcp_accept**.

## See Also

ctl_tcp_unbind, ctl_tcp_accept, CTL_NET_PORT_t, ctl_tcp_init

# ctl_tcp_use_callback

## Synopsis

```
CTL_STATUS_t ctl_tcp_use_callback(CTL_NET_PORT_t port,
                                  CTL_TCP_SERVER_FN_t serverFn);
```

## Description

**ctl_tcp_use_callback** sets **serverFn** to be the callback function for the the bound TCP port **port**. You should call **ctl_tcp_use_callback** after **ctl_tcp_bind** but before **ctl_tcp_accept**. **port** is specified in network byte order.

'Callback' and 'Event' TCP server models are mutually exclusive—invoking this function will nullify the behavior set in a previous call to **ctl_tcp_use_event**.

## See Also

ctl_tcp_use_event, ctl_tcp_use_callback, ctl_soc_use_event, ctl_tcp_accept, ctl_tcp_bind

# ctl_tcp_use_event

## Synopsis

```
CTL_STATUS_t ctl_tcp_use_event(CTL_NET_PORT_t port,
                               CTL_EVENT_SET_t *wake_event,
                               CTL_EVENT_SET_t wake_value);
```

**ctl_tcp_use_event** assigns the wake event pointer and value used for sockets used by a TCP server on port **port**. 'Callback' and 'event' TCP server models are mutually exclusive—invoking this function will nullify the behavior set in a previous call to **ctl_tcp_use_callback**.

## See Also

ctl_tcp_use_callback, ctl_soc_use_callback, ctl_soc_use_event

# ctl_udp_bind

## Synopsis

```
CTL_STATUS_t ctl_udp_bind(CTL_NET_PORT_t port,
                          CTL_UDP_RECV_FN_t callback);
```

## Description

**ctl_udp_bind** registers the callback function **callback** for received datagrams on UDP port **port**. To unbind a port to enable reuse of the port's resources, use **ctl_udp_unbind**.

**ctl_udp_bind** returns **CTL_NO_ERROR** if the call was successful; i.e. the number of bound ports was less than the value passed to **ctl_udp_init**.

## See Also

CTL_UDP_RECV_FN_t, ctl_udp_unbind

# ctl_udp_init

## Synopsis

```
CTL_STATUS_t ctl_udp_init(const CTL_UDP_CONFIGURATION_t *init_data);
```

## Description

**ctl_udp_init** initializes the UDP layer with the configuration parameters in **init_data**.

The configuration parameter **max_bound_ports** sets the maximum number of bound UDP ports. The UDP layer will attempt to allocate a buffer for its state data, of approximately eight bytes times **max_bound_ports**, and then register the UDP layer with the IP layer.

The configuration parameters **min_ephemeral_port** and **max_ephemeral_port** define the UDP ephemeral port range.

You can elect to use a default configuration by passing a null pointer for **init_data**. In this case, the UDP layer is initialized with a maximum of 20 bound UDP ports with the ephemeral UDP port range being between 1024 and 65535.

## Note

**ctl_udp_init** must be called prior to calling **ctl_udp_bind**.

## See Also

CTL_UDP_RECV_FN_t, ctl_udp_bind

# ctl_udp_sendto

## Synopsis

```
void ctl_udp_sendto(void *data,
                    size_t byte_count,
                    const CTL_UDP_INFO_t *info,
                    unsigned flags);
```

## Description

**ctl_udp_sendto** sends a UDP datagram to a remote host. The member **other_port** of **info** is the remote port and the member **otherIpAddr** of **info** is the remote IP address.

**ctl_udp_sendto** will return almost immediately, after the outgoing datagram has been queued for transmission by the MAC layer or queued for ARP hold at the IP layer.

The UDP datagram will be dropped by the network library if:

- The destination IP address is not on the local subnet (as returned by **ctl_net_is_local_ip_address**) and no gateway is configured, or
- The network library cannot allocate an Ethernet transmission frame for the datagram, or
- The network library cannot allocate network memory for the datagram payload.

**flags** may be zero or one of the following:

**CTL_UDP_SENDTO_NOCOPY**
: Perform a zero-copy send of static data. This flag indicates that **data** meets the target CPU's requirement for network DMA memory (if any) and that **data** will remain in scope indefinitely. Buffer pointer **data** will be passed through the stack directly to the MAC layer instead of copying its data to network memory first.

**CTL_UDP_SENDTO_FREE**
: Perform a zero-copy send of dynamic data. This flag indicates that **data** was allocated by application code using **ctl_net_mem_alloc_data** and that the library is to use **ctl_net_mem_free** to free it after it is sent.

## See Also

CTL_UDP_INFO_t, ctl_udp_init, ctl_udp_bind, ctl_net_mem_alloc_data, ctl_net_mem_free

CrossWorks TCP/IP Library

# ctl_udp_unbind

## Synopsis

```
CTL_STATUS_t ctl_udp_unbind(CTL_NET_PORT_t port);
```

## Description

**ctl_udp_unbind** unregisters any associated callback function associated with UDP port **port**.

**ctl_udp_unbind** returns **CTL_NO_ERROR** if the call was successful; i.e. the port is current bound, otherwise an error code.

## See Also

CTL_UDP_RECV_FN_t, ctl_udp_init

# <ctl_net_hw.h>

## Overview

This is the private set of functions and types that are required to implement a MAC or PHY driver when porting the Network Library to a new device.

## API Summary

| | |
|---|---|
| **Constants** | |
| CTL_NET_ETHERNET_HEADER_SIZE | The number of bytes in an Ethernet header |
| CTL_NET_ETHERNET_PDU_SIZE | The size of the PDU of a Ethernet II frame |
| **Types** | |
| CTL_ETH_RX_FRAME_t | Receive frame buffer descriptor |
| CTL_ETH_TX_FRAME_t | Transmit frame descriptor |
| CTL_NET_INTERFACE_t | A network interface |
| **MAC** | |
| CTL_MAC_STATE_t | MAC states |
| CTL_NET_MAC_DRIVER_t | MAC driver |
| ctl_mac_get_state | Return MAC state |
| ctl_mac_init | MAC-layer driver initialization function |
| ctl_mac_send | Send Ethernet frame to MAC |
| ctl_mac_update | Wapper for MAC update |
| ctl_mac_wake_net_task | Wake network task for MAC event |
| ctl_net_process_received_frame | Process received frame |
| **PHY** | |
| CTL_NET_PHY_DRIVER_t | PHY driver |
| CTL_PHY_ERROR_t | RMII, MII, and PHY layer errors |
| CTL_PHY_STATE_t | PHY state |
| ctl_net_get_phy_name | Get PHY name |
| ctl_net_read_phy_operating_mode | Returns PHY operating mode |
| ctl_net_read_phy_register | Read a PHY register |
| ctl_net_read_phy_state | Read state of the PHY driver |
| ctl_net_search_for_first_phy | Search for attached PHY |
| ctl_net_update_phy | PHY-layer update function |
| ctl_phy_read_id | Read the PHY ID |

| | |
|---|---|
| ctl_phy_reset | Reset the PHY |
| **MII** | |
| CTL_NET_MAC_MII_DEFERRED_READ_FN_t | Initiates a read of the MII management interface |
| ctl_mac_mii_deferred_read | Initiate an asynchronous read of the MII management interface |
| ctl_mac_mii_deferred_read_result | Get result of last asynchronous MII read |
| ctl_mac_mii_read | Read the MII management interface |
| **Memory management** | |
| CTL_NET_MEM_DRIVER_t | Network stack memory manager |
| ctl_net_set_mem_driver | Set the network memory allocator |
| **Utility** | |
| ctl_net_do_mac_dis_connect | Signal change of media connection state |
| **Stock PHY drivers** | |
| ctl_phy_lm3s_init_driver | Luminary Stellaris integrated PHY driver setup |
| **Frames** | |
| CTL_ETH_HEADER_t | 802.3 Ethernet Header |

# CTL_ETH_HEADER_t

## Synopsis

```
typedef struct {
  unsigned short __required_align;
  unsigned char ethDstMac[];
  unsigned char ethSrcMac[];
  unsigned short ethType;
} CTL_ETH_HEADER_t;
```

## Description

The Ethernet header is 14 bytes long. In order to make the subsequent IP and TCP/UDP headers align on a 32-bit word, an extra short is added to the start of the structure. The 1536 byte frame buffer passed back and forth with the hardware actually begins at &**ethDstMac[0]**.

Some MAC layers have a short word length field preceding the Ethernet header when the data is sent/received to the hardware. The `__required_align` short mentioned in the preceding paragraph is used for that purpose. An example of this is the Ethernet FIFO on the TI LM3S Stellaris devices.

# CTL_ETH_RX_FRAME_t

## Synopsis

```
typedef struct {
  CTL_ETH_HEADER_t *data;
  unsigned byteCount;
  unsigned ethAndIpByteCount;
} CTL_ETH_RX_FRAME_t;
```

## Description

For frame-based MACs (LPC2xxx, STR91x, SAM7X), there is a rotating ring of receive frame buffers that are passed to the network task during processing of received frames.

For FIFO-based MACs (Stellaris, ENC28J60), there is a single static receive frame buffer that is filled and then passed to the network task as frames arrive.

In either case, the stack or application code must not hold on to any data in the received frame outside the context of **ctl_net_process_received_frame**, nor should it block in **ctl_net_process_received_frame** (which includes any UDP callback handler).

The members are:

### `data`

Pointer to the complete Ethernet receive frame; the Ethernet header and payload data are held in a single chunk, unlike transmission frames which separate header and payload.

### `byteCount`

The total count of bytes in the received Ethernet frame, which excludes the `__required_align` member in the Ethernet header, and excludes the FCS appended by the transmitting MAC.

### `ethAndIpByteCount`

Set in the IP layer. The offset of the start of the TCP, UDP, or ICMP header after IP options have been parsed, relative to *the start of the Ethernet frame, excluding the alignment short*.

# CTL_ETH_TX_FRAME_t

## Synopsis

```
typedef struct {
  CTL_ETH_HEADER_t *header;
  unsigned short header_byte_count;
  unsigned short payload_byte_count;
  void *payload;
  void *payload_free;
} CTL_ETH_TX_FRAME_t;
```

## Description

Transmit frames are allocated from the network stack's private heap by the highest-level stack code (TCP or UDP or ICMP) and then passed down the stack to the MAC layer, which ctl_net_mem_free()s the memory allocated to the frame and its header data.

A separate pointer, **payload_free**, is provided for the MAC layer to free payload data. This is decoupled from the actual 'payload' pointer for a number of reasons:

- TCP payload data is not freed from the MAC layer; a null pointer does double duty as a 'do not free' flag.
- It can be desirable for the "payload" data to be a subset of a larger block of memory which should all be freed on transmit completion.
- With fragmented IP packets, the entire buffer is freed after the final fragment is transmitted.

The members are:

### `header`

A pointer to the header data to transmit, guaranteed to be correctly aligned for the MAC. Data transmission starts with `header->ethDstMac`.

### `header_byte_count`

The number of header bytes to transmit. This byte count always excludes the `__required_align` member from the count. Frames presented to the MAC driver for transmission are guaranteed that `header_byte_count+2` is divisible by four.

### `payload`

A pointer to the payload data, if any, and guaranteed to be correctly aligned for the MAC. If there is no payload, this member must be set to zero.

### `payload_byte_count`

The number of bytes in the payload. If there is no payload, this member must be set to zero.

### `payload_free`

The data to free once the frame is sent. If there is nothing to free, this member must be zero.

# CTL_MAC_STATE_t

## Synopsis

```
typedef enum {
  CTL_MAC_STATE_FATAL_ERROR,
  CTL_MAC_STATE_NO_LINK,
  CTL_MAC_STATE_NEEDS_REINIT,
  CTL_MAC_STATE_CONNECTED
} CTL_MAC_STATE_t;
```

## Description

**CTL_MAC_STATE_t** defines the internal states that the MAC state machine may go though. A MAC driver can use this to maintain its internal state.

# CTL_NET_ETHERNET_HEADER_SIZE

## Synopsis

```
#define CTL_NET_ETHERNET_HEADER_SIZE   (6+6+2)
```

## Description

**CTL_NET_ETHERNET_HEADER_SIZE** defines the number of bytes in an Ethernet II header. The Ethernet header comprises six bytes of source MAC address, six bytes of destination MAC address, and two bytes for the EtherType field.

Note that we do not support 802.1Q VLAN tagging nor do we support non-Ethernet LAN protocols that rely on IEEE 802.2 LLC encapsulation at present.

# CTL_NET_ETHERNET_PDU_SIZE

## Synopsis

```
#define CTL_NET_ETHERNET_PDU_SIZE      1500
```

## Description

**CTL_NET_ETHERNET_PDU_SIZE** defines the number of bytes of payload data (the network PDU) in an Ethernet II frame.

In general, when dealing with Ethernet MAC drivers, we have:

1. A 16-bit padding `short`, 2 bytes. (Required to align TCP headers)
2. Destination MAC, 6 bytes.
3. Source MAC, 6 bytes.
4. Ethernet Type/Frame Size, 2 bytes. (16 bits, including padding)
5. Payload of 1,500 bytes.
6. FCS, 4 bytes.

Excluding the padding, 1518 bytes. Including the padding, 1520 bytes, which is divisible by four.

# CTL_NET_INTERFACE_t

## Synopsis

```
typedef struct {
  CTL_NET_MAC_DRIVER_t mac;
  CTL_NET_PHY_DRIVER_t phy;
} CTL_NET_INTERFACE_t;
```

## Description

**CTL_NET_INTERFACE_t** describes a single network interface.

## Structure

### mac

The MAC driver that the network interface uses.

### phy

The PHY driver associated with the MAC interface.

# CTL_NET_MAC_DRIVER_t

## Synopsis

```
typedef struct {
  CTL_NET_MAC_ADDR_t mac_addr;
  CTL_NET_MAC_INIT_FN_t init_fn;
  CTL_NET_MAC_UPDATE_FN_t update_fn;
  CTL_NET_MAC_GET_STATE_FN_t get_state_fn;
  CTL_NET_MAC_SEND_FN_t send_fn;
  CTL_NET_MAC_MULTICAST_ACCEPT_FN_t multicast_accept_fn;
  CTL_NET_MAC_MULTICAST_QUERY_FN_t multicast_query_fn;
  CTL_NET_MAC_MII_WRITE_FN_t mii_write_fn;
  CTL_NET_MAC_MII_READ_FN_t mii_read_fn;
  CTL_NET_MAC_MII_DEFERRED_READ_FN_t mii_deferred_read_fn;
  CTL_NET_MAC_MII_DEFERRED_READ_RESULT_FN_t mii_deferred_read_result_fn;
  CTL_NET_MAC_SELECT_PHY_FN_t select_phy_fn;
  CTL_NET_MAC_PRIVATE_s *device;
} CTL_NET_MAC_DRIVER_t;
```

## Associated types

```
typedef CTL_STATUS_t (*CTL_NET_MAC_INIT_FN_t)(CTL_NET_INTERFACE_t *);
```

```
typedef void (*CTL_NET_MAC_UPDATE_FN_t)(CTL_NET_INTERFACE_t *, unsigned);
```

```
typedef CTL_MAC_STATE_t (*CTL_NET_MAC_GET_STATE_FN_t)(CTL_NET_INTERFACE_t *);
```

```
typedef void (*CTL_NET_MAC_SEND_FN_t)(CTL_NET_INTERFACE_t *, CTL_ETH_TX_FRAME_t *);
```

```
typedef unsigned (*CTL_NET_MAC_MULTICAST_ACCEPT_FN_t)(CTL_NET_INTERFACE_t *, const
 CTL_NET_MAC_ADDR_t *, unsigned);
```

```
typedef unsigned (*CTL_NET_MAC_MULTICAST_QUERY_FN_t)(CTL_NET_INTERFACE_t *, const
 CTL_NET_MAC_ADDR_t *);
```

```
typedef CTL_STATUS_t (*CTL_NET_MAC_MII_WRITE_FN_t)(CTL_NET_INTERFACE_t *, int , int);
```

```
typedef CTL_STATUS_t (*CTL_NET_MAC_MII_READ_FN_t)(CTL_NET_INTERFACE_t *, int);
```

```
typedef CTL_STATUS_t (*CTL_NET_MAC_MII_DEFERRED_READ_FN_t)(CTL_NET_INTERFACE_t *, int);
```

```
typedef CTL_STATUS_t (*CTL_NET_MAC_MII_DEFERRED_READ_RESULT_FN_t)(CTL_NET_INTERFACE_t *);
```

```
typedef CTL_STATUS_t (*CTL_NET_MAC_SELECT_PHY_FN_t)(CTL_NET_INTERFACE_t *);
```

## Description

**CTL_NET_MAC_DRIVER_t** holds the data and functions that handle the MAC layer.

### mac_addr

The Ethernet MAC address that the network interface uses. You must set this before calling **init_fn**.

**init_fn**

This should return non-zero if MAC hardware initialization was successful.

**update_fn**

The network stack will call **update_fn** called periodically (with a non-zero isHousekeeping) or when the network task is activated by **ctl_mac_wake_net_task**.

**get_state_fn**

The network stack will will call **get_state_fn** to query the state of the MAC in various layers.

**send_fn**

The IP layer will call **send_fn** to send a frame to the MAC for transmission. **send_fn** must be thread-safe.

**multicast_accept_fn**

Enable or disable accepting packets given the layer 2 destination address. Returns non-zero if successful.

**multicast_query_fn**

Returns non-zero if the MAC layer is currently accepting mulitcast packets with the given MAC address.

**mii_write_fn**

Writes to a PHY register. This doesn't need to be thread-safe as it is only called from the network task.

**mii_read_fn**

Reads a PHY register. This doesn't need to be thread-safe as it is only called from the network task.

**mii_deferred_read_fn**

Start a deferred read of an MII register. The result will be read by calling **mii_deferred_read_result_fn**.

**mii_deferred_read_result_fn**

Return the PHY register requested by **mii_deferred_read_fn**. If the result is not ready, return **CTL_PHY_AGAIN**, or an error code if there is an error, else the register contents.

**select_phy_fn**

Select the appropriate PHY attached to the MAC.

**device**

Additional MAC data, if any.

# CTL_NET_MAC_MII_DEFERRED_READ_FN_t

## Synopsis

```
typedef CTL_STATUS_t (*CTL_NET_MAC_MII_DEFERRED_READ_FN_t)(CTL_NET_INTERFACE_t *, int);
```

## Description

**CTL_NET_MAC_MII_DEFERRED_READ_FN_t** is the MAC-layer MII management interface deferred read function signature. This function in the network interface initializes a read of the MII/RMII management interface and immediately returns.

# CTL_NET_MEM_DRIVER_t

## Synopsis

```
typedef struct {
  CTL_NET_MEM_FREE_FN_t free_fn;
  CTL_NET_MEM_ALLOC_FN_t alloc_xmit_fn;
  CTL_NET_MEM_ALLOC_FN_t alloc_data_fn;
  CTL_NET_MEM_TRIM_FN_t trim_fn;
} CTL_NET_MEM_DRIVER_t;
```

## Description

In order to get the most flexibility out of a limited resource, the network library dynamically allocates RAM where and when it needs it. Systems that have dedicated Ethernet memory may use the network stack's built-in 'net memory manager' to manage the pool of Ethernet memory that is used for outgoing frames and TCP and UDP buffers.

Targets that don't have dedicated Ethernet memory may still benefit from using the net memory manager. Because the stack memory allocations are extremely transitory, more often than not there is no net memory allocated and the net memory heap is thus not fragmented. Using a private sub-heap is much more efficient than using the general heap in this particular case.

If you must squeeze every last bit of flexibility from dynamic RAM, then there is a stack version of the net memory manager that uses the general heap. You gain access to "all" of the heap, but you will be sharing it with the rest of the application and you will take a performance hit because of fragmentation issues.

The MAC layer is responsible for freeing net memory. After transmit, it should call **ctl_net_mem_free** on transmit frames (and their data) that it gets from the higher stack layers. The **hdrData** pointer of the **CTL_ETH_TX_FRAME_t** should always be **ctl_net_mem_free**'d, as well as the **payload_free** pointer (if it is non-null) and the **CTL_ETH_TX_FRAME_t** itself.

**ctl_net_mem_alloc_xmit** and **ctl_net_mem_alloc_data** both allocate memory for the network stack. The difference is that **ctl_net_mem_alloc_xmit** will attempt to take every last byte in the heap if that is what is required, while **ctl_net_mem_alloc_data** will attempt to leave a few bytes for future transmit allocations.

The reason for this duality is to prevent a potential fatal embrace whereby there is data available to be sent, but a transmit frame cannot be allocated to send it. Application code should always use **ctl_net_mem_alloc_data** when allocating memory from the network heap.

### `free_fn`

Method to free previously-allocated memory.

### `alloc_xmit_fn`

Method to allocate data for a transmit frame.

### `alloc_data_fn`

Method to allocate data for payload.

## See Also

CTL_NET_MEM_FREE_FN_t, CTL_NET_MEM_ALLOC_FN_t

# CTL_NET_PHY_DRIVER_t

## Synopsis

```
typedef struct {
  int addr;
  unsigned short operating_mode;
  unsigned short configuration_flags;
  unsigned short mii_mode;
  unsigned short auto_negotiation;
  CTL_NET_PHY_INIT_FN_t init_fn;
  CTL_NET_PHY_UPDATE_FN_t update_fn;
  CTL_PHY_STATE_t state;
  CTL_MUTEX_t mutex;
  const char *name;
} CTL_NET_PHY_DRIVER_t;
```

## Description

**CTL_NET_PHY_DRIVER_t** contains data and hardware-specific function overloads for the PHY layer. The **CTL_NET_PHY_DRIVER_t** structure has the following members:

### addr

The address of the PHY in use, 0 through 31. The network stack sets this member before initializing the PHY using **init_fn**.

### flags

The PHY-layer flags including link capability and operating mode.

### state

The logical state of the PHY. This member must only be written by the **update_fn** method, to reflect the current link state.

### init_fn

The MAC layer should call the wrapper version of this function, **ctl_phy_init**, during hardware initialization, after the MII is initialized.

### update_fn

The network task will periodically call the wrapper version of this function, **ctl_net_update_phy**, to update the PHY state.

### mutex

When user-level code wants access to PHY registers, this holds off the periodic functions so we can access the PHY ourselves. There is no need for direct access to this mutex as the wrapper functions **ctl_net_read_phy_register** and **ctl_net_get_phy_state** lock the mutex to prevent simultaneous access by the network task.

### name

The device name of the PHY.

## See Also

ctl_net_update_phy, ctl_net_get_phy_state, ctl_net_read_phy_operating_mode, ctl_net_read_phy_register.

# CTL_PHY_ERROR_t

## Synopsis

```
typedef enum {
  CTL_PHY_MII_READ_FAILURE,
  CTL_PHY_MII_WRITE_FAILURE,
  CTL_PHY_RESET_FAILURE,
  CTL_PHY_NOT_FOUND,
  CTL_PHY_INCORRECT_ID,
  CTL_PHY_UNSUPPORTED_ID,
  CTL_PHY_AGAIN
} CTL_PHY_ERROR_t;
```

## Description

**CTL_PHY_ERROR_t** defines the potential errors from the MII, RMII, and PHY.

# CTL_PHY_STATE_t

## Synopsis

```
typedef enum {
  CTL_PHY_STATE_ERROR,
  CTL_PHY_STATE_NO_LINK,
  CTL_PHY_STATE_NEGOTIATING,
  CTL_PHY_STATE_LINKED,
  CTL_PHY_STATE_INITIALIZE
} CTL_PHY_STATE_t;
```

## Description

**CTL_PHY_STATE_t** is the set of values that a PHY driver should report as its 'state' to the outside world, even if its actual state machine is more complicated than that represented here.

### CTL_PHY_STATE_INITIALIZE

Indicates that the PHY requires initializing.

### CTL_PHY_STATE_ERROR

An error prevents the PHY from operating.

### CTL_PHY_STATE_NO_LINK

The Ethernet cable or other media is unplugged.

### CTL_PHY_STATE_NEGOTIATING

The PHY is negotiating duplex and transmission rate with its partner.

### CTL_PHY_STATE_LINKED

The PHY and its partner have completed negotiating, the link is active.

## See Also

ctl_net_get_phy_state

# ctl_mac_get_state

## Synopsis

```
CTL_MAC_STATE_t ctl_mac_get_state(CTL_NET_INTERFACE_t *self);
```

## Description

**ctl_mac_get_state** returns the MAC state for the network interface **self**.

# ctl_mac_init

## Synopsis

```
CTL_STATUS_t ctl_mac_init(CTL_NET_INTERFACE_t *self);
```

## Description

**ctl_mac_init** initializes the MAC on the network interface **self**. In effect, **ctl_mac_init** is a wrapper around the **init_fn** member of the the **CTL_NET_MAC_DRIVER_t** driver. You need to call **ctl_mac_init** from your application code. **ctl_mac_init** returns a MAC-layer or PHY-layer error status.

## See Also

CTL_NET_MAC_DRIVER_t, ctl_phy_init

# ctl_mac_mii_deferred_read

## Synopsis

```
CTL_STATUS_t ctl_mac_mii_deferred_read(CTL_NET_INTERFACE_t *net,
                                       int reg);
```

## Description

**ctl_mac_mii_deferred_read** is a wrapper around the **mii_deferred_read_fn** member of the network MAC driver.

The valid range for **devAddr** is 0 through 31 and needs to match the PHY chip's physical address, which is typically set on the PHY hardware using strapping pins. See your PHY chip's datasheet for valid values of **reg**.

You can retrieve the result of the deferred read using **ctl_mac_mii_deferred_read_result**.

## See Also

ctl_mac_mii_deferred_read_fn_t, CTL_NET_MAC_DRIVER_t, ctl_mac_mii_deferred_read_result

# ctl_mac_mii_deferred_read_result

## Synopsis

```
CTL_STATUS_t ctl_mac_mii_deferred_read_result(CTL_NET_INTERFACE_t *net);
```

## Description

**ctl_mac_mii_deferred_read_result** returns the result of the last read of the MII management interface without blocking or busy-wait. This is a wrapper around the **mii_deferred_read_result_fn** member of the network driver.

Return values are the same as **ctl_mac_mii_read**: negactive for failure, a number between 0 and 0xFFFF (inclusive) for success.

## See Also

ctl_mac_mii_deferred_read_fn_t, CTL_NET_MAC_DRIVER_t

# ctl_mac_mii_read

## Synopsis

```
CTL_STATUS_t ctl_mac_mii_read(CTL_NET_INTERFACE_t *net,
                              int reg);
```

## Description

**ctl_mac_mii_read** busy-waits until the result is available. This is a wrapper around the **mii_read_fn** member of the network MAC driver.

The valid range for **devAddr** is 0 through 31 and needs to match the PHY chip's physical address, which is typically set on the PHY hardware using strapping pins. See your PHY chip's datasheet for valid values of **reg**. Return values are negative for failure, a number between 0 and 0xFFFF (inclusive) for success.

## See Also

ctl_mac_mii_read_fn_t, CTL_NET_MAC_DRIVER_t, ctl_mac_mii_deferred_read

# ctl_mac_send

## Synopsis

```
void ctl_mac_send(CTL_ETH_TX_FRAME_t *frame);
```

## Description

**ctl_mac_send** sends the Ethernet frame **frame** to the MAC for transmission. Note that this can be called from any thread, not just the network thread dealing with TCP segments. For instance, UDP frames are sent in the context of the sending thread.

# ctl_mac_update

## Synopsis

```
void ctl_mac_update(unsigned isHousekeeping);
```

## Description

**ctl_mac_update** is a wrapper for the `update` method of the network interface.

# ctl_mac_wake_net_task

## Synopsis

```
void ctl_mac_wake_net_task(void);
```

## Description

**ctl_mac_wake_net_task** must be called by the MAC-layer driver's interrupt service routine when there is action to be taken in the network stack task. **ctl_mac_wake_net_task** will wake the network task, which will call **ctl_mac_update** in due course.

## See Also

ctl_mac_update, ctl_net_process_received_frame

# ctl_net_do_mac_dis_connect

## Synopsis

```
void ctl_net_do_mac_dis_connect(void);
```

## Description

**ctl_net_do_mac_dis_connect** signals to the network stack that the media connected to an network interface has changed state, such as unplugging or plugging the Ethernet cable.

It is intended that MAC-layer or PHY-layer drivers call **ctl_net_do_mac_dis_connect** when they detect that the media has changed as the PHY will renegotiate its operating parameters and the MAC may well need to be reconfigured for inter-packet gaps and so on. In addition, the network stack must renegotiate its DHCP parameters.

# ctl_net_get_phy_name

## Synopsis

```
char *ctl_net_get_phy_name(CTL_NET_INTERFACE_t *self);
```

## Description

**ctl_net_get_phy_name** returns the name of the PHY driver attached to the network interface **self**.

CrossWorks TCP/IP Library

# ctl_net_process_received_frame

## Synopsis

```
void ctl_net_process_received_frame(CTL_ETH_RX_FRAME_t *frame);
```

## Description

**ctl_net_process_received_frame** should be called by the network interface's MAC update function for each Ethernet frame the interface receives.

# ctl_net_read_phy_operating_mode

## Synopsis

```
int ctl_net_read_phy_operating_mode(CTL_NET_INTERFACE_t *self);
```

## Description

**ctl_net_read_phy_operating_mode** returns the PHY flags for the network interface **net**.

## See Also

CTL_NET_PHY_DRIVER_t

# ctl_net_read_phy_register

## Synopsis

```
CTL_STATUS_t ctl_net_read_phy_register(CTL_NET_INTERFACE_t *self,
                                       int reg);
```

## Description

**ctl_net_read_phy_register** reads PHY register **reg** from the PHY associated with the network interface **self**.

You can call this from any task to read the PHY register as access to the MAC and PHY is protected by a mutex.

## Return Value

**ctl_net_read_phy_register** returns a standard status code.

# ctl_net_read_phy_state

## Synopsis

```
CTL_PHY_STATE_t ctl_net_read_phy_state(CTL_NET_INTERFACE_t *self);
```

## Description

**ctl_net_read_phy_state** is a wrapper for the **get_state_fn** member of the PHY-layer driver.

## See Also

CTL_NET_PHY_DRIVER_t, CTL_PHY_STATE_t

# ctl_net_search_for_first_phy

## Synopsis

```
CTL_STATUS_t ctl_net_search_for_first_phy(CTL_NET_INTERFACE_t *net);
```

## Description

**ctl_net_search_for_first_phy** tries to read the PHY identification registers from each PHY on the MAC interface **net**, starting at address zero and progressing through address 31. If a PHY is found, **ctl_net_search_for_first_phy** returns the address corresponding to that PHY and the PHY address is set in the network interface's PHY driver. If no PHY is found, **ctl_net_search_for_first_phy** returns **CTL_PHY_NOT_FOUND**.

## Return Value

**ctl_net_search_for_first_phy** returns a standard status code.

# ctl_net_set_mem_driver

## Synopsis

```
void ctl_net_set_mem_driver(const CTL_NET_MEM_DRIVER_t *mem);
```

## Description

**ctl_net_set_mem_driver** sets the memory driver to **mem**.

CrossWorks TCP/IP Library

# ctl_net_update_phy

## Synopsis

```
void ctl_net_update_phy(CTL_NET_INTERFACE_t *self);
```

## Description

**ctl_net_update_phy** is a wrapper around the **update_fn** of the PHY layer driver. It is called periodically by the CTL stack task when nothing else is happening.

# ctl_phy_lm3s_init_driver

## Synopsis

```
void ctl_phy_lm3s_init_driver(CTL_NET_PHY_DRIVER_t *self);
```

## Description

**ctl_phy_lm3s_init_driver** initializes **driver** with functions that implement the PHY state machine for the Luminary Stellaris integrated PHY.

# ctl_phy_read_id

## Synopsis

```
CTL_STATUS_t ctl_phy_read_id(CTL_NET_INTERFACE_t *self,
                             unsigned long *id);
```

## Description

**ctl_phy_read_id** reads the PHY device identification register. The ID is returned with the least significant four bits, which indicates the PHY revision, set to zero.

## Return Value

**ctl_phy_read_id** returns a standard status code.

# ctl_phy_reset

## Synopsis

```
CTL_STATUS_t ctl_phy_reset(CTL_NET_INTERFACE_t *self);
```

## Description

**ctl_phy_reset** resets the PHY using the standard BMCR register.

## Return Value

**ctl_phy_reset** returns a standard status code.

# <ctl_net_private.h>

## API Summary

| | |
|---|---|
| **IP** | |
| CTL_IPV4_HEADER_t | IPv4 header |
| CTL_IP_STATS_t | IP statistics |
| **Utility** | |
| ctl_eth_tx_frame_total_count | Compute total Ethernet frame size |
| ctl_ipv4_rx_payload_start | Get a pointer to receive frame's payload |
| **Transmission Frames** | |
| ctl_eth_alloc_tx_frame | Allocate a transmission frame |
| ctl_eth_free_tx_frame | Free a transmission frame |
| **ARP** | |
| ctl_arp_init | Initialize ARP |
| **IP Function** | |
| ctl_ipv4_rx_payload_byte_count | Calculate IPv4 payload length |
| **Utility functions** | |
| ctl_ipv4_make_multicast_mac_addr | Create a multicast MAC address |
| **TCP** | |
| ctl_net_calc_cksum | Calculates the TCP checksum over 16-bit data |
| ctl_net_normalize_cksum_and_comp | Normalize and complement a calculated TCP checksum |
| ctl_net_sum_bytes | Calculates the TCP checksum over 16-bit data |
| ctl_tcp_register_stats | Register TCP statistics |
| ***** UNASSIGNED GROUP ***** | |
| ctl_dns_register_stats | Register statistics for the DNS module |

# CTL_IPV4_HEADER_t

## Synopsis

```
typedef struct {
  unsigned short __required_align;
  unsigned char ethDstMac[];
  unsigned char ethSrcMac[];
  unsigned short ethType;
  unsigned char ipVerHl;
  unsigned char ipDifServEcn;
  unsigned short ipTotalLen;
  unsigned short ipIdent;
  unsigned short ipFlagsFragOffst;
  unsigned char ipTtl;
  unsigned char ipProtocol;
  unsigned short ipHdrChecksum;
  unsigned long ipSrcAddr;
  unsigned long ipDstAddr;
  unsigned short ipOptions[];
} CTL_IPV4_HEADER_t;
```

## Description

**CTL_IPV4_HEADER_t** describes the layout of the IPv4 header. We include the Ethernet header because they are always adjacent. But this is the last layer we can do this with. Because of the variable-length **ipOptions** field, we can't fix where the start of the transport (or user datagram) layer is after the IP layer.

# CTL_IP_STATS_t

## Synopsis

```
typedef struct {
  long rxPackets;
  long badRxHdrSize;
  long checksumFail;
  long badRxSize;
  long promiscousPacket;
  long rxBroadcastPacket;
  long badRxProt;
  long sendFragMallocFail;
  long unsupportedProtocol;
  long txFramesDropped;
  long txFramesHeldForARP;
  long txFramesDirectToMAC;
} CTL_IP_STATS_t;
```

## Description

**CTL_IP_STATS_t** holds the statistics related to IP.

### `txFramesHeldForARP`

The number of frames that required ARP lookup before being passed to the MAC driver.

### `txFramesDirectToMAC`

The number of frames passed directly to the MAC driver as the Ethernet address of the frame is known without broadcasting an ARP request for it.

# ctl_arp_init

## Synopsis

```
CTL_STATUS_t ctl_arp_init(void);
```

## Description

**ctl_arp_init** initializes the ARP protocol and creates an ARP cache with a default eight entries. You *must* call **ctl_arp_init** before initializing other protocols.

By default the ARP cache will use the system memory allocator **ctl_system_memory_allocator** to allocate its cache. If you want to use a different memory allocator, for instance to dedicate a fixed memory size to the ARP cache, you can replace the default allocator using **ctl_arp_set_memory_allocator**.

## Thread Safety

**ctl_arp_init** is thread-safe.

## See Also

ctl_arp_set_cache_size

# ctl_dns_register_stats

## Synopsis

```
void ctl_dns_register_stats(void);
```

# ctl_eth_alloc_tx_frame

## Synopsis

```
CTL_ETH_TX_FRAME_t *ctl_eth_alloc_tx_frame(size_t header_byte_count,
                                           CTL_TIME_t timeout);
```

## Description

**ctl_eth_alloc_tx_frame** allocate a new transmission frame from network memory and initializes fields within the frame. A header is allocated and assigned to the **header** member of the allocated frame. **header_byte_count** is the number of byes to allocate for the header, *must* be a multiple of four, and *must* include the alignment short. If the header size is not a multiple of four, the frame isn't allocated.

Once allocated, the **header_byte_count** of the frame is initialized to the **header_byte_count** parameter adjusted to remove the alignment short size.

## Return Value

**ctl_eth_alloc_tx_frame** returns a pointer to the allocated frame or zero if the frame cannot be allocated.

# ctl_eth_free_tx_frame

## Synopsis

```
void ctl_eth_free_tx_frame(CTL_ETH_TX_FRAME_t *frame);
```

## Description

**ctl_eth_free_tx_frame** frees the transmission frame **frame** along with any memory that needs to be freed from the frame's header and payload.

# ctl_eth_tx_frame_total_count

## Synopsis

```
unsigned long ctl_eth_tx_frame_total_count(const CTL_ETH_TX_FRAME_t *frame);
```

## Description

**ctl_eth_tx_frame_total_count** computes the total number of bytes in the Ethernet frame **frame** which is the sum of the header size and payload size. The header size includes the 12 bytes of Ethernet header.

# ctl_ipv4_make_multicast_mac_addr

## Synopsis

```
void ctl_ipv4_make_multicast_mac_addr(unsigned char *dst,
                                      CTL_NET_IPv4_ADDR_t ip_addr);
```

## Description

**ctl_ipv4_make_multicast_mac_addr** creates a multicast Ethernet MAC address in **dst** for the IPv4 address **ip_addr**.

# ctl_ipv4_rx_payload_byte_count

## Synopsis

```
unsigned ctl_ipv4_rx_payload_byte_count(CTL_ETH_RX_FRAME_t *frame);
```

# ctl_ipv4_rx_payload_start

## Synopsis

```
void *ctl_ipv4_rx_payload_start(CTL_ETH_RX_FRAME_t *rxFrame);
```

# ctl_net_calc_cksum

## Synopsis

```
unsigned short ctl_net_calc_cksum(unsigned long seed,
                                  const unsigned short *data,
                                  size_t byte_count);
```

## Description

**ctl_net_calc_cksum** calculates the checksum over an array of shorts. See RFC 1071. The returned value is 0 through 65535 with all end-around carries accounted for.

## Note

Data in and out of checksum functions are in network byte order. Actually, it doesn't matter which byte order is used as long as the answer is the same byte order.

**seed** contains the value calculated from the TCP or UDP pseudo-header.

# ctl_net_normalize_cksum_and_comp

## Synopsis

```
unsigned short ctl_net_normalize_cksum_and_comp(unsigned long sum);
```

## Description

**ctl_net_normalize_cksum_and_comp** normalizes the checksum **sum** and complements it such that the output is a correct 16-bit TCP checksum in network byte order. See RFC 1071.

# ctl_net_sum_bytes

## Synopsis

```
unsigned long ctl_net_sum_bytes(unsigned long sum,
                                const unsigned char *data,
                                size_t byte_count);
```

## Description

**ctl_net_sum_bytes** calculates the checksum over an array of shorts. See RFC 1071. The returned value is 0 through 65535 with all end-around carries accounted for.

## Note

Data in and out of checksum functions are in network byte order. Actually, it doesn't matter which byte order is used as long as the answer is the same byte order.

# ctl_tcp_register_stats

## Synopsis

```
void ctl_tcp_register_stats(void);
```

## Description

**ctl_tcp_register_stats** registers the statistics associated with TCP. Note that statistics regarding TCP are always collected but they are exposed to the user only by registering with the statistics manager.

The statistics are:

### `failed_checksum`

The number of TCP segments received with a failed checksum.

### `bad_length`

The number of TCP segments received which had a bad length.

### `tx_malloc_fail`

When a TCP segment is ready for transmission, the network stack attempts to allocate a transmission frame. If the stack fails allocate a transmission frame because there is insufficient memory, it is recorded in this statistic.

### `state_error`

This records the number of times that the TCP state machine is detected to be in error. This can happen when packets arrive that do not conform to the TCP state machine.

### `bad_mss`

The number of socket connections attempted with an invalid MSS.

### `cnx_refused_unsupported`

The TCP connection request was refused because there are no listeners registered for the port.

### `cnx_refused_ports`

The TCP connection request was refused because the maximum number of connections are already open for the port.

### `cnx_refused_sockets`

The TCP connection request was refused because there are insufficient sockets in the socket pool to establish a connection.

### `tx_total_retrans`

The total number of retransmission requests because an ACK from the other TCP was lost.

### `tx_1_retrans`

A count of the number of segments that required a single retransmission as an ACK form the other TCP was lost.

**tx_2_retrans**

A count of the number of segments that required two retransmission as an ACK form the other TCP was lost.

**tx_unreach**

A count of the number of segments that exceeded two retransmissions and considered the other TCP unreachable.

**rx_fast_retrans**

A count of the number of received segments that are lost and the network stack re-requested using the *fast restransmission* algorithm.

CrossWorks TCP/IP Library

# <ctl_net_tcp_private.h>

## API Summary

| | |
|---|---|
| **Segments** | |
| CTL_TCP_SEGMENT_t | A TCP segment |
| **Types** | |
| CTL_TCP_APP_LAYER_CMD_t | Application-layer command |
| CTL_TCP_SOCKET_STATE_t | TCP socket states |

# CTL_TCP_APP_LAYER_CMD_t

## Synopsis

```
typedef enum {
  alcNone,
  alcBlockedOnWrite,
  alcBlockedOnRead,
  alcConnect,
  alcConnectAndBlock,
  alcShutdown,
  alcCloseHard,
  alcCloseGraceful,
  alcLingeringClose,
  alcRecycle
} CTL_TCP_APP_LAYER_CMD_t;
```

# CTL_TCP_SEGMENT_t

## Synopsis

```
typedef struct {
  CTL_TCP_SEGMENT_s *next;
  size_t allocatedByteSize;
  size_t byteCount;
  unsigned long segStart;
  CTL_TIME_t timeStamp;
  unsigned long *freeExternalBuf;
  unsigned short sentCount;
  unsigned char flags;
  unsigned long data[];
} CTL_TCP_SEGMENT_t;
```

## Description

**CTL_TCP_SEGMENT_t** describes a single TCP segment in a transmit or receive queue.

### next

The next segment in the list; null indicates no further segments.

### allocatedByteSize

The number of bytes allocated to segment payload data (in the `data` member).

### byteCount

The number of valid payload bytes in the payload data. This will be less than or equal to `allocatedByteSize`.

### segStart

The segment start sequence number.

### timeStamp

The last 'sent' time for a transmit segment or 'received' time for a receive segment.

### freeExternalBuf

Additional memory to free when the segment is itself freed. Only transmit frames set this to a non-null value.

### sentCount

A count of the number of times this frame has been sent; this is only manipulated for segments in the send queue.

# CTL_TCP_SOCKET_STATE_t

## Synopsis

```
typedef enum {
  CTL_NCP_SOCKET_STATE_UNALLOCATED,
  CTL_NCP_SOCKET_STATE_CLOSED,
  CTL_NCP_SOCKET_STATE_LISTEN,
  CTL_NCP_SOCKET_SYN_SENT,
  CTL_NCP_SOCKET_SYN_RECEIVED,
  CTL_NCP_SOCKET_ESTABLISHED,
  CTL_NCP_SOCKET_FIN_WAIT1,
  CTL_NCP_SOCKET_FIN_WAIT2,
  CTL_NCP_SOCKET_CLOSE_WAIT,
  CTL_NCP_SOCKET_CLOSING,
  CTL_NCP_SOCKET_LAST_ACK,
  CTL_NCP_SOCKET_TIME_WAIT
} CTL_TCP_SOCKET_STATE_t;
```

## Description

**CTL_TCP_SOCKET_STATE_t** describes the state of the socket along the lines of RFC 793.

### CTL_NCP_SOCKET_STATE_UNALLOCATED

NOT RFC 793...kind of like "Super Duper Closed". The RFC assumes that the system has dynamic socket allocation; we don't.

### CTL_NCP_SOCKET_STATE_CLOSED

No connection state at all.

### CTL_NCP_SOCKET_STATE_LISTEN

Waiting for a connection request from any remote TCP and port.

### CTL_NCP_SOCKET_SYN_SENT

Waiting for a matching connection request after having sent a connection request.

### CTL_NCP_SOCKET_SYN_RECEIVED

Waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

### CTL_NCP_SOCKET_ESTABLISHED

An open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

### CTL_NCP_SOCKET_FIN_WAIT1

Waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

### CTL_NCP_SOCKET_FIN_WAIT2

Waiting for a connection termination request from the remote TCP.

**CTL_NCP_SOCKET_CLOSE_WAIT**

Waiting for a connection termination request from the local user.

**CTL_NCP_SOCKET_CLOSING**

Waiting for a connection termination request acknowledgment from the remote TCP.

**CTL_NCP_SOCKET_LAST_ACK**

Waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).

**CTL_NCP_SOCKET_TIME_WAIT**

Waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

# <designware_emac_v2.h>

## Overview

Synopsis DesignWare 10/100 Ethernet MAC driver.

This is implemented in the following device families:

- LPC1700
- LPC2300, LPC2400
- LPC3000, LPC3100, LPC3200

## API Summary

| | |
|---|---|
| **Setup** | |
| designware_emac_v2_init_mac_driver | Initialize the network interface |
| **Control** | |
| designware_emac_v2_isr | Handle network interrupt |
| designware_emac_v2_start | Start the network interface |
| **Status** | |
| designware_emac_v2_first_free | Return extent of memory consumed |

# designware_emac_v2_first_free

## Synopsis

```
void *designware_emac_v2_first_free(CTL_NET_INTERFACE_t *self);
```

## Description

**designware_emac_v2_first_free** returns a pointer to the first byte free for use by the application after the allocation of transmit and receive descriptors. The client can use this to add all remaining memory to the network heap, for example.

You can call this after initializing the network interface using **designware_emac_v2_init**.

## See Also

designware_emac_v2_init.

# designware_emac_v2_init_mac_driver

## Synopsis

```
void designware_emac_v2_init_mac_driver(CTL_NET_INTERFACE_t *self,
                                        void *emac,
                                        void *mem,
                                        int tx_descriptor_count,
                                        int rx_descriptor_count,
                                        unsigned clock,
                                        int interruptSource);
```

## Description

**designware_emac_v2_init_mac_driver** initializes the network interface **self** but does not start it. The DesignWare 10/100 EMAC register interface is specified in **emac** and the memory required to hold the transmit and receive descriptors is specified in **mem**.

The number of transmit and receive descriptors are passed in **tx_descriptor_count** and **rx_descriptor_count**. At least two transmit descriptors are required, and transmit performance of the TCP/IP library will scale with the number of descriptors allocated. At least one receive descriptor is required, and receive performance of the TCP/IP library will scale with the number of descriptors allocated.

The clock provided to the module is passed in **clock**, in Hertz. The driver automatically configures the MAC to divide the module clock in order to clock the management interface at a maximum of 2.5 MHz.

The interrupt source associated with the MAC is passed in **interruptSource**.

## Note

**mem** must be accessible by the DMA engine of the Ethernet MAC. Please ensure that you pass an appropriate address for the descriptors by consulting the user manual of your device.

# designware_emac_v2_isr

## Synopsis

```
void designware_emac_v2_isr(CTL_NET_INTERFACE_t *self);
```

## Description

**designware_emac_v2_isr** must be called to handle the interrupt generated by the network interface **self**. **designware_emac_v2_isr** will iterate the transmit descriptors preparing for transmission and wake the network task to process received packets.

# designware_emac_v2_start

## Synopsis

```
CTL_STATUS_t designware_emac_v2_start(CTL_NET_INTERFACE_t *self);
```

## Description

**designware_emac_v2_start** starts the network interface **self**. The start sequence calls the method to select and initialize the PHY, initialize the receive and transmit descriptors, and enable interrupts.

## Return Value

**designware_emac_v2_start** returns a standard status code.

# <designware_emac_v3.h>

## Overview

Synopsis DesignWare 10/100 Ethernet MAC driver.

This is implemented in the following device families:

- STM32F1, STM32F2, STM32F4
- LPC1800, LPC4000, LPC4300
- XMC4500
- TMC4129x

The following MAC versions are currently known and some of the marketing material from the various devices transcribed:

### Version 3.4

- IEEE 802.3-2002 standard for Ethernet MAC
- IEEE 1588-2002 standard for precision networked clock synchronization

Present on STM32F1.

### Version 3.5

- IEEE 802.3-2002 standard for Ethernet MAC
- IEEE 1588-2002 standard for precision networked clock synchronization

Present on STM32F4.

### Version 3.6

Present on LPC4300.

### Version 3.7

- IEEE 802.3-2008 standard for Ethernet MAC
- IEEE 1588-2008 standard for precision networked clock synchronization

Present on XMC4500 and TMC4129X.

## API Summary

| | |
|---|---|
| **Setup** | |
| designware_emac_v3_init | Initialize the network interface |
| **Control** | |
| designware_emac_v3_isr | Handle network interrupt |

| | |
|---|---|
| designware_emac_v3_start | Start the network interface |
| **Status** | |
| designware_emac_v3_first_free | Return extent of memory consumed |
| designware_emac_v3_version | Return the Synopsis version of the EMAC |

# designware_emac_v3_first_free

## Synopsis

```
void *designware_emac_v3_first_free(CTL_NET_INTERFACE_t *self);
```

## Description

**designware_emac_v3_first_free** returns a pointer to the first byte free for use by the application after the allocation of transmit and receive descriptors. The client can use this to add all remaining memory to the network heap, for example.

You can call this after initializing the network interface using **designware_emac_v3_init**.

## See Also

designware_emac_v3_init.

# designware_emac_v3_init

## Synopsis

```
void designware_emac_v3_init(CTL_NET_INTERFACE_t *self,
                             void *emac,
                             void *mem,
                             int tx_descriptor_count,
                             int rx_descriptor_count,
                             int clock);
```

## Description

**designware_emac_v3_init** initializes the network interface **self** but does not start it. The DesignWare 10/100 EMAC register interface is specified in **emac** and the memory required to hold the transmit and receive descriptors is specified in **mem**.

The number of transmit and receive descriptors are passed in **tx_descriptor_count** and **rx_descriptor_count**. At least two transmit descriptors are required, and transmit performance of the TCP/IP library will scale with the number of descriptors allocated. At least one receive descriptor is required, and receive performance of the TCP/IP library will scale with the number of descriptors allocated.

The clock provided to the module is passed in **clock**, in Hertz. The driver automatically configures the MAC to divide the module clock in order to clock the management interface at a maximum of 2.5 MHz.

## Note

**mem** must be accessible by the DMA engine of the Ethernet MAC. Please ensure that you pass an appropriate address for the descriptors by consulting the user manual of your device.

# designware_emac_v3_isr

## Synopsis

```
void designware_emac_v3_isr(CTL_NET_INTERFACE_t *self);
```

## Description

**designware_emac_v3_isr** must be called to handle the interrupt generated by the network interface **self**. **designware_emac_v3_isr** will iterate the transmit descriptors preparing for transmission and wake the network task to process received packets.

# designware_emac_v3_start

## Synopsis

```
CTL_STATUS_t designware_emac_v3_start(CTL_NET_INTERFACE_t *self);
```

## Description

**designware_emac_v3_start** starts the network interface **self**. The start sequence calls the method to select and initialize the PHY, initialize the receive and transmit descriptors, and enable interrupts.

## Return Value

**designware_emac_v3_start** returns a standard status code.

# designware_emac_v3_version

## Synopsis

```
CTL_STATUS_t designware_emac_v3_version(CTL_NET_INTERFACE_t *self,
                                        char *version);
```

## Description

**designware_emac_v3_version** returns the Synopsis version number of the DesignWare 10/100 EMAC **self**.

If **version** is non-zero, it must point to an array of at least six characters where the decoded version number is written as a null-terminated string.

## Return Value

The version number as an 8-bit value where the most significant four bits define the major version number and the least significant four bits define the minor version number.

CrossWorks TCP/IP Library

# <enc28j60.h>

## Overview

Driver for a Microchip ENC28J60 MAC and integrated PHY.

## API Summary

| | |
|---|---|
| **MAC** | |
| enc28j60_mac_setup | Configure ENC28J60 MAC |
| **PHY** | |
| ENC28J60_PHY_ID | PHY ID |
| enc28j60_phy_init_driver | Initialize ENC28J60 integrated PHY driver |

# ENC28J60_PHY_ID

## Synopsis

```
#define ENC28J60_PHY_ID  0x00831400
```

## Description

**ENC28J60_PHY_ID** is the ID returned by the ENC28J60 PHY.

# enc28j60_mac_setup

## Synopsis

```
CTL_STATUS_t enc28j60_mac_setup(CTL_NET_INTERFACE_t *self,
                                CTL_SPI_DEVICE_t *dev);
```

## Description

**enc28j60_mac_setup** initializes **self** with functions that implement the MAC interface for the ENC28J60. The ENC28J60 is addressed using the SPI device **dev**.

## Return Value

**enc28j60_mac_setup** returns a standard status code.

# enc28j60_phy_init_driver

## Synopsis

```
void enc28j60_phy_init_driver(CTL_NET_PHY_DRIVER_t *self);
```

## Description

**enc28j60_phy_init_driver** initializes **self** with functions that implement the PHY state machine for the Microchip ENC28J60 integrated PHY.

CrossWorks TCP/IP Library

# <dp83848.h>

## Overview

PHY driver for a Texas Instruments DP83848.

## API Summary

| PHY | |
|---|---|
| DP83848_PHY_ID | PHY ID |
| dp83848_phy_init_driver | PHY driver setup |

# DP83848_PHY_ID

## Synopsis

```
#define DP83848_PHY_ID  0x20005C90
```

## Description

**DP83848_PHY_ID** is the ID returned by the DP83848 PHY.

# dp83848_phy_init_driver

## Synopsis

```
void dp83848_phy_init_driver(CTL_NET_PHY_DRIVER_t *self);
```

## Description

**dp83848_phy_init_driver** initializes **self** with functions that implement the PHY state machine for the Texas Instruments DP83848.

CrossWorks TCP/IP Library

# <ksz8721bl.h>

## Overview

PHY driver for a Micrel KSZ8721BL.

## API Summary

| PHY | |
|---|---|
| KSZ8721BL_PHY_ID | PHY ID |
| ksz8721bl_phy_init_driver | PHY driver setup |

# KSZ8721BL_PHY_ID

## Synopsis

```
#define KSZ8721BL_PHY_ID  0x00221610
```

## Description

**KSZ8721BL_PHY_ID** is the ID returned by the KSZ8721BL PHY.

# ksz8721bl_phy_init_driver

## Synopsis

```
void ksz8721bl_phy_init_driver(CTL_NET_PHY_DRIVER_t *self);
```

## Description

**ksz8721bl_phy_init_driver** initializes **self** with functions that implement the PHY state machine for the Micrel KSZ8721BL.

CrossWorks TCP/IP Library

# <lan8720a.h>

## Overview

PHY driver for a SMSC LAN8720A.

## API Summary

| PHY | |
|---|---|
| LAN8720A_PHY_ID | PHY ID |
| lan8720a_phy_init_driver | PHY driver setup |

# LAN8720A_PHY_ID

## Synopsis

```
#define LAN8720A_PHY_ID  0x0007C0F0
```

## Description

**LAN8720A_PHY_ID** is the ID returned by the LAN8720A PHY.

# lan8720a_phy_init_driver

## Synopsis

```
void lan8720a_phy_init_driver(CTL_NET_PHY_DRIVER_t *self);
```

## Description

**lan8720a_phy_init_driver** initializes **self** with functions that implement the PHY state machine for the SMSC LAN8720A.

CrossWorks TCP/IP Library

# <lm3s_phy.h>

## Overview

PHY driver for the integrated LM3S Stellaris PHY.

## API Summary

| PHY | |
|---|---|
| LM3S_PHY_ID | PHY ID |
| lm3s_phy_init_driver | PHY driver setup |

# LM3S_PHY_ID

## Synopsis

```
#define LM3S_PHY_ID  0x0161B410
```

## Description

**LM3S_PHY_ID** is the ID returned by the LM3S Stellaris PHY.

# lm3s_phy_init_driver

## Synopsis

```
void lm3s_phy_init_driver(CTL_NET_PHY_DRIVER_t *self);
```

## Description

**lm3s_phy_init_driver** initializes **self** with functions that implement the PHY state machine for the LM3S integrated PHY.