# IBEX
# ETHERNET TCP/IP SOURCE CODE DRIVER PROJECT

# CONTENTS

# DRIVER OVERVIEW

## GENERAL

Ethernet is the perfect solution to providing connectivity for all sorts of embedded devices. Not only does the incredibly versatile range of TCP/IP stack functions lend themselves to all sorts of communication applications, the internet and web browsers allow easy communications across the world and with anyone's computer or smart phone, effectively for free. The barriers to use once caused by expensive embedded device program memory are now all but gone with today's low end microcontrollers typically providing plenty of memory for a TCP/IP stack.

The simplicity of the design of each individual part of a typical TCP/IP stack has allowed it to become the universal standard it is today. However whilst simple in concept the implementation of a complete set of TCP/IP stack components is a large and complex task by any programmers standards. This driver removes you from all of that complexity and allows you to add an Ethernet interface to your project with ease. Each of the stack components is configurable to allow you to easily tailor the overall stack depending on each of your projects requirements.

## DRIVER INTRODUCTION

This TCP/IP stack is designed from the ground up to be lean, yet extremely fast and powerful. The aim of the driver is for it to be the perfect choice for applications ranging from small 8 bit low speed embedded devices needing TCP/IP functionality, to fast 16 and 32 bit devices that need to provide high throughput and performance without requiring huge resources. Whilst these aims may seem to contradict each other, accomplishing this is actually relatively straightforward with the right decisions made about the architecture of the stack. With a typical TCP/IP stack being comprised of a large number of lines of code it can be very daunting for anyone other than the original programmers to navigate and understanding it, let alone tailor it to their own needs with any sense of confidence. However the other main aim of this stack is for it to be easy for programmers to understand and customise if need be and it has been implemented with huge importance placed on the source code remaining simple, straightforward and easy to understand.

The driver code has been designed using ANSI compliant C compliers. Using the driver with other ANSI compliant C compliers and with other processors / microcontrollers should not present significant problems, but you should ensure that you have sufficient programming expertise to carry out any modifications that may be required to the source code. Embedded-code.com source code is written to be very easy to understand by programmers of all levels. The code is very highly commented with no lazy programming techniques. All function, variable and constant names are fully descriptive to help you modify and expand the code with ease.

The driver and associated files are provided under a licence agreement. Please see www.embedded-code.com/licence.php for full details.

The remainder of this manual provides simple guides to using the driver as well as a wealth of technical information about how the driver works. We welcome any feedback on this manual and the driver.

## FEATURES

Optimised for small memory embedded designs. Processor ram usage is kept to a minimum and the Ethernet Controller IC memory is released as soon as packets are transmitted, allowing for high throughput applications.

Intelligent TCP handling to provide automatic re-transmission in the event of lost packets, but without using any large data buffers. If a packet gets lost it is automatically re-sent with the lost packet data simply re-generated.

Powerful web server with built in dynamic data handling, allowing your application to automatically add additional information to each web page as it is transmitted. Design your web pages using your favourite web design package and then use the included PC application to automatically convert the entire web site, including its images, scripts, etc, into either:

> A single C compliant .h header file which you simply add to your project files and the entire web site is then automatically stored in the program memory of your device when you next compile. No additional work is required by you – the driver automatically deals with serving the web pages and content on demand.
>
> A single binary data file ready to store in a flash memory IC, with a header file to add to your project allowing the HTTP server to automatically deal with serving the entire web site as you designed it. You just provide the method to store the single binary data block to your flash memory and to read the binary data from it on request by the HTTP server driver.

Receive input from a web browser using GET and POST methods. GET provides simple form or hyperlink inputs to be received. POST provides more advanced form entry and file upload. The driver incorporates full handling of uploaded files allowing users to upload configuration files, new firmware, new web site content, etc. The driver simply passes the received and decoded file data to your function ready for you to store or process as you wish.

As the majority of internet connected networks will allow outbound communications but will block inbound connections this driver also provides the functionality to allow your embedded device to connect to the internet itself, avoiding the need for a remote device or PC to have to make a connection to it. Having this functionality avoids special changes needing to be made by network administrators to permit inbound communications to your embedded device, which is often an issue in today's security conscious world. POP3 email functionality allows your embedded device to connect to a POP3 mail server to check for incoming emails. SMTP and MIME email functionality allows your embedded device to send emails periodically or in response to events or emails received, with files attached if you wish. DNS functionality allows you to use URL's rather than IP addresses, ensuring your device will continue to work should your mail server or remote server IP address be changed.

NetBIOS functionality allowing you to connect to your device by name rather than IP address on local networks.

This driver uses Internet Protocol Version 4 (IPv4) which is the dominant protocol of the Internet. Although its successor Internet Protocol Version 6 (IPv6) is now being actively deployed, the abundance of IPv4 devices means that it isn't going away anytime soon, if ever.

This detailed project technical manual with simple guidance on how to include the driver in your project and detailed explanations for people who want to know more about the inner workings the stack.

Full source code supplied for you to use and modify as required.

## The following protocols are included

UDP – Fast non-managed data transfer

> Client and server functionality.
>
> Multiple connections may be open and active at the same time.

TCP – Fast connection based managed data transfers

> Client and server functionality.
>
> Multiple connections may be open and active at the same time.

HTTP – Web server

> Multiple connections may be open and active at the same time.

DHCP – Automatic network configuration client

> Automatic request of IP address, subnet and gateway address so your device can simply plug and play on a network.

NetBIOS – Use names instead of IP address on local networks

Find your device by a text based name rather than IP address on a local network.

Can be very useful when using DHCP.

DNS – Lookup of IP address from a URL

Allows your embedded device to use text URL's to connect to remote devices, such as mail servers or your own remote servers.

An important feature to protect against changes in a remote servers IP address effectively stopping your device from connecting to it.

POP3 – Receive email

Automatically check and download email from a standard POP3 mailbox.

Get round strict network administrators in this security critical world. If your embedded device will be behind routers or firewalls use POP3 to allow your device to retrieve messages via the internet itself, avoiding the need to configure the routers or firewalls to allow an incoming connection.

SMTP – Send email

Automatically send anyone emails in response to events or at fixed intervals. Request an email to be returned to you using POP3

MIME Base64 encoding – Attach files to emails being sent

Easy attachment of files to outgoing emails. For instance you could attach .txt files containing configuration settings or.csv spreadsheet files with logging data for opening with Excel and other spreadsheet applications. Any file type may be attached.

ICMP – Ping response

Responds to ping requests.

ARP – Address resolution protocol

A basic requirement for a TCP/IP device.

SNTP Time Server Client

Use internet public time servers to obtain the current time.

## Drivers for the following Network Interface Controller IC's are included

10Base-T

Microchip ENC28J60 (SPI bus hardware interface)

Realtek RTL8019AS (8 bit hardware interface)

10Base-T / 100Base-TX

SMSC LAN91C111 (8 or 16 bit hardware interface)

NXP LPC2000 series ARM microcontroller with built in Ethernet interface and external KSZ8001 PHY IC.

Adding support for other network interface controller IC's is straightforward due to the driver's simple approach to the handling of the Ethernet receive and transmit buffers. There are no requirements to dynamically reserve memory areas for active connections. Instead a single transmit buffer is assigned for outgoing packets and the remainder of the memory is available for the rolling receive buffer. As well as making it straightforward to add new devices this approach also means that there is always the maximum possible receive buffer space available to queue incoming packets and avoid lost packets in heavy network traffic conditions.

# Adding The Driver To Your Project

## Notes About Our Source Code Files

### How We Organise Our Project Files

There are many different ways to organise your source code and many different opinions on the best method! We have chosen the following as a very good approach that is widely used, well suited to both small and large projects and simple to follow.

Each .c source code file has a matching .h header file. All function and memory definitions are made in the header file. The .c source code file only contains functions. The header file is separated into distinct sections to make it easy to find things you are looking for. The function and data memory definition sections are split up to allow the defining of local (this source code file only) and global (all source code files that include this header file) functions and variables. To use a function or variable from another .c source code file simply include the .h header file.

Variable types BYTE, WORD, SIGNED_WORD, DWORD, SIGNED_DWORD are used to allow easy compatibility with other compilers. A WORD is 16 bits and a DWORD is 32 bits. Our projects include a ‘main.h’ global header file which is included in every .c source code file. This file contains the typedef statements mapping these variable types to the compiler specific types. You may prefer to use an alternative method in which case you should modify as required. Our main.h header file also includes any project wide global defines.

This is much easier to see in use than to try and explain and a quick look through any of the included sample projects will show you by example.

Please also refer to the resources section of the embedded-code.com web site for additional documentation which may be useful to you.

### Modifying Our Project Files

We may issue new versions of our source code files from time to time due to improved functionality, bug fixes, additional device / compiler support, etc. Where possible you should try not to modify our source codes files and instead call the driver functions from other files in your application. Where you need to alter the source code it is a good idea to consider marking areas you have changed with some form of comment marker so that if you need to use an upgraded driver file it is as easy as possible to upgrade and still include all of the additions and changes that you have made.

## Step By Step Instructions

### Move The Main Driver Files To Your Project Directory

The following files are the main driver files which you need to copy to your main project directory:

eth-main.c – TCP/IP Stack main functions

eth-main.h

eth-arp.c – Address resolution protocol functions

eth-arp.h

eth-ip.c – IP functions

eth-ip.h

eth-nic.c – Network Interface IC functions. Select the pair of files for the nic

eth-nic.h device being used.

## Move The Generic Global Defines File To You Project Directory

The generic global file is located in each driver sample project directory. Select the most suitable sample project based on the compiler being used and copy the following file to your main project directory:

main.h – The embedded-code.com generic global file

## Check Driver Definitions

Check the definitions in each of the following files to see if any need to be adjusted for the microcontroller / processor you are using, and your hardware connections:-

eth-main.h

eth-nic.h

## Timers

You will need to provide some form of timer for the driver. Typically this can be done in your applications general heartbeat timer if you have one. These timers do not have to be interrupt based.

Do the following every 1mS (if DHCP is being used):-

```
//----- NIC DHCP TIMER -----
if (eth_dhcp_1ms_timer)
      eth_dhcp_1ms_timer--;
```

Do the following every 10mS:-

```
//----- ETHERNET GENERAL TIMER -----
ethernet_10ms_clock_timer_working++;
```

Do the following every 1 Second (if DHCP is being used):-

```
//----- NIC DHCP TIMERS -----
if (eth_dhcp_1sec_renewal_timer)
      eth_dhcp_1sec_renewal_timer--;
if (eth_dhcp_1sec_lease_timer)
      eth_dhcp_1sec_lease_timer--;
```

## Application Requirements

In each .c file of your application that will use the driver functions include the following file

eth-main.h

If your application will use manually configured IP settings it needs to do the following function as part of its power-up initialisation:

```
//----- CONFIGURE ETHERNET -----
eth_dhcp_using_manual_settings = 1;
our_ip_address.v[0] = 192;           //MSB
our_ip_address.v[1] = 168;

our_ip_address.v[2] = 0;

our_ip_address.v[3] = 51;            //LSB
our_subnet_mask.v[0] = 255;          //MSB
our_subnet_mask.v[1] = 255;

our_subnet_mask.v[2] = 255;

our_subnet_mask.v[3] = 0;            //LSB
our_gateway_ip_address.v[0] = 192;
our_gateway_ip_address.v[1] = 168;
our_gateway_ip_address.v[2] = 0;
our_gateway_ip_address.v[3] = 1;

//----- SET OUR ETHENET UNIQUE MAC ADDRESS -----
our_mac_address.v[0] = 0x00;  //MSB (This is a generic address - replace
	our_mac_address.v[1] = 0x50;  //with your own globally unique MAC
	our_mac_address.v[2] = 0xC2;  //address for release products)
	our_mac_address.v[3] = 0x50;
	our_mac_address.v[4] = 0x10;
	our_mac_address.v[5] = 0x32;  //LSB

	//----- INITIALISE ETHERNET -----
	tcp_ip_initialise();
```

Or if your application will use DHCP (Dynamic Host Configuration Protocol) to obtain IP settings from a network DHCP server it instead needs to do the following function as part of its power-up initialisation:

```
	//----- CONFIGURE ETHERNET -----
	eth_dhcp_using_manual_settings = 0;

	//----- SET OUR ETHENET UNIQUE MAC ADDRESS -----
	our_mac_address.v[0] = 0x00;  //MSB (This is a generic address - replace
	our_mac_address.v[1] = 0x50;  //with your own globally unique MAC
	our_mac_address.v[2] = 0xC2;  //address for release products)
	our_mac_address.v[3] = 0x50;
	our_mac_address.v[4] = 0x10;
	our_mac_address.v[5] = 0x32;  //LSB

	//----- INITIALISE ETHERNET -----
	tcp_ip_initialise();
```

Your application needs to periodically call the drivers background processing function. Typically this should be done as part of your applications main loop. This function deals with all the background TCP/IP stack processing tasks. Add the following call:-

```
	//----- PROCESS ETHERNET STACK -----
	tcp_ip_process_stack();
```

## MAC Address

Every Ethernet device requires a globally unique MAC (Media Access Control) address. This is the lowest level adaptor address which is used at the level below IP addresses. Blocks of MAC addresses may be officially purchased from the IEEE.

When you purchase a network interface card for say a PC the card will contain a unique MAC address that will have been assigned to it by the manufacturer. However when you use a nic (network interface controller) IC you typically need to tell it what MAC address to use as part of its power up initialisation sequence. Therefore, unless the nic you are using incorporates its own MAC address, or you are using eeprom memory connected to the nic that has been pre-programmed with a unique MAC address, your application will need to provide the means to be assigned a MAC address at manufacture, which is unique to each device manufactured, so that it can pass it to the nic every time it is powered up.

The following MAC addresses are assigned to us and may be used for development purposes:-

00:50:c2:50:10:32

00:50:c2:50:10:33

00:50:c2:50:10:34

00:50:c2:50:10:35

00:50:c2:50:10:36

## IPAddress

The Internet Assigned Numbers Authority (IANA) manages the global IP address space. IPv4 addresses are becoming a precious resource these days, but the use of routers has put off the predicted crises that had been forecast of IP addresses running out. As embedded devices will typically sit behind a router of some sort before any connection to the internet (if they are connected to the internet at all) it is typically the case that IP addresses from the unassigned general use blocks will be used. IP addresses must be unique to every device on a local network, but they do not need to be globally unique as when a router passes packets on to a higher level network (for instance the internet or a corporate network) it will replace the packets sender address with its own IP address and remember which device any response will need to be directed to on the local network, should a response be received. This is called Network Address Translation (NAT) and its why routers are so good at providing a firewall – if an unsolicited packet arrives from the higher level network they have no knowledge of where to sent it on the local network so have to discard it.

The following are IANA reserved private network ranges that you may use on a local network:

Size, Start of range, End of range, Total addresses

24-bit Block 10.0.0.0, 10.255.255.255, 16,777,216

20-bit Block 172.16.0.0 172.31.255.255, 1,048,576

16-bit Block 192.168.0.0, 192.168.255.255, 65,536

## INDIVIDUAL STACK COMPONENTS

### Adding UDP Functionality

For fast non-managed data transfer. Client and server functionality provided by the driver.

Required for some other stack components.

Copy the following files to your main project directory:

eth-udp.c

eth-udp.h

In each .c file of your application that will use the UDP functions include the following file:

eth-udp.h

Check the definitions in the following file to see if any need to be adjusted for your application:

eth-udp.h

**UDP Checksum Header File Option**

UDP checksum field transmission and received packet checking of checksums checksum can be turned off if desired to improve performance. If you application will only communicate over a local network then its safe to turn off the UDP checksum functions as the Ethernet packet CRC (generated by the nic) provides error detection. However if your UDP packets may travel over the internet or some other link, or if you don't care about the extra bit of processing time required, then you should turn the UDP checksum functions on (using the UDP_CHECKSUMS_ENABLED define in eth-udp.h) as there is no guarantee that all the links between source and destination will provide overall packet error checking. Bear in mind that although UDP provides checksum error checking, a packet that contains an error will simply be discarded by the TCP/IP stack and never reach the socket it was intended for (there is no automatic re-try).

### Adding TCP Functionality

For fast connection based managed data transfers, but with a significant overhead. Client and server functionality provided by the driver.

Required for some other stack components.

Copy the following files to your main project directory:

eth-tcp.c

eth-tcp.h

In each .c file of your application that will use the TCP functions include the following file:

eth-tcp.h

Check the definitions in the following file to see if any need to be adjusted for your application:

eth-tcp.h

## Adding HTTP Server Functionality

To provide web server functionality.

Requires TCP to be included.

Copy the following files to your main project directory:

eth-http.c

eth-http.h

In each .c file of your application that will use the HTTP functions include the following file:

eth-http.h

Check the definitions in the following file to see if any need to be adjusted for your application:

eth-http.h

## Adding HTTP Client Functionality

To provide the ability for the driver to connect to a remote http server and download files.  A really useful way of allowing embedded products to access updated information which can be created using normal web publishing tools.

Requires TCP to be included.  The HTTP server does not need to be included.

Copy the following files to your main project directory:

eth-http-client.c

eth-http-client.h

In each .c file of your application that will use the HTTP client functions include the following file:

eth-http-client.h

Check the definitions in the following file to see if any need to be adjusted for your application:

eth-http-client.h

## Adding DHCP Functionality

Not required if you will manually configure the IP address and subnet mask (and gateway address for some stack components). Include DHCP to allow your device to automatically request these settings from a network DHCP server on power-up and periodically when they need to be renewed (if there is no dedicated server on your network typically DSL routers include a DHCP server for simple home or small office networks).

Requires UDP to be included.

Copy the following files to your main project directory:

eth-dhcp.c

eth-dhcp.h

In each .c file of your application that will use the DHCP functions include the following file:

eth-dhcp.h

See the timers section earlier in this manual for details of the timers your application needs to implement for DHCP.

Check the definitions in the following file to see if any need to be adjusted for your application:

eth-dhcp.h

## Adding NetBIOS Functionality

Allows you to find your device using a text based name rather than IP address when trying to connect to it via a local network (not via a router). This can be very useful when using DHCP as you may not know what IP address has been assigned by a DHCP server, but with a NetBIOS name set you can use this to connect to it.

Requires UDP to be included.

Copy the following files to your main project directory:

eth-netbios.c

eth-netbios.h

In each .c file of your application that will use the NetBIOS functions include the following file:

eth-netbios.h

## Adding DNS Functionality

Allows your embedded device to use text URL's to connect to remote devices, such as mail servers or your own remote servers. An important feature to protect against changes in a remote servers IP address effectively stopping your device from connecting to it.

Requires UDP to be included.

Copy the following files to your main project directory:

eth-dns.c

eth-dns.h

In each .c file of your application that will use the DNS functions include the following file:

eth-dns.h

## Adding POP3 Functionality

Automatically check for and download emails from a standard POP3 mailbox. Apart from obvious uses POP3 is a great way to work around strict network administrators in this security conscious world. If your embedded devices will be behind routers or firewalls you can use POP3 to allow your device to retreive messages via the internet, avoiding the need to configure routers or firewalls to allow an incoming connection.

Requires DNS and TCP to be included.

Copy the following files to your main project directory:

eth-pop3.c

eth-pop3.h

In each .c file of your application that will use the POP3 functions include the following file:

eth-pop3.h

Check the definitions in the following file to see if any need to be adjusted for your application:

eth-pop3.h

## Adding SMTP Functionality

Automatically send anyone emails in response to events or at fixed intervals. A file attachment may also be included with each outgoing email using the built in MIME BASE64 encoding.

Requires DNS and TCP to be included.

Copy the following files to your main project directory:

eth-smtp.c

eth-smtp.h

In each .c file of your application that will use the SMTP functions include the following file:

eth-smtp.h

Check the definitions in the following file to see if any need to be adjusted for your application:

eth-smtp.h

## Adding ICMP Functionality

An automatic component that responds to ping requests. Not a requirement but typically included in a TCP/IP Stack.

Copy the following files to your main project directory:

eth-icmp.c

eth-icmp.h

In each .c file of your application that will use the ICMP functions include the following file:

eth-icmp.h

## Adding SNTP Functionality

Allows your embedded device to request the current time (in seconds since a reference date) from public NTP time servers.

Requires UDP to be included.

Copy the following files to your main project directory:

eth-sntp.c

eth-sntp.h

In each .c file of your application that will use the SNTP functions include the following file:

eth-sntp.h

Check the definitions in the following file to see if any need to be adjusted for your application:

eth-sntp.h

## Important Hardware Design Notes

When using a nic good PCB layout is important. Apart from the usual reasons associated with high speed digital designs, reading and writing data to and from a nic is often achieved using fast DMA transfers or clocked serial busses such as SPI. These require clean signals on the Read and Write or Clock pins to the nic, as should noise be present false peeks can either cause the nic pointer to increment to the next memory location, causing additional bytes / words written to the nic or missed bytes / words read from the nic, or SPI transfers to be corrupted.

Please see the:

'Signal Noise Issues With MMC & SD Memory Cards (& Clocked Devices In General)'

page in the resources area of our web site for details of a common problem that may be experienced when using clocked devices of this type to ensure that you are aware of the potential problem and can make sure you avoid it in your hardware design.

# Sample NIC Drivers and Sample Projects

## Overview

Sample nic.c & nic.h network interface controller files are included for several nic's. If using one of these nic's they can be used directly or if using a different nic IC they can be used as a basis for your new device. See later in this manual for details on creating your own version for a new nic. To use them copy all of the files in the chosen nic IC directory into the same directory as the main driver files.

Sample projects are included with the driver for specific devices and compilers. The example schematics at the end of this manual detail the circuit each sample project is designed to work with. You may use the sample projects with the circuit shown or if desired use them as a starting block for your own project with a different device or compiler. To use them copy all of the files in the chosen sample project directory into the same directory as the driver files and then open and run using the development environment / compiler the project was designed with

## Sample NIC Drivers Included

Sample nic files are included for each of the following nic's:-

NXP LPC2000 series 32bit ARM microcontroller

Dual speed 10/100Mps

Built in Ethernet interface with external KSZ8001 PHY IC

Microchip PIC32 series 32bit MIPS microcontroller

Dual speed 10/100Mps

Built in Ethernet interface with external KSZ8001 PHY IC

SMSC LAN91C111

Dual speed 10/100Mps

Parallel 8 or 16 bit interface

Realtek RTL8019AS

Single speed 10Mps

Parallel 8 bit interface

Microchip ENC28J60

Single speed 10Mps

Serial SPI interface

See later in this manual for details on creating your own version for a new nic model.

## Sample Projects Included

### Rowley CrossWorks 2 Compiler

Compiler:

Rowley Associates CrossWorks 2 C Compiler for ARM

Device:

NXP LPC2365

NIC:

Built in. Uses external KSZ8001 PHY IC.

### Microchip C18 Compiler using PICDEM.net 2 Development Board

Compiler:

Microchip C18 MPLAB C Compiler for PIC18 family of 8 bit microcontrollers

Device:

PIC18F97J60

NIC:

Microchip ENC28J60

Notes:

JP5 Link centre to RD2

JP9 Link

Connect Ethernet to J2

### Microchip C30 Compiler using Explorer 16 Development Board & Ethernet PICtail+

Compiler:

Microchip C30 MPLAB C Compiler for PIC24 family of 16 bit microcontrollers and dsPIC digital signal controllers

Device:

PIC24FJ128GA010

NIC:

Microchip ENC28J60

### Microchip C32 Compiler using Explorer 16 Development Board & Ethernet PICtail+

Compiler:

Microchip C32 MPLAB C Compiler for PIC32 family of 32 bit microcontrollers

Device:

PIC32MX360F512L

NIC:

Microchip ENC28J60

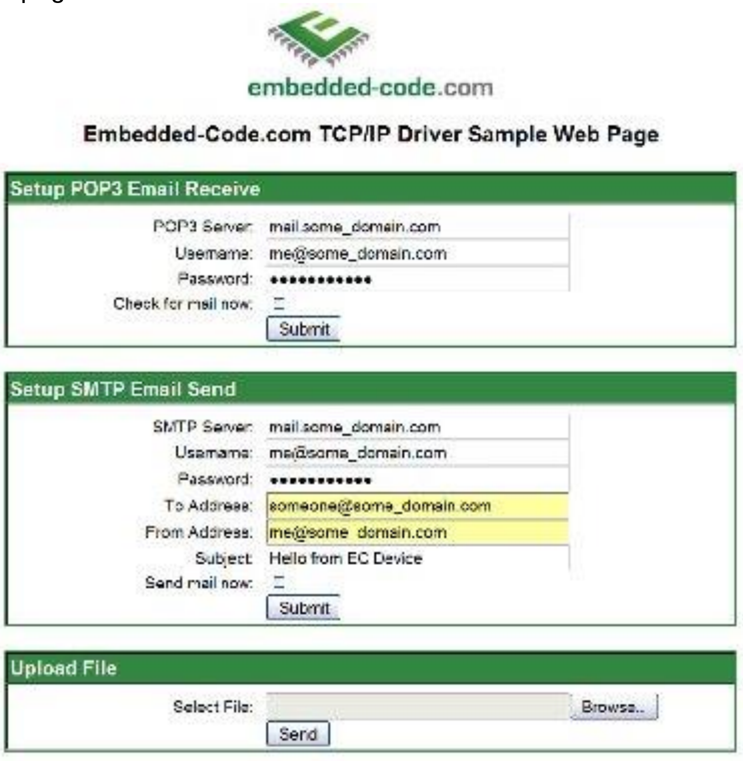
## SAMPLE PROJECT FUNCTIONS

If you haven't already it's a good idea to read the Network Protocol Analyzer Utilities and PC Ethernet Ports sections of this manual now so that you can setup Wireshark to monitor your computers Ethernet port and see activity on the network as it happens. Without this it's hard to see with your own eyes what the stack is actually doing!

The sample project, which is all provided in ap-main.c and ap-main.h, is configured with DHCP enabled. If you wish to manually set an IP address, for instance if your network does not have a DHCP server or if you are directly connecting your device to your computer, alter the initialise() function.

Compile and run the project on your hardware. If the Network Interface Controller IC (nic) is connected to a network (connected to a hub, switch or directly to another Ethernet device) LED A will light. LED B will light when the driver has an IP address. If you are using manual IP settings then this will be immediate but if you are using DHCP this will be once the DHCP server has been found and the IP settings obtained.

To view the sample web page:



embedded-code.com

Embedded-Code.com TCP/IP Driver Sample Web Page

Setup POP3 Email Receive

| | |
|---|---|
| POP3 Server: | mail.some_domain.com |
| Username: | me@some_domain.com |
| Password: | ●●●●●●●●●●● |
| Check for mail now: | ☐ |
| | Submit |

Setup SMTP Email Send

| | |
|---|---|
| SMTP Server: | mail.some_domain.com |
| Username: | me@some_domain.com |
| Password: | ●●●●●●●●●●● |
| To Address: | someone@some_domain.com |
| From Address: | me@some_domain.com |
| Subject: | Hello from EC Device |
| Send mail now: | ☐ |
| | Submit |

Upload File

| | |
|---|---|
| Select File: | Browse.. |
| | Send |

enter the following in your browsers address bar:

"http://embedded-device/", or just "embedded-device"

This is the name the sample project sets as the drivers NetBIOS name. If your browser doesn't display the sample web page shown above and doesn't display a 'not found' error page then its likely that your DNS service is stopping your computer trying the NetBIOS service, so instead enter the IP address of the device in the browsers address bar. If you are using DHCP you can find what IP address the DHCP server assigned by looking at your Wireshark log or by using your debugger to view the value of the variable our_ip_address.

The sample page shown demonstrates the serving of a single web page and an image, the inclusion of dynamic data in each of the form fields, and the three different types of form input methods you may use.

If you have a POP3 or SMTP account to use the web page allows you to trigger the sending or receiving of email. The sample functions that is passed each received email doesn't do anything with the data but can be made to by adding your own code. The sample functions that is called to provide the email contents when triggering an email send will generate a simple email with a text file attached. Triggering a send or receive of email can also be monitored using Wireshark so you can see POP3 and SMTP driver in action.

You can also test the uploading of a file using the third form. Again the sample functions don't do anything with the received file data but can be made to if you add your own code to them.

The 2 switch inputs provide the following functions:-

Switch A Press to trigger the following:

Requests time from SNTP server

Opens a client UDP socket, sends a broadcast UDP packet to port 6452 containing the text "Hello World" and the waits for a response, timing out and resetting if none is received

Switch B Press to trigger the following:

Attempts to opens a client TCP socket connected to a remote device with IP address 192.168.0.20 (alter as required) using port 4102. If the connection is accepted by the remote device it will then sends a TCP packet containing the text "Hello World" and the wait for a response. If a response is received or if it times out it will then request the socket to be closed.

The sample project also implements a constantly running UDP server socket and TCP server socket which provide the following functions:

If you setup another device to send a UDP packet to port 6451 the sample socket will automatically respond with a broadcast packet containing the text "Hello World".

If you setup another device to connect using TCP to 4101 the driver will accept the connection and wait for a single packet. It will automatically respond with a packet containing the text "Hello World". The TCP server will remain connected until the remote client disconnects.

The sample projects may be used as a starting point to write a new application or just as a reference for including the driver in your own project.

# Using The Driver In Your Project

## General Information

### Network Protocol Analyzer Utilities

Anyone undertaking development of TCP/IP projects should download a copy of Wireshark from:

http://www.wireshark.org

This is a very popular free network protocol analyzer which allows you to see exactly what you device is sending and receiving.

A couple of tips when using Wireshark:

Using filter lines such as:

eth.addr == 00:50:c2:50:10:32

can be a very useful way of removing other network traffic captured from the Wireshark screen, so you can concentrate only on the packets being sent to and from your device.

If you are using Wireshark on a PC that is not generating or receiving the packets you want to monitor remember that if you are using an Ethernet Switch it will typically only send packets out of its ports to which a destination device is connected (unless a packet is broadcast). In this situation using an Ethernet Hub instead of a Switch is a really simple way of seeing all of the packets any other device connected to the Hub is seeing. An Ethernet Hub is non intelligent and simply distributes incoming and outgoing packets on a send to everyone basis, whereas an Ethernet Switch only transfers packets to ports that need to receive them.

### PC Ethernet Ports

When developing and testing TCP/IP functions using a computer that has more than one Ethernet connection (e.g. a RJ45 port and a WiFi card) it is often a good idea to disable all but one of them so that you know for sure which port your computer will use and which port to capture packets from when using Wireshark.

## Basic Driver Notes

The eth-main.c and eth-main.h files are the central files of the TCP/IP driver. The central stack operations and definitions are provided in these files. If you want to get a feel for how the stack works at the core level have a quick scan through the tcp_ip_process_stack() function.

On power up your initialisation function needs to call the tcp_ip_initialise() function which initialises the driver and automatically calls any other driver initialisation functions required by the selected driver / stack components. This function also initialises the nic (network interface controller) IC, provides it with its MAC address and enables the receiving of packets.

All your application then needs to do is regularly call the tcp_ip_process_stack() function. Typically this is done as part of an applications main loop. Whilst some the driver / stack components are initiated by the user application (for instance sending email), there are other components that are handled completely automatically by the driver, for instance responding to ARP requests. Therefore the driver needs to be called regularly to check for received packets, transmit automatically generated responses and check for updating of automatic processes such as DHCP.

The stack is designed as a single thread and no stack component is permitted to halt operation in wait states. Every operation that may require a delay or wait is handled by simple state machines so that completion of tasks or delays is checked for and handled each time the tcp_ip_process_stack() function is called. Therefore the stack cannot cause the user application to become halted.

This manual documents all of the features of the driver an how to use them. Due to the very broad nature of TCP/IP and all of its components this manual doesn't attempt to explain in depth how all of the individual protocols work, as these are already widely documented in many books and online resources. However this manual does explain how each of the driver / stack components work in addition to the very comprehensive commenting throughout the source code, should you wish to gain an understanding of it. There is no need to understand the inner workings of the stack though and you can simply use the driver as a plug in module for your overall application, following the instructions in these beginning sections of this manual for simple information on how to configure and use it.

### Some Of The Terminology Used In This Manual

Port

UDP and TCP ports are simply a number that is used in the packet IP header to designate the port a packet has been sent from and the port the packet is being sent to. UDP and TCP have pre defined lists of ports with many being assigned to particular uses. For instance TCP port 80 is defined as being the default port on which to contact a web server. If you design a web server you don't have to use port 80, but it's the default port that a browser will use to try and connect to a web server. As well as all the pre-defined ports there are also lots of undefined ports. Therefore a web browser would send a page request packet 'to' port 80 and would select a port number that it is not already using to send the packet 'from', for instance 3561. When a web server responds it will send its response packet(s) to the port 3561 and from port 80.

Sockets

A socket is simply a term used to describe a connection. Both UDP and TCP sockets are opened by defining a local UDP or TCP port number the socket should use. Once a socket is opened any packets that are received where the IP header 'to' port number matches the port number assigned to an open socket will be passed for processing to the handler function for that socket.

Server & Client

When talking about TCP connections a client is a device that is making a TCP connection to a remote device and a server is a device has a TCP socket listening for connections from a remote device. Therefore don't think of a server as just a typical PC running server software, but rather any device or PC that has a TCP port open listening for an incoming connection. A device can be both client and server at the same time by opening some sockets as client connections and some sockets as listening server connections.

## USING UDP

For more detailed information about how UDP works see the 'How The Driver Works' section later in this manual.

The driver allows your application to create both client and server UDP (User Datagram Protocol) sockets. If your application is only utilising functionality such as DHCP, SNTP, etc then these components of the stack / driver will automatically use UDP without you needing to work directly with it. However if your application needs to communicate directly with another device or PC the UDP examples below provides the means to communicate in an unmanaged way.

Although these examples demonstrate typical functionality well, UDP is a connectionless protocol so you don't have to think in terms of client and server and you don't actually create a socket as one or the other. Once you open a socket you can use it to transmit and receive as you wish, effectively behaving as both a client and a server if required.

### Use UDP Server Sockets To Receive From Remote Clients

Your application may create as many UDP server sockets as it needs from the UDP sockets that are available (the number of available sockets is set by UDP_NO_OF_AVAILABLE_SOCKETS define). When a socket is created nothing is sent, but the driver will automatically pass packets received by the socket to your handler for processing. Once you open a socket you must regularly check for received packets until you close the socket, to avoid the TCP/IP stack stalling while it waits for your handler to remove a received packet.

```
static BYTE our_udp_socket = UDP_INVALID_SOCKET;
static BYTE our_udp_server_state = SM_OPEN_SOCKET;
BYTE data;
BYTE array_buffer[4];

if (!nic_linked_and_ip_address_valid)
{
      //----- WE ARE NOT CONNECTED OR DO NOT YET HAVE AN IP ADDRESS -----
      our_udp_server_state = SM_OPEN_SOCKET;

      //Ensure our socket is closed if we have just lost the Ethernet connection
      udp_close_socket(our_udp_socket);

      return;           //Exit as we can't do anything without a connection
}

switch (our_udp_server_state)
{
case SM_OPEN_SOCKET:
      //----- OPEN SOCKET -----
//(Leave device_info as null to setup to receive from anyone, remote_port
//can be anything for rx)
      our_udp_socket = udp_open_socket(0x00, 6451, 1);
      if (our_udp_socket != UDP_INVALID_SOCKET)
      {
            our_udp_server_state = SM_PROCESS_SOCKET;
            break;
      }
      //Could not open a socket - none currently available - keep trying
      break;

case SM_PROCESS_SOCKET:
      //----- PROCESS SOCKET -----
      if (udp_check_socket_for_rx(our_udp_socket))
      {
            //SOCKET HAS RECEIVED A PACKET - PROCESS IT

            //READ THE PACKET AS REQURIED
            if (!udp_read_next_rx_byte(&data))
            {
                  //Error - no more bytes in rx packet
            }
            //OR USE
            if (!udp_read_rx_array (array_buffer, sizeof(array_buffer)))
            {
                  //Error - no more bytes in rx packet
            }

            //DUMP THE PACKET
            udp_dump_rx_packet();

            //SEND RESPONSE
            our_udp_server_state = SM_TX_RESPONSE;
      }
      break;

case SM_TX_RESPONSE:
      //----- TX RESPONSE -----
      //SETUP TX

      //To respond to the sender leave our sockets remote device info as
//this already contains the remote device settings
      //Or to broadcast on our subnet do this:
      udp_socket[our_udp_socket].remote_device_info.ip_address.val =
our_ip_address.val | ~our_subnet_mask.val;
      udp_socket[our_udp_socket].remote_device_info.mac_address.v[0] = 0xff;
      udp_socket[our_udp_socket].remote_device_info.mac_address.v[1] = 0xff;
      udp_socket[our_udp_socket].remote_device_info.mac_address.v[2] = 0xff;
      udp_socket[our_udp_socket].remote_device_info.mac_address.v[3] = 0xff;
      udp_socket[our_udp_socket].remote_device_info.mac_address.v[4] = 0xff;
      udp_socket[our_udp_socket].remote_device_info.mac_address.v[5] = 0xff;
      udp_socket[our_udp_socket].remote_port = 6450;
      udp_socket[our_udp_socket].local_port = 6451;

      if (!udp_setup_tx(our_udp_socket))
      {
            //Can't tx right now - try again next time

            //Return the socket back to broadcast ready to receive from anyone
again
//Only enable the line below if you are broadcasting responses and
//don't want to miss incoming packets to this socket from other devices
            //udp_socket[our_udp_socket].remote_device_info.ip_address.val =
                  0xffffffff;

            break;
      }

      //WRITE THE UDP DATA
      udp_write_next_byte('H');
      udp_write_next_byte('i');
      udp_write_next_byte(0x00);
      //You can also use udp_write_array()

      //SEND THE PACKET
      udp_tx_packet();

      //RETURN THE SOCKET BACK TO BROADCAST READY TO RECEIVE FROM ANYONE AGAIN
      udp_socket[our_udp_socket].remote_device_info.ip_address.val = 0xffffffff;

      our_udp_server_state = SM_PROCESS_SOCKET;
      break;

} //switch (our_udp_server_state)
```

## Use UDP Client Sockets To Transmit To Remote Servers

Your application may create as many UDP client sockets as it needs from the UDP sockets that are available (the number of available sockets is set by UDP_NO_OF_AVAILABLE_SOCKETS define). Before a socket can be created you need to determine the IP and MAC address of the device you wish to connect to (use ARP if you need to determine the MAC address), or you may instead transmit to a broadcast address without using ARP. The UDP driver will then send or receive data as required.

The following is a full example of how your application can create and process a client socket:

```
static BYTE our_udp_socket = UDP_INVALID_SOCKET;
static BYTE our_udp_client_state = SM_OPEN_SOCKET;
static DEVICE_INFO remote_device_info;
BYTE data;
BYTE array_buffer[4];

if (!nic_linked_and_ip_address_valid)
{
      //----- WE ARE NOT CONNECTED OR DO NOT YET HAVE AN IP ADDRESS -----
      our_udp_client_state = SM_IDLE;

      //Ensure our socket is closed if we have just lost the Ethernet connection
      udp_close_socket(our_udp_socket);
      return;          //Exit as we can't do anything without a connection
}

switch (our_udp_client_state)
{
case SM_IDLE:
      //----- DO NOTHING -----
      break;

case SM_OPEN_SOCKET:
      //----- OPEN SOCKET -----

      //Set to broadcast on our subnet (alternatively set the IP and MAC address
//to a remote devices address - use ARP first if the MAC address is unknown)
      remote_device_info.ip_address.val = our_ip_address.val |
~our_subnet_mask.val;
      remote_device_info.mac_address.v[0] = 0xff;
      remote_device_info.mac_address.v[1] = 0xff;
      remote_device_info.mac_address.v[2] = 0xff;
      remote_device_info.mac_address.v[3] = 0xff;
      remote_device_info.mac_address.v[4] = 0xff;
      remote_device_info.mac_address.v[5] = 0xff;

      //Set the port numbers as desired
      our_udp_socket = udp_open_socket(&remote_device_info, (WORD)6453,
(WORD)6452);
      if (our_udp_socket != UDP_INVALID_SOCKET)
      {
            our_udp_client_state = SM_TX_PACKET;
            break;
      }
      //Could not open a socket - none currently available - keep trying
      break;

case SM_TX_PACKET:
      //----- TX PACKET TO REMOTE DEVICE -----
      //SETUP TX
      if (!udp_setup_tx(our_udp_socket))
      {
            //Can't tx right now - try again next time
            break;
      }

      //WRITE THE TCP DATA
      udp_write_next_byte('H');
      udp_write_next_byte('i');
      udp_write_next_byte(0x00);
      //You can also use udp_write_array()

      //SEND THE PACKET
      udp_tx_packet();

      udp_client_socket_timeout_timer = 10;
      our_udp_client_state = SM_WAIT_FOR_RESPONSE;
      break;

case SM_WAIT_FOR_RESPONSE:
      //----- WAIT FOR RESPONSE -----
      if (udp_check_socket_for_rx(our_udp_socket))
      {
            //SOCKET HAS RECEIVED A PACKET - PROCESS IT

            //READ THE PACKET AS REQURIED
```

```
			if (!udp_read_next_rx_byte(&data))
			{
				//Error - no more bytes in rx packet
			}
			//OR USE
			if (!udp_read_rx_array (array_buffer, sizeof(array_buffer)))
			{
				//Error - no more bytes in rx packet
			}

			//DUMP THE PACKET
			udp_dump_rx_packet();

			//EXIT
			our_udp_client_state = SM_CLOSE_SOCKET;
		}

		if (udp_client_socket_timeout_timer == 0)
		{
			//TIMED OUT - NO RESPONSE FROM REMOTE DEVICE
			our_udp_client_state = SM_CLOSE_SOCKET;
		}
		break;

	case SM_CLOSE_SOCKET:
		//----- CLOSE THE SOCKET -----
		udp_close_socket(our_udp_socket);

		our_udp_client_state = SM_IDLE;
		break;

	} //switch (our_udp_client_state)
```

## Getting Packet Sender Data When A Packet Is Received

If you want to know the IP or MAC address of the remote device a packet has been received from they are available in the following variables:

```
sender_ip_addr.val =
udp_socket[our_udp_socket].remote_device_info.ip_address.val;
sender_mac_addr.v[0] =
udp_socket[our_udp_socket].remote_device_info.mac_address.v[0];
sender_mac_addr.v[1] =
udp_socket[our_udp_socket].remote_device_info.mac_address.v[1];
sender_mac_addr.v[2] =
udp_socket[our_udp_socket].remote_device_info.mac_address.v[2];
sender_mac_addr.v[3] =
udp_socket[our_udp_socket].remote_device_info.mac_address.v[3];
sender_mac_addr.v[4] =
udp_socket[our_udp_socket].remote_device_info.mac_address.v[4];
sender_mac_addr.v[5] =
udp_socket[our_udp_socket].remote_device_info.mac_address.v[5];
```

If you need to know if received packet was broadcast or sent to our unique IP address:

```
sender_ip_address_packet_was_sent_to =
udp_socket[our_udp_socket].destination_ip_address.val;
```

## Designing your own protocol

If you are designing your own UDP protocol and will be transferring large amounts of data remember to bear in mind the available receive buffer space in the network interface controller (nic) IC you are using. As packets are received, either to the device directly or broadcast, they will be stored in the nic's receive buffer until your application has read and discarded them. Therefore if you could potentially overfill the nic's receive buffer your communication protocol should be designed so that only a limited amount of data is sent before an acknowledge response is required or a delay is inserted, or so that there is sufficient time between groups of packets to allow them to have been processed by the receiving devices. If you don't and the nic's receive buffer becomes full it will simply dump packets it receives until there is space available again.

### Multiple sockets using same local port number

You may open more than 1 socket with the same local port number, to allow multiple remote devices to connect to a single port number. When a new UDP packet is received the handler looks for the first used UDP socket using that port number that is set to receive from the remote device the packet is from or set to receive from any device. If one is found the packet is passed to that socket.

If your application returns a socket back to the receive from anyone state using:

```
	udp_socket[socket].remote_device_info.ip_address.val = 0xffffffff;
```

as soon as any packet is received to that socket (i.e. as part of the receive handler), then there is no need to create additional sockets using the same port number as that socket will always accept incoming packets from any sender to that port.

### Viewing The State Of The UDP Sockets

The array udp_socket contains the current status and all the variables associated with each of the UDP sockets that are available to the stack. Every process that opens a UDP socket is given a BYTE socket number, which is the pointer into the udp_socket array. It can sometimes be useful to view the array in a watch window when debugging.

## USING TCP

For more detailed information about how TCP works see the 'How The Driver Works' section later in this manual.

The driver allows your application to create both client and server TCP (Transmission Control Protocol) sockets. If your application is only utilising functionality such as email, HTTP server, etc then these parts of the stack / driver will automatically use TCP without you needing to work directly with it. However if your application needs to communicate directly with another device or PC the TCP examples below provides the means to communicate in a managed way.

### Create TCP Server Sockets To Accept A Connection From A Remote Client

Your application may create as many TCP server sockets as it needs from the TCP sockets that are available (the number of available sockets is set by the TCP_NO_OF_AVAILABLE_SOCKETS define). When a socket is created nothing is sent, but the driver will automatically accept connection requests to the socket and then pass packets received by the socket to your handler for processing. Once you open a socket you must regularly check for received packets or retransmissions until you close the socket, to avoid the TCP/IP stack stalling while it waits for your handler to remove a received packet.

The following is a full example of how your application can create and process a server socket:

```
static BYTE our_tcp_server_socket = TCP_INVALID_SOCKET;
static BYTE our_tcp_server_state = SM_OPEN_SOCKET;
BYTE data;
BYTE array_buffer[4];

if (!nic_linked_and_ip_address_valid)
{
	//----- WE ARE NOT CONNECTED OR DO NOT YET HAVE AN IP ADDRESS -----
	our_tcp_server_state = SM_OPEN_SOCKET;

	//Ensure our socket is closed if we have just lost the Ethernet connection
	tcp_close_socket_from_listen(our_tcp_server_socket);

	return;			//Exit as we can't do anything without a connection
```

```
}

switch (our_tcp_server_state)
{
case SM_OPEN_SOCKET:
      //----- OPEN SOCKET -----
//We will listen on port 4101 (change as required)

//We shouldn't have a socket currently, but make sure
      if (our_tcp_server_socket != TCP_INVALID_SOCKET)
            tcp_close_socket(our_tcp_server_socket);

our_tcp_server_socket = tcp_open_socket_to_listen(4101);

if (our_tcp_server_socket != TCP_INVALID_SOCKET)
      {
            our_tcp_server_state = SM_WAIT_FOR_CONNECTION;
            break;
      }
      //Could not open a socket - none currently available - keep trying
      break;

case SM_WAIT_FOR_CONNECTION:
      //----- WAIT FOR A CLIENT TO CONNECT -----
      if(tcp_is_socket_connected(our_tcp_server_socket))
      {
            //A CLIENT HAS CONNECTED TO OUR SOCKET
            our_tcp_server_state = SM_PROCESS_CONNECTION;

//Set our client has been lost timeout (to avoid client
//disappearing and causing this socket to never be closed)
            tcp_server_socket_timeout_timer = 10;
      }
      break;

case SM_PROCESS_CONNECTION:
      //----- PROCESS CLIENT CONNECTION -----

      if (tcp_server_socket_timeout_timer == 0)
      {
            //THERE HAS BEEN NO COMMUNICATIONS FROM CLIENT TIMEOUT
//RESET SOCKET AS WE ASSUME CLIENT HAS BEEN LOST
            tcp_close_socket(our_tcp_server_socket);
//As this socket is a server the existing connection will be closed
//but the socket will be reset to wait for a new connection (use
//tcp_close_socket_from_listen if you want to fully close it)
            our_tcp_server_state = SM_WAIT_FOR_CONNECTION;
      }

      if (tcp_check_socket_for_rx(our_tcp_server_socket))
      {
            //SOCKET HAS RECEIVED A PACKET - PROCESS IT
            tcp_server_socket_timeout_timer = 10;     //Reset our timeout timer

            //READ THE PACKET AS REQURIED
            if (tcp_read_next_rx_byte(&data) == 0)
            {
                  //Error - no more bytes in rx packet
            }
            //OR USE
            if (tcp_read_rx_array (array_buffer, sizeof(array_buffer)) == 0)
            {
                  //Error - no more bytes in rx packet
            }
            //DUMP THE PACKET
            tcp_dump_rx_packet();

            //SEND RESPONSE
            our_tcp_server_state = SM_TX_RESPONSE;
      }

      if (tcp_does_socket_require_resend_of_last_packet(our_tcp_server_socket))
      {
            //RE-SEND LAST PACKET TRANSMITTED
            //(TCP requires resending of packets if they are not acknowledged
and to
            //avoid requiring a large RAM buffer the application needs to
remember
            //the last packet sent on a socket so it can be resent if required).
            our_tcp_server_state = SM_TX_RESPONSE;
      }

      if(!tcp_is_socket_connected(our_tcp_server_socket))
      {
            //THE CLIENT HAS DISCONNECTED
            our_tcp_server_state = SM_WAIT_FOR_CONNECTION;
      }

      break;

case SM_TX_RESPONSE:
      //----- TX RESPONSE -----
      if (!tcp_setup_socket_tx(our_tcp_server_socket))
      {
            //Can't tx right now - try again next time
            break;
      }

      //WRITE THE TCP DATA
      tcp_write_next_byte('H');
      tcp_write_next_byte('i');
      tcp_write_next_byte(0x00);
      //You can also use tcp_write_array()

      //SEND THE PACKET
      tcp_socket_tx_packet(our_tcp_server_socket);

      our_tcp_server_state = SM_PROCESS_CONNECTION;
      break;
}
```

If you want to know the IP or MAC address of the remote client they are available in the following variables:

```
sender_ip_addr.val =
tcp_socket[our_tcp_server_socket].remote_device_info.ip_address.val;
sender_mac_addr.v[0] =
tcp_socket[our_tcp_server_socket].remote_device_info.mac_address.v[0];
sender_mac_addr.v[1] =
tcp_socket[our_tcp_server_socket].remote_device_info.mac_address.v[1];
sender_mac_addr.v[2] =
tcp_socket[our_tcp_server_socket].remote_device_info.mac_address.v[2];
sender_mac_addr.v[3] =
tcp_socket[our_tcp_server_socket].remote_device_info.mac_address.v[3];
sender_mac_addr.v[4] =
tcp_socket[our_tcp_server_socket].remote_device_info.mac_address.v[4];
sender_mac_addr.v[5] =
tcp_socket[our_tcp_server_socket].remote_device_info.mac_address.v[5];
```

## Create TCP Client Sockets To Connect To A Remote TCP Server Socket

Your application may create as many TCP client sockets as it needs from the TCP sockets that are available (the number of available sockets is set by TCP_NO_OF_AVAILABLE_SOCKETS). The TCP driver will then attempt to connect to the specified port on the remote device, using ARP first if the MAC address is unknown. Once connected you may then send or receive data before you or the remote device close the connection.

The following is a full example of how your application can create and process a client socket:

```
static BYTE our_tcp_client_socket = TCP_INVALID_SOCKET;
static WORD our_tcp_client_local_port;
static BYTE our_tcp_client_state = SM_OPEN_SOCKET;
static DEVICE_INFO remote_device_info;
BYTE data;
BYTE array_buffer[4];

if (!nic_linked_and_ip_address_valid)
{
      //----- WE ARE NOT CONNECTED OR DO NOT YET HAVE AN IP ADDRESS -----
      our_tcp_client_state = SM_OPEN_SOCKET;

      //Ensure our socket is closed if we have just lost the Ethernet connection
      tcp_close_socket(our_tcp_client_socket);

      return;           //Exit as we can't do anything without a connection
}

if (our_tcp_client_socket != TCP_INVALID_SOCKET)
{
      //----- CHECK OUR CLIENT SOCKET HASN'T DISCONNECTED -----
      if ((tcp_socket[our_tcp_client_socket].sm_socket_state == SM_TCP_CLOSED)
|| (tcp_socket[our_tcp_client_socket].local_port != our_tcp_client_local_port))
            //If the local port has changed then our socket was closed and taken
by some other application process since we're last called
            our_tcp_client_socket = TCP_INVALID_SOCKET;

      if (our_tcp_client_state == SM_WAIT_FOR_DISCONNECT)
            our_tcp_client_state = SM_OPEN_SOCKET;
      else if (our_tcp_client_state != SM_OPEN_SOCKET)
            our_tcp_client_state = SM_COMMS_FAILED;
}


switch (our_tcp_client_state)
{
case SM_OPEN_SOCKET:
      //----- OPEN SOCKET -----
      remote_device_info.ip_address.v[0] = 192; //The IP address of the remote
      remote_device_info.ip_address.v[1] = 168; //device we want to connect
      remote_device_info.ip_address.v[2] = 0;   //to (change as required)
      remote_device_info.ip_address.v[3] = 20;
      remote_device_info.mac_address.v[0] = 0;  //Set to zero so TCP
      remote_device_info.mac_address.v[1] = 0;  //will automatically use ARP
      remote_device_info.mac_address.v[2] = 0;  //to find MAC address
      remote_device_info.mac_address.v[3] = 0;
      remote_device_info.mac_address.v[4] = 0;
      remote_device_info.mac_address.v[5] = 0;

      //Connect to remote device port 4102 (the port it is listening on - change
//as required)
```

```
//We shouldn't have a socket currently, but make sure
      if (our_tcp_client_socket != TCP_INVALID_SOCKET)
                  tcp_close_socket(our_tcp_client_socket);

      our_tcp_client_socket = tcp_connect_socket(&remote_device_info, 4102);
      if (our_tcp_client_socket != TCP_INVALID_SOCKET)
      {
            our_tcp_client_local_port =
tcp_socket[our_tcp_client_socket].local_port;
//Set our wait for connection timeout
            tcp_client_socket_timeout_timer = 10;

            our_tcp_client_state = SM_WAIT_FOR_CONNECTION;
            break;
      }
      //Could not open a socket - none currently available - keep trying

      break;

case SM_WAIT_FOR_CONNECTION:
      //----- WAIT FOR SOCKET TO CONNECT -----
      if (tcp_is_socket_connected(our_tcp_client_socket))
            our_tcp_client_state = SM_TX_PACKET;

      if (tcp_client_socket_timeout_timer == 0)
      {
            //CONNECTION REQUEST FAILED
            our_tcp_client_state = SM_COMMS_FAILED;
      }

      break;

case SM_TX_PACKET:
      //----- TX PACKET TO REMOTE DEVICE -----
      if (!tcp_setup_socket_tx(our_tcp_client_socket))
      {
            //Can't tx right now - try again next time
            break;
      }

      //WRITE THE TCP DATA
      tcp_write_next_byte('H');
      tcp_write_next_byte('i');
      tcp_write_next_byte(0x00);
      //You can also use tcp_write_array()

      //SEND THE PACKET
      tcp_socket_tx_packet(our_tcp_client_socket);

//Set our wait for response timeout
      tcp_client_socket_timeout_timer = 10;

our_tcp_client_state = SM_WAIT_FOR_RESPONSE;
      break;

case SM_WAIT_FOR_RESPONSE:
      //----- WAIT FOR RESPONSE -----

      if (tcp_client_socket_timeout_timer == 0)
      {
            //WAIT FOR RESPOSNE TIMEOUT
            tcp_close_socket(our_tcp_client_socket);
            our_tcp_client_state = SM_COMMS_FAILED;
      }
      if (tcp_check_socket_for_rx(our_tcp_client_socket))
      {
            //SOCKET HAS RECEIVED A PACKET - PROCESS IT

            //READ THE PACKET AS REQURIED
            if (tcp_read_next_rx_byte(&data) == 0)
            {
                  //Error - no more bytes in rx packet
            }
            //OR USE
            if (tcp_read_rx_array (array_buffer, sizeof(array_buffer)) == 0)
            {
                  //Error - no more bytes in rx packet
            }

            //DUMP THE PACKET
            tcp_dump_rx_packet();

            our_tcp_client_state = SM_REQUEST_DISCONNECT;
      }

      if (tcp_does_socket_require_resend_of_last_packet(our_tcp_client_socket))
      {
            //RE-SEND LAST PACKET TRANSMITTED
            //(TCP requires resending of packets if they are not acknowledged
and to
            //avoid requiring a large RAM buffer the application needs to
remember
            //the last packet sent on a socket so it can be resent if requried).
            our_tcp_client_state = SM_TX_PACKET;
      }

      if(!tcp_is_socket_connected(our_tcp_client_socket))
      {
            //THE CLIENT HAS DISCONNECTED
            our_tcp_client_state = SM_COMMS_FAILED;
      }
      break;

case SM_REQUEST_DISCONNECT:
      //----- REQUEST TO DISCONNECT FROM REMOTE SERVER -----
      tcp_request_disconnect_socket (our_tcp_client_socket);

//Set our wait for disconnect timeout
      tcp_client_socket_timeout_timer = 10;

our_tcp_client_state = SM_WAIT_FOR_DISCONNECT;
      break;

case SM_WAIT_FOR_DISCONNECT:
      //----- WAIT FOR SOCKET TO BE DISCONNECTED -----

      if (tcp_is_socket_closed(our_tcp_client_socket))
      {
            our_tcp_client_state = SM_COMMS_COMPLETE;
      }

      if (tcp_client_socket_timeout_timer == 0)
      {
            //WAIT FOR DISCONNECT TIMEOUT
//Force the socket closed at our end
            tcp_close_socket(our_tcp_client_socket);
            our_tcp_client_state = SM_COMMS_FAILED;
```

```
		}
		break;

	case SM_COMMS_COMPLETE:
		//----- COMMUNICATIONS COMPLETE -----
		break;

	case SM_COMMS_FAILED:
		//----- COMMUNICATIONS FAILED -----
		break;
	}
```

### Multiple sockets using same local port number

You may open more than 1 server socket with the same local port number, to allow multiple remote devices to connect to a single port number. In this case when a new TCP packet is received the handler looks to see if the remote device is already connected to any of the local TCP sockets. If it is the packet is passed to that socket. If not the packet is passed to the first open socket that is not currently connected to a client.

### Viewing The State Of The TCP Sockets

The array tcp_socket contains the current status and all the variables associated with each of the TCP sockets that are available to the stack. Every process that opens a TCP socket is given a BYTE socket number, which is a pointer into the tcp_socket array. It can be useful to view the array in a watch window when debugging.

## USING HTTP SERVER

For more detailed information about how HTTP works see the 'How The Driver Works' section later in this manual.

The driver includes a very powerful embedded HTTP (Hypertext Transfer Protocol) server which may be used to provide simple web pages, pages with dynamically generated data and pages that allow a user to edit settings, enter data and upload files. The 'Generating HTTP Web Content' section later in this manual contains further information on designing your web files.

### Storing Files In Program Memory

The HTTP_USING_C_FILES define should be un-commented for this file storage option.

The simplest (and fastest) method of including your html, image and any other web files is to use the web pages converter application to automatically output them all as a single C header file. The outputted html_c.h file should then be copied into your project directory and it contains all of the individual files that we're converted. You don't need to do anything more.

### Storing Files In External Memory

The HTTP_USING_BINARY_FILES define should be un-commented for this file storage option.

If you don't have the program memory space available or maybe you want to use cheaper lower cost external flash memory to store your html, image and any other web files you can use the web pages converter application to automatically output them all as a single binary data block file called html_bin.bin together with a C header file called html_bin.h The .h file contains all of the filenames and their address within the data block html_bin.bin. This allows the HTTP driver to automatically find files as they are requested. However you need to provide two functions:

> You will need to implement the means to transfer the contents of the html_bin.bin file into your flash memory.
>
> You also need to provide the function defined as HTTP_BINARY_FILE_NEXT_BYTE in eth-http.h. The function will be called with the address of the next byte that is required by the HTTP driver and it must retrieve the byte from the flash memory and return it

## Storing Files Using An External Filing System

The HTTP_USING_FILING_SYSTEM define should be un-commented for this file storage option.

If you are using a filing system in your application (for instance one of the embedded-code.com FAT drivers) you can alternatively store your html, image and any other web files using this file system. This option is slightly harder for you to implement, as your own functions must located requested files and return specified bytes from each file on demand to the HTTP driver. However this approach allows you to easily change the content of the files the HTTP driver serves. You need to provide these functions:

> You will need to implement the means to transfer all of the files into your filing system. If using a removable memory card this is obviously as simple as copying the files onto it using a PC.
>
> You need to provide the function defined as HTTP_EXTERNAL_FILE_FIND_FILE in eth-http.h. The function is called each time a new request is received from a HTTP client and it needs to search the available files and return the file size and a pointer / address of the first byte if the file is found.
>
> You also need to provide the function defined as HTTP_EXTERNAL_FILE_NEXT_BYTE in eth-http.h. This function will be called with the pointer / address of the next byte that is required by the HTTP driver and it must retrieve the byte from the filing system and return it.

Note that if you don't fancy implementing functions to handle providing the individual web content files you could instead use the 'Storing Files In External Memory' option with an external filing system. In this case you would simply store the html_bin.bin file that is output by the Web Pages Converter Application in your filing system / on your memory volume. If you do this there is only a single file to deal with and the driver will automatically work out the address of each byte within this binary file that it needs. You would just implement the function defined as HTTP_BINARY_FILE_NEXT_BYTE in eth-http.h to return the requested byte from this single file.

## Serving Basic Web Pages

To serve basic web pages you don't need to do anything more. As pages and their linked files are requested the HTTP driver will automatically find and return them without any intervention required by the user application.

## Providing Dynamic Data

To provide dynamically generated content within your web pages, that is values, images, text etc that is added to the page as it is sent to a client, the driver provides a very simply yet powerful dynamic data function. As a .htm file is read by the HTTP driver and added to a TCP packet it looks for the tilde '~' character, which acts as a special marker. When this character is found the driver doesn't add it to the TCP packet but instead continues reading the following variable name until it gets to a trailing hyphen '-' character. This variable name is then passed to the optional function in your application which you define with the HTTP_DYNAMIC_DATA_FUNCTION define in eth-http.h. Your function should then return a null terminated string which the driver will send before carrying on with the rest of the .htm file. Further details can be found in the 'Generating HTTP Web Content' section later in this manual.

This simple approach is actually incredibly powerful. As an HTML file is simply an ASCII text file you are not limited to adding dynamically generated text that the user sees, but you can actually dynamically add anything you wish to a file as it is sent, for instance selecting which image to display, adding hidden script code or values, etc. The only limit is a maximum of 100 characters per variable name.

In addition to the variable name the TCP socket ID is also passed to your applications function, allowing it to identify a user by their unique MAC or IP address if desired.

See the eth-http.h file for full details of the function you need to include.

## Receiving Inputs From Clients

Being able to generate dynamic data is great but in many embedded applications the user needs to be able to alter settings or enter values. The HTTP driver provides a complete set of input handling capabilities. The 'Sending Data To Your Embedded Http Server' section later in this manual provides details of the standard GET and POST input methods that may be used in your web pages to provide one or more input values from a html form or with a page request. As each input is received from a client the HTTP driver will call the optional function in your application which you define with HTTP_PROCESS_INPUT_FUNCTION in eth-http.h. The call will include the input name string (i.e. the name of the item on a form), the value entered or selected, the filename that is being requested, and the TCP socket number in case you need to identify the client by their unique MAC or IP address. Your function may alter the filename being requested if required so that the a different file is returned to the client, which can be useful for instance when asking users to log in so that they can be re-directed to a login success page if the details they entered are correct.

Whenever a client sends a request to the HTTP driver that includes inputs, each of the inputs will be passed to your applications function before the requested file is returned. This allows your application to update values as necessary which then may be included in the page sent back to the client, for instance confirming the settings they have just changed.

See the eth-http.h file for full details of the function you need to include.

The included sample project web page ‘Setup POP3’ section is a working example of a GET form.

The included sample project web page ‘Setup SMTP’ section is a working example of a POST “application/x-www-form-urlencoded” form.

## Receiving Data Blocks & Files From Clients

One piece of functionality that is often missing or not fully implemented in embedded HTTP drivers is the ability to receive multipart form data using the ‘multipart/form-data’ type of POST request. The simpler ‘application/x-www-form-urlencoded’ POST request is inefficient for sending large quantities of binary data or text containing non-ASCII characters. The content type ‘multipart/form-data’ is used instead to solve these problems and is also the standard method supported by all mainstream browsers to allow files to be uploaded by a user (the PUT request, whilst part of the HTTP specification, is not generally supported by browsers).

The ‘Sending Data To Your Embedded Http Server’ section later in this manual provides further details of the POST input methods.

The HTTP driver provides simple but effective handling of this type of POST request, and decodes the multipart data before passing it to the user application, so that the application gets the data exactly as submitted by a user.

When a multipart/form-data POST request is received the HTTP driver will call the following functions in your application to receive the input data. Each of these optional functions needs to be defined in eth-http.h.

HTTP_POST_MULTIPART_HEADER_FUNCTION

> This function is called with each header found for a new multipart section of the input request (a request may contain one or more sections, each of which relates to an individual input). It will be called 1 or more times, and it signifies that when HTTP_POST_MULTIPART_NEXT_BYTE_FUNCTION is next called it will be with the data for this new section of the multipart message (i.e. any call to this function means your application is about to receive data for a new multipart section, so reset whatever your application needs to reset to start dealing with the new data).

HTTP_POST_MULTIPART_NEXT_BYTE_FUNCTION

> This function is called with each decoded byte of a multipart section. The data you get here is the same as the data submitted by the user (the driver deals with all decoding).

HTTP_POST_LAST_MULTIPART_DONE_FUNCTION

> This function is called after the last byte has been received for a multipart section, to allow your application to carry out any operations required with the data just received.

See the eth-http.h file for full details of the functions you need to include.

The included sample project web page ‘Upload File’ section is a working example of a POST “multipart/form-data” form.

### Authorising Users
You may optionally include define HTTP_AUTHORISE_REQUEST_FUNCTION in eth-http.h. If you do this function in your application will be called each time a request is received from an HTTP client. Your function can then check the senders IP address or MAC address and optionally block the request from being serviced. Alternatively your function can also check the filename that is being requested and optionally modify it to re-direct the request to a different file. This is a great feature for applications where only certain users are permitted access or where users must log in before they can access certain files.

See the eth-http.h file for full details of the function you need to include.

### HTTP 1.1 compliance
This driver generally complies with HTTP 1.1 apart from the following:

> Chunked Transfer Coding is a requirement of HTTP 1.1 but is not supported by this driver. Chunked encoding allows a client to modify the body of a message in order to transfer it as a series of chunks, each with its own size indicator. The main purpose of this is to allow dynamically produced content to be transferred. This is not typically a requirement of an embedded application and to avoid a large amount of additional code space being required to handle this it is not supported.

## USING HTTP CLIENT

The driver includes a really useful HTTP client fucntion.  This allows the driver to connect to a remote HTTP server (a typical web server) and download individual files.  This is a great way of allowing embedded devices to access up to date information via the internet which may be uploaded and updated using standard web publishing tools or web services.  The files downaloded can be any type of file, not just html files, allowing your device to periodically updated it's own media files for instance.

### Requesting A File From A HTTP Server
Call the start_http_client_request function:

```
CONSTANT BYTE REMOTE_HTTP_SERVER_STRING[] = {"www.somedomain.com"};
CONSTANT BYTE REMOTE_HTTP_FILENAME_STRING[] = {"devices/file01.html"};

      if (start_http_client_request(REMOTE_HTTP_SERVER_STRING,
REMOTE_HTTP_FILENAME_STRING))
      {
            //The request was started.
            //The HTTP_CLIENT_REQUEST_RECEIVE_FUNCTION function will deal with
receiving the response if sucessful
      }
```

### Processing Received HTTP Files
Define this with your applications function that will receive the files from the http server:

HTTP_CLIENT_REQUEST_RECEIVE_FUNCTION

Your function definition needs to be:

```
void my_function_name (BYTE op_code, DWORD content_length, BYTE
*requested_host_url, BYTE *requested_filename)
```

op_code

> 0 = error – http client failed.
>
> 1 = OK.  First section of TCP file data ready to be read.  Function should use the tcp_read_next_rx_byte or tcp_read_rx_array functions to read all of the data from this TCP packet before returning.
>
> 2 = Next section of TCP file data ready to be read.  Function should use the tcp_read_next_rx_byte or tcp_read_rx_array functions to read all of the data from this TCP packet before returning.

0xff = The remote server has closed the connection (this will mark the end of the file if content-length was not provided by the server.

content_length

The file length specified by the server at the start of the response.  Note that the server is not requried to specify this (and many don't) and in this instance the value will be 0xffffffff;

requested_host_url

Pointer to a null terminated string containing the host url that was originally requested

requested_filename

Pointer to a null terminated string containing the filename that was originally requested

Read each data byte in the packet using:

```
    while (tcp_read_next_rx_byte(&my_byte_variable))
    {
```

This function is always called after a request, either to either indicate the request failed, or with 1 or more packets of file data. If the request was sucessful the file data packets will be received in the correct order and the tcp data may simply be stored as it is read. The HTTP Client get request headers specify that no encoding may be used by the remote server so the file will be received exactly as it is stored on the server.

An example function

```
//**********************************************
//**********************************************
//********** PROCESS HTTP CLIENT REQUEST **********
//**********************************************
//**********************************************
void process_http_client_request (BYTE op_code, DWORD content_length, BYTE
*requested_host_url, BYTE *requested_filename)
{
      static DWORD byte_id;
      BYTE data;

      if (op_code == 0)
      {
            //--------------------------
            //----- REQUEST FAILED -----
            //--------------------------

            return;
      }
      else if (op_code == 0xff)
      {
            //-----------------------------------------------
            //----- REMOTE SERVER HAS CLOSED CONNECTION -----
            //-----------------------------------------------
            //(This will mark the end of the file if content-length was not
provided by the server)

            return;
      }
      else if (op_code == 1)
      {
            //-------------------------------------
            //----- FIRST PACKET OF FILE DATA -----
            //-------------------------------------
            byte_id = 0;

        }

        //-------------------------
        //----- READ THE DATA -----
        //-------------------------
        while (tcp_read_next_rx_byte(&data))
        {
            //----- GOT NEXT BYTE -----
            //data has the next byte

            byte_id++;
            if (byte_id >= content_length)
            {
                //----- WE HAVE READ ALL OF THE FILE DATA -----

            }
        }
    }
```

## USING ARP

For more detailed information about how ARP works see the 'How The Driver Works' section later in this manual.

ARP (Address Resolution Protocol) is the method used to discover a devices hardware MAC address when only its IP address is known. An IP address is not fixed and may be assigned to a device and later changed. A MAC address however is unique and assigned to an Ethernet interface at manufacture. Although IP addresses are used to communicate between devices, it is the MAC address that is used at the lowest level and this is the address your NIC (Network Interface Controller) IC will compare to determine if a packet should be received and passed to the stack or not.

Received ARP requests are dealt with automatically by the stack – no application intervention is required.

The driver / stack components will typically carry out ARP automatically when communicating with a remote device. The following information is provided in case you want to use ARP for a specific reason in your application.

Call the arp_resolve_ip_address() function with the IP address that needs resolving (that you want the mac address to be returned for)

Periodically call the arp_is_resolve_complete() function to determine when the ARP response has been received .

The calling function must use a timeout in case of no response.

If trying to resolve an IP address that is not on the same subnet then the resolve function will automatically resolve the address for the network gateway (our_gateway_ip_address) which should then do its job and forward on communications to the remote device. Therefore for IP addresses outside of the same subnet bear in mind that just because ARP was successful the remote IP address is not necessarily actually available.

```
        //----- RESOLVING AN IP ADDRESS EXAMPLE -----
        IP_ADDR destination_ip_address;
        MAC_ADDR destination_mac_address;

        case SM_GET_DESTINATION_MAC_ADDRESS:
            //----- DO THE ARP REQUEST -----
            destination_ip_address.v[0] = 213;
            destination_ip_address.v[1] = 171;
            destination_ip_address.v[2] = 193;
            destination_ip_address.v[3] = 5;

            if (arp_resolve_ip_address(&destination_ip_address))
            {
                  arp_response_timeout_timer = 10;
                  our_state = SM_GET_DESTINATION_MAC_ADDRESS_WAIT
            }
            //Could not transmit right now - try again next time
            break;

      case SM_GET_DESTINATION_MAC_ADDRESS_WAIT:
            //----- WAIT FOR ARP RESPONSE -----
            if (arp_is_resolve_complete (&destination_ip_address,
&destination_mac_address))
            {
                  //ARP IS COMPLETE
            }

            if (arp_response_timeout_timer == 0)
            {
                  //ARP FAILED - REMOTE DEVICE NOT FOUND
            }
            break;
```

## USING DHCP

For more detailed information about how DHCP works see the ‘How The Driver Works’ section later in this manual.

DHCP (Dynamic Host Configuration Protocol) is an optional but widely used protocol that may be used by networked devices (clients) to automatically obtain an IP address, subnet mask and gateway address on power up and to renew this information periodically.

When the DHCP driver is used the following should be done by your application initialisation function prior to calling the tcp_ip_initialise() function. These variables may also be changed later at any time to switch between manual and DHCP IP settings and the driver will automatically handle the changeover.

```
      //----- TO USE A MANUALLY CONFIGURED IP SETTINGS -----
      eth_dhcp_using_manual_settings = 1;
      our_ip_address.v[0] = 192;                //MSB
      our_ip_address.v[1] = 168;
      our_ip_address.v[2] = 0;
      our_ip_address.v[3] = 51;                 //LSB
      our_subnet_mask.v[0] = 255;               //MSB
      our_subnet_mask.v[1] = 255;
      our_subnet_mask.v[2] = 255;
      our_subnet_mask.v[3] = 0;                 //LSB
      our_gateway_ip_address.v[0] = 192;
      our_gateway_ip_address.v[1] = 168;
      our_gateway_ip_address.v[2] = 0;
      our_gateway_ip_address.v[3] = 1;

or


      //----- TO USE DHCP CONFIGURED IP SETTINGS -----
      eth_dhcp_using_manual_settings = 0;
```

If your DHCP server stores name information and you want to include your device name as part of the DHCP process you can use the following pointer. Some DHCP servers will store received names of devices, others will not (and may use NetBIOS if need be instead). There is no requirement to pass a name using DHCP and it is typically used where there is a specific need on a network.

```
//SET NAME TO BE RETURED TO DHCP SERVER IN DHCP PACKETS
eth_dhcp_our_name_pointer = &netbios_our_network_name[0];
//Use the NetBIOS name (if NetBIOS is being used) or replace with any
//ram array containing your null terminated device name - fixed length
//of 15 bytes excluding terminating null.
```

DHCP is dealt with automatically by the stack if DHCP is included – no application intervention is required

When enabled the driver will automatically obtain an IP address from a DHCP server and renew it as required

The following variable may be useful in your user application

> our_ip_address_is_valid //= 1 if we have valid IP settings (either manually or from a DHCP server), 0 otherwise

## USING NETBIOS

For more detailed information about how NetBIOS works see the 'How The Driver Works' section later in this manual.

The driver responds to NetBIOS (Network Basic Input/Output System) requests from other local network devices where the name in the request matches the name given to the driver. Therefore to use the NetBIOS function all you need to do is ensure there is a UDP socket available for NetBIOS to use and load the name to be used into the following array:

```
netbios_our_network_name[0] = 'e';        //16 byte null terminated array
netbios_our_network_name[1] = 'm';        //(Not case sensitive)
netbios_our_network_name[2] = 'b';
netbios_our_network_name[3] = 'e';
netbios_our_network_name[4] = 'd';
netbios_our_network_name[5] = 'd';
netbios_our_network_name[6] = 'e';
netbios_our_network_name[7] = 'd';
netbios_our_network_name[8] = '-';
netbios_our_network_name[9] = 'd';
netbios_our_network_name[10] = 'e';
netbios_our_network_name[11] = 'v';
netbios_our_network_name[12] = 'i';
netbios_our_network_name[13] = 'c';
netbios_our_network_name[14] = 'e';
netbios_our_network_name[15] = 0x00;
```

The driver will automatically open a UDP socket and listen for NetBIOS requests. When a request is received with a query name that matches this the driver automatically responds with a NetBIOS response containing the IP address.

When using NetBIOS you may simply enter the name instead of an IP address into computer applications. For instance if using this drivers HTTP server you may enter the name on its own in your browsers address bar to view the devices web interface.

There is one potential complication when using a NetBIOS name in a web browser. Windows will generally perform a DNS query for an new name first, as most browser entered names will be domain names of devices on the internet. If DNS fails Windows will then send a NetBIOS request to see if the name is on the local network. However if your gateway is using a DNS service that returns a valid address even if DNS has failed (for instance OpenDNS which does this to show search and advertising results when DNS fails) the browser will not know that DNS actually failed, will display the fake web page and NetBIOS will never be attampted. The solution to this is to:

Configure your DNS service not to perform this action

Configure your gateway router to use a different DNS service

Setup Windows with a registry change to make NetBIOS happen first

Add DNS records to your local DNS server for your local NetBIOS devices.

## USING DNS

For more detailed information about how DNS works see the 'How The Driver Works' section later in this manual.

DNS (Domain Name System) is an internet protocol that allows human friendly names to be converted into computer friendly numbers (IP addresses). A DNS server simply receives a request for the IP address of a server from a provided text name (e.g. www.yahoo.com). Your devices internet connection will be provided via some form of 'router' and this router should incorporate a DNS server. The driver simply uses the address of the network gateway (either automatically learnt on power up using DHCP or manually provided by the user) and sends its DNS requests to be looked up by the gateway. The gateway responds with the IP address if found. The exact feature required by the gateway device is 'recursion' and this is provided by virtually all modern routers, from full blown servers to simple low cost DSL routers.

DNS requests are automatically dealt with by the stack, for instance when connecting to a POP3 or SMTP server.

No user application intervention is required. The following example is shown in case you want to use DNS for a specific reason in your application. Note that only 1 DNS resolution may be carried out at a time.

```
IP_ADDR dns_resolved_ip_address;

      //----- START NEW DNS QUERY -----
      if (!do_dns_query(temp_string, QNS_QUERY_TYPE_HOST))
      {
            //DNS query not currently available (already doing a query)
//try again next time
      }

      //----- IS DNS QUERY IS COMPLETE? -----
      //(Do this periodically until complete)
      dns_resolved_ip_address = check_dns_response();
      if (dns_resolved_ip_address.val == 0xffffffff)
      {
            //DNS QUERY FAILED
            //(Timed out or invalid response)
      }
      else if (dns_resolved_ip_address.val)
      {
            //DNS QUERY SUCESSFUL
            //Store the IP address
            my_remote_device_ip_address.val = dns_resolved_ip_address.val;
      }
      else
      {
            //DNS NOT YET COMPLETE
      }
```

## USING POP3

For more detailed information about how POP3 works see the 'How The Driver Works' section later in this manual.

POP3 (Post Office Protocol version 3) and IMAP4 are the two most prevalent Internet standard protocols for receiving email. POP3 tends to be more suited to general embedded devices and is therefore the protocol supported by this driver. A POP3 email account will be provided by any typical internet service provider (many don't support IMAP).

The POP3 driver allows you to specify a POP3 server URL, username and password, using either constants or variables. POP3 is carried out completely automatically by the driver and once triggered by a call to email_start_receive() the driver will use DNS to resolve the address of the POP3 server, connect to it, authenticate itself and then query how many emails are in the inbox. The driver will then read each email in turn and as it does it will first pass the email subject string to a specially defined function in your application and then each line of the email, for your application to process as desired. As the reading of each email is completed a final call is made to the function in your application with your return value indicating to the driver if it should delete or leave the email in the POP3 inbox.

MIME is used as the standard for attachments and non ASCII text in e-mail. Although POP3 does not require MIME formatted email, essentially most Internet email comes MIME formatted. This driver does not provide MIME decoding of email. Where you are sending ASCII text via email this is not a limitation as you simply send the email as plain text and the text will be located at the start of the email body section. Although it may be followed by MIME message fields and boundaries, you can simply ignore these. Sending an email as plain text ensures that the textual contents of the email will be sent as entered and simple parsing techniques can be used by your application when reading each line of the received email body. If you wish to send file attachments via email these will be MIME encoded by the sending software and your application will need to provide MIME decoding functionality. This driver does not provide this for you as typically the receiving of emails by embedded devices is simply required to pass configuration data or commands and these can be contained in the email body as text. As MIME has multiple encoding options and a receiver has no control over which encoding type will have been used by the sender of the email it is code heavy to implement this decoding. However if you require it for your application it is certainly possible to implement as each line of the MIME encoded email body is passed to your application. MIME decoding is undertaken for the HTTP driver POST method and these functions may be helpful to incorporate email MIME decoding.

Alternative methods of passing a file to your embedded device are to use the HTTP driver POST method via a web page or use POP3 email as a method of sending a trigger and web URL to the device and then using the much more straightforward method of the device using the driver to connect to the web site via TCP and download the file.

Each call by the driver to your application function includes the emails senders address so that you can optionally verify who sent the email or store it to respond with an email sent using SMTP.

To avoid potential problems POP3 receive can not be started if SMTP send is currently active

The driver includes an optional pointer pop3_email_progress_string_pointer which is updated at each stage of the sending process. If your device includes a screen then you may wish to display the state of the email receive process to the user as it occurs, to confirm the operation is happening or so the user can determine at which stage the operation fails if there is a problem and take appropriate action, such as correcting the entered mailbox settings.

See the eth-pop3.h file for full details of the functions you need to include.

### An example of how to receive emails using POP3

```
	//-------------------------------
	//----- START EMAIL RECEIVE -----
	//-------------------------------
	//If POP3_USING_CONST_ROM_SETTINGS is commented out load the byte arrays
you are using for the following string defines:
      POP3_SERVER_STRING
      POP3_USERNAME_STRING
      POP3_PASSWORD_STRING

      email_start_receive();        //Trigger receiving email

//------------------------------------------------------------------
//----- FUNCTION TO PROCESS EACH LINE OF EACH RECEIVED EMAIL -----
//------------------------------------------------------------------
BYTE process_pop3_received_email_line (BYTE status, BYTE *string, BYTE
*sender_email_address)
{
      BYTE count;
      BYTE *p_dest;

      if (status == 0)
      {
            //----- START OF A NEW EMAIL - THIS IS THE SUBJECT -----
            //Process as desired

            return(0);
      }
      else if (status == 1)
      {
            //----- NEXT LINE OF THIS EMAIL -----
            //Process as desired

            return(0);
      }
      else
      {
            //----- END OF EMAIL -----
            return(1);  //0x01 = delete email, 0x00 = don't delete this email
      }
}
```

### An example of how to use the optional progress information strings

```
      //Include the DO_POP3_PROGRESS_STRING define to enable these messages
      //Edit the strings in eth-pop3.h as desired
      if (email_is_receive_active())
      {
            //----- WE ARE RECEIVING EMAIL -----
            //CHECK FOR UPDATE USER DISPLAY
            if (pop3_email_progress_string_update)
            {
                  //RECEIVE EMAIL STATUS HAS CHANGED - UPDATE DISPLAY
                  pop3_email_progress_string_update = 0;
//Display the status null terminated string to the user
//on our screen
                  //... = &pop3_email_progress_string[0];
            }
      }
```

Alternatively the POP3 state machine variable sm_pop3 may be used to determine the current state of email receive.

## USING SMTP

For more detailed information about how SMTP works see the ‘How The Driver Works’ section later in this manual.

SMTP (Simple Mail Transfer Protocol) is the standard protocol used by mail clients to send emails. A SMTP server facility is typically provided with an internet connection (for instance by a DSL or dial up internet provider) and also by hosting providers who often provide remote a SMTP which may be used with authentication. The advantage of an internet providers SMTP server is that it will typically be free, however there may be requirements on the sender settings of emails and a device will need to be configured with the settings required by an individual connection. The advantage of a remote SMTP server is that all embedded devices can use the same server and login details regardless of where they are physically connected to the internet.

Whilst the receiving of POP3 emails is potentially made harder by the use of MIME formatting, sending emails with MIME is much easier. This is because it is the sender that decides how to MIME format an email and therefore the driver can use a pre determined approach that is simple and doesn't require large amounts of program memory. MIME is used as the standard for attachments and non ASCII text in e-mail. Although SMTP is not required to send MIME formatted email, most Internet email is MIME formatted.

The SMTP driver allows you to specify a SMTP server URL, a username and password (if authentication is used), the recipients email address, the senders email address, the subject and an optional filename, using either constants or variables. SMTP is carried out completely automatically by the driver and once triggered by a call to email_start_send() the driver will use DNS to resolve the address of the SMTP server, connect to it, authenticate itself if necessary and then send an email with an optional file attachment. As the email is sent a special function in your application is called to request each byte of the email text and then each byte of the file attachment. The email is automatically MIME formatted and the file attachment BASE64 encoded by the driver, allowing any file type to be sent.

As TCP packets may need to be resent should a packet be lost, a variable is passed to the user application which is normally zero. However if it is not zero the application needs to be move back the specified number of bytes, as the driver is having to resend the previous packet. This may typically be accomplished by in your application using a simple byte counter or pointer variable to determine the next byte to be sent.

To avoid potential problems SMTP send can not be started if POP3 receive is currently active

The driver includes an optional pointer smtp_email_progress_string_pointer which is updated at each stage of the sending process. If your device includes a screen then you may wish to display the state of the email send process to the user as it occurs, to confirm the operation is happening or so the user can determine at which stage the operation fails if there is a problem and take appropriate action, such as correcting the entered server settings.

## An example of how to send an email using SMTP

```
static WORD send_email_body_next_byte;
      static DWORD send_email_file_next_byte;

      //----- START EMAIL SEND -----
      //If SMTP_USING_CONST_ROM_SETTINGS is commented out load the byte arrays
//your using for the following string defines:
      SMTP_SERVER_STRING
      SMTP_USERNAME_STRING
      SMTP_PASSWORD_STRING
      SMTP_TO_STRING
      SMTP_SENDER_STRING
      SMTP_SUBJECT_STRING

      send_email_body_next_byte = 0;
      send_email_file_next_byte = 0;
      email_start_send(1, 1);       //Trigger send email. Use authenticated
//login, include a file attachment

//-----------------------------------------------------------------------------
//----- FUNCTION TO PROVIDE EACH BYTE OF EMAIL BODY AND FILE ATTACHMENT -----
//-----------------------------------------------------------------------------
BYTE provide_smtp_next_data_byte (BYTE sending_email_body, BYTE
start_of_new_tcp_packet, WORD resend_move_back_bytes, BYTE* next_byte)
{
      if (sending_email_body)
```

```
	{
		//---------------------------------------------
		//----- PROVIDE NEXT BYTE OF EMAIL BODY -----
		//---------------------------------------------
		if (start_of_new_tcp_packet)
		{
			//LAST PACKET CONFIMRED AS SENT - IF WE NEED TO STORE VALUES
//FOR THE NEXT PACKET OF DATA TO BE SENT DO IT NOW

		}
		if (resend_move_back_bytes)
		{
			//WE NEED TO RESEND THE LAST PACKET
			//(if should always be true but check in case of error)
			if (resend_move_back_bytes >= send_email_body_next_byte)
				send_email_body_next_byte -= resend_move_back_bytes;
		}

		//GET NEXT BYTE OF TEXT
		if (send_email_body_text_array[send_email_body_next_byte] != 0x00)
		{
			//NEXT BYTE TO SEND
			*next_byte =
send_email_body_text_array[send_email_body_next_byte++];
			return(1);
		}
		else
		{
			//ALL BYTES SENT
			return(0);
		}
	}
	else
	{
		//-------------------------------------------------------
		//----- PROVIDE NEXT BYTE OF EMAIL FILE ATTACHMENT -----
		//-------------------------------------------------------
		if (start_of_new_tcp_packet)
		{
			//LAST PACKET CONFIMRED AS SENT - IF WE NEED TO STORE VALUES
//FOR THE NEXT PACKET OF DATA TO BE SENT DO IT NOW

		}
		if (resend_move_back_bytes)
		{
			//WE NEED TO RESEND THE LAST PACKET
			//(if should always be true but check in case of error)
			if (resend_move_back_bytes >= send_email_file_next_byte)
				send_email_file_next_byte -= resend_move_back_bytes;
		}

		//GET NEXT BYTE OF FILE ATTACHMENT
		if (our_file_to_send_size > send_email_file_next_byte)
		{
			//NEXT BYTE TO SEND
			*next_byte =
our_file_to_send_array[send_email_file_next_byte++];
			return(1);
		}
		else
		{
			//ALL BYTES SENT
			return(0);
		}
```

```
        }
}
```

### An example of how to use the optional progress information strings

```
        //Include the DO_SMTP_PROGRESS_STRING define to enable these messages
        //Edit the strings in eth-smtp.h as desired
        if (email_is_send_active())
        {
                //----- WE ARE SENDING AN EMAIL -----
                //CHECK FOR UPDATE USER DISPLAY
                if (smtp_email_progress_string_update)
                {
                        //SEND EMAIL STATUS HAS CHANGED - UPDATE DISPLAY
                        smtp_email_progress_string_update = 0;
//Display the status string to the user on our screen
                        //... = &smtp_email_progress_string[0];
                }
        }
```

Alternatively the SMTP state machine variable sm_smtp may be used to determine the current state of email send.

## USING ICMP

For more detailed information about how ICMP works see the ‘How The Driver Works’ section later in this manual.

ICMP (Internet Control Message Protocol) is provided by the driver to respond to ping requests received. When a ping request is received (which actually is an ICMP ‘Echo request’) the driver will automatically respond with an ICMP ‘Echo reply’ message. ICMP ECHO requests are automatically dealt with by the stack and no user application intervention is required

## USING SNTP

For more detailed information about how SNTP works see the ‘How The Driver Works’ section later in this manual.

A SNTP (Simple Network Time Protocol) client is included to allow your embedded device to retrieve the current time from a public NTP server. This can be a very useful way of avoiding including a battery backed real time clock in your system. A common misconception regarding SNTP and NTP is that it allows a device to automatically retrieve the current time without any special programming or end user configuration required. What SNTP actually provides is a 32 bit value of the number of seconds since 00:00:00 on 1 January 1900. Whilst this is the current time the value requires processing to be useful for most applications to deal with time zones, daylight saving and leap years. A typical usage of SNTP is to provide the means for the user to set the current time and to then use SNTP to accurately track time from that point. For instance if your device has its time set by a user and immediately sends a SNTP request, it will receive the current SNTP seconds value back. The device now knows the entered date and time equates to the received SNTP time value and can store this to non volatile memory. Whilst powered the device should keep track of time using its own oscillator. Every second that passes the internal real time clock can be incremented and a live count of the SNTP value incremented. Periodically (every few hours for instance) the device can send an SNTP request and compare the value received back with its live seconds count value. If the values differ, implying that the devices internal real time clock has become incorrect, the real time clock can be adjusted accordingly. Once the internal real time clock value has been adjusted (if it needs to be) the device then saves the new real time clock value and SNTP seconds value over the previous values in the non volatile memory. This carry’s on, ensuring that the devices real time clock is regularly corrected, but not too often to avoid swamping SNTP servers or burning out the non volatile memory.

Should the device loose power, when it next powers up it can read its last saved real time clock value and SNTP seconds count from the non volatile memory and then send a SNTP request. The received value can then be compared to the saved SNTP seconds count and the difference used to update the real time clock value to the current time.

This provides an elegant and relatively simple method of providing a non volatile real time clock for embedded devices that will be connected to the internet. However remember that you will need to allow for daylight saving and leap years yourself, and this can typically be accomplished by creating some form of hard coded lookup table of the relevant dates these will occur on for a block of future years.

The NTP timestamp is represented as a 64-bit unsigned fixed-point number. The integer part (seconds) is in the first 32 bits, and the fraction part in the last 32 bits. This driver deals with the integer seconds value only, and ignores the fraction part as inherent internet communication delays make using SNTP with an accuracy better than a second difficult. However it would be easy to modify the driver code to read the fractional part of the value if desired.

The maximum 32 bit value that can be represented is 4,294,967,295 seconds. Since some time in 1968 (second 2,147,483,648), the most significant bit has been set and the entire 64-bit field will overflow on 7 February 2036 (second 4,294,967,296). A simple and convenient way to extend the useful life of NTP timestamps is to use the following convention: If bit 0 is set, the UTC time is in the range 1968-2036, and UTC time is reckoned from 0h 0m 0s UTC on 1 January 1900. If bit 0 is not set, the time is in the range 2036-2104 and UTC time is reckoned from 6h 28m 16s UTC on 7 February 2036.

Below is an example of how to get the current SNTP time from your application:

```
	//----- START SNTP GET TIME PROCESS -----
	sntp_get_time();

//----- SNTP GET TIME EVENTS FUNCTION -----
//We have set this function to be called in eth-sntp.h when we trigger the
//SNTP client.  Its called once when the SNTP request gets sent (after DNS
//and ARP lookup has occurred, in case we want to start a tight timeout timer)
//and once when the response is received.
void sntp_send_receive_handler (BYTE event, DWORD sntp_seconds_value)
{

	if (event == 1)
	{
		//----- JUST SENT SNTP REQUEST -----

	}
	else if (event == 2)
	{
		//----- JUST RECEIVED SNTP RESPONSE -----
		//sntp_seconds_value is SNTP server time in seconds

	}
	else if (event == 3)
	{
		//----- SNTP FAILED -----
		//There was no response at one of the steps to get a SNTP response

	}
}
```

## USING DRIVER WITH A NEW NETWORK INTERFACE CONTROLLER IC

Using the driver with a new nic (Network Interface Controller) requires adjusting the functions in the nic.c file to suit your nic (this file contains just the functions that are nic specific and doesn't contain any of the general TCP/IP stack driver functions). This is simply a case of adapting each of the functions that are called by the stack to suit accessing your particular nic and following this guide should make the process straightforward.

## Copy A Pair Of Sample Files

First copy the sample nic.c and nic.h pair of files for the included nic models that is most similar to your nic, to use as a reference. These can then be modified as necessary for your nic.

## Nic Specific Defines

Go through each of the defines in the nic.h file and update as required for your nic.

## IO Defines

Alter the IO defines to provide all of the input and output pins required by your nic and serial port access if your nic uses a serial rather than parallel interface.

## Bus Access Delay Defines

NIC_DELAY_READ_WRITE

> This define may be used with parallel interface nic's to add in one or more null (no operation) instructions to allow data bus signals to stabilise during read or write access. 2 layer PCB's (no ground plane) and PCB's with long track lengths are more likely to require this. If you are experiencing problems setting up a new nic or PCB it is worth using this define to give a reasonable delay and then remove it later once you have the nic working.

## Nic Specific Functions

The sample nic.c file you have copied may contain functions other than the below, but they will simply be included to support and be called by these main functions (i.e. used locally and not of interest to the rest of the TCP/IP driver).

If your nic has a 16 (or 32 bit) interface then use a sample file which also has an interface wider than 8 bits to see the simple technique to deal with the functions below that work with reading and writing individual bytes.

void nic_initialise (BYTE init_config)

> Adjust to provide resetting of the nic and initialisation of all registers that require initialisation. Only 1 transmit buffer is required and all of the nics remaining memory can be allocated to receiving packets. Transmitted packets can be discarded once sent or if send fails.
>
> This function also needs to set the nic's MAC address from the following array:
>
> our_mac_address
>
> If the nic supports both 10Mbps and 100Mbps then use the BYTE value it is called with to set the nic's speed:
>
> 0 = allow speed 10 / 100 Mbps
>
> 1 = force speed to 10 Mbps
>
> Typically the interrupt output pin needs to be configured to be active when a packet has been received or the link status has changed.
>
> Finally, enable packet reception and do the following:
>
> //—— DO FINAL FLAGS SETUP ——
>
> nic_is_linked = 0;
>
> nic_speed_is_100mbps = 0;
>
> nic_rx_packet_waiting_to_be_dumped = 0;

WORD nic_check_for_rx (void)

Read the relevant nic register and return 0 if no rx waiting or the number of bytes in the packet if rx is waiting

void nic_setup_read_data (void)

Set the nic's read pointer to the beginning of the next unprocessed received packet so that the next operation can read the nic and data will be transferred from the data buffer

BYTE nic_read_next_byte (BYTE *data)

Read the next byte from the nic's buffer memory (nic_setup_read_data will have already been called) and return 1 if the read was successful or 0 if nic_rx_bytes_remaining is zero and there are no more bytes in the rx buffer.

If a byte was read do the following:

nic_rx_bytes_remaining–;

BYTE nic_read_array (BYTE *array_buffer, WORD array_length)

Read the specified number of bytes from the nic's receive buffer (nic_setup_read_data will have already been called) and return 1 if the read was successful or 0 if nic_rx_bytes_remaining becomes zero and there are no more bytes in the rx buffer.

As each byte is read do the following:

nic_rx_bytes_remaining–;

void nic_move_pointer (WORD move_pointer_to_ethernet_byte)

Move the nic's read pointer to a specified byte location ready to be read next (a value of 0 = the first byte of the Ethernet header). Also do the following:

nic_rx_bytes_remaining = nic_rx_packet_total_ethernet_bytes – move_pointer_to_ethernet_byte;

void nic_rx_dump_packet (void)

Discard any remaining bytes in the current received packet and free up the nic for the next received packet. Also do the following:

//EXIT IF PACKET HAS ALREADY BEEN DISCARDED

if(nic_rx_packet_waiting_to_be_dumped == 0)

return;

//Discard the packet

//Flag that packet has been dumped

nic_rx_packet_waiting_to_be_dumped = 0;

BYTE nic_setup_tx (void)

Check the nic to see if it is ready to accept a new packet to be transmitted. If not return 0. If yes then it needs to set up the nic ready for the first byte of the data area (Ethernet frame) to be sent. The following checks may need to be made depending on your nic:

Check the nic isn't overflowed

Check the nic isn't tied up still sending the last transmitted packet

If also needs to:

nic_tx_len = 0;

void write_eth_header_to_nic (MAC_ADDR *remote_mac_address, WORD ethernet_packet_type)

This function can be a copy from the sample driver used.

void nic_write_next_byte (BYTE data)

Write the byte to the nic (nic_setup_tx will have already been called). It needs to do the following before outputting the byte:

if(nic_tx_len >= 1536)

return;

nic_tx_len++;

void nic_write_array (BYTE *array_buffer, WORD array_length)

Write the specified number of bytes to the nic (nic_setup_tx will have already been called). It needs to do the following before outputting each byte:

if(nic_tx_len >= 1536)

break;

nic_tx_len++;

void nic_write_tx_word_at_location (WORD byte_address, WORD data)

The byte_address supplied will be word aligned. Move the nic's transmit buffer pointer to the specified location, write the value and then restore the pointer back to its previous location. This function is used when writing information required at the beginning of a packet that is not necessarily known before the end of the packet is written (such as length).

void nix_tx_packet (void)

Transmit the packet that has been constructed in the nic's transmit buffer. The following needs to be included:

//If packet is below minimum length add pad bytes ––

while (nic_tx_len < 60)

nic_write_next_byte(0x00);

//Add the Ethernet CRC if the nic doesn't do this automatically

//(most do)

BYTE nic_ok_to_do_tx (void)

Return 0 if the nic is not currently connected, is waiting for a received packet to be finished with and dumped (nic_rx_packet_waiting_to_be_dumped == 1) or is busy with a previous transmission. Otherwise return 1.

## Getting A New Nic To Work

Often the nic manufacturer will provide sample code showing how to initialise, read and write the nic which can be used as the basis for the functions this driver requires. If problems are experienced its often useful to create a simple function to read the entire contents of the nic's control registers so that you can debug why the nic isn't doing what you expect.

One of the simplest ways to check a new nic is working correctly is to enable the drivers DHCP function, ensure there is a DHCP server on the network and then monitor the network traffic using Wireshark or a similar program to see if the DHCP packets are sent and received correctly. It that is successful then ping the device the driver is running on to confirm that array reading and writing is also working. If not breakpoint at important points in the above functions to find out where the problem is that needs to be corrected.

# GENERATING HTTP WEB CONTENT

## CREATING YOUR WEB PAGES

Standard HTML web sites are really very simple. You design one or more HTML pages, which are simply ASCII text files with the extension .htm or .html (you can open them in Windows Notepad). The HTML file contains specific sections which contain instructions to the browser receiving the file. In the <BODY> section of the file is the text that is displayed on the web page, surrounded by special tags that tell the browser how to format it, or styles that tell the browser to style it using styles specified elsewhere. Images are included in a web page using special tags which specify the URL of the image file. As a browser encounters these it then sends separate requests to the HTML server to download each image file and it then adds them to the displayed page. Other files that are required by the web page (for instance style sheets) are also separately requested by the browser as it encounters references to them.

If a web page has special functionality it will either be accomplished using 'client side' or 'server side' functions. As 'client side' functionality is provided by the users computer (for instance general JAVA, Flash) the code to use this functionality can be included in your HTML pages if desired. However 'server side'. functionality is provided by a server (for instance PHP) and you therefore can't include this in you HTML pages as this driver provides an HTML server, not an HTML server with PHP, CGI, etc, plug ins.

The HTML specification is available from the World Wide Web Consortium (W3C) at www.w3.org. The specifics of HTML creation are outside the scope of this manual and there are many resources available to teach you this if required. Although HTML can be created using a text editor it can be a painful task and there are many GUI applications available to design web pages. Applications such as Adobe Dreamweaver provide very powerful HTML tools, but with a bigger learning curve required to effectively use them. Many simpler applications are also available although if you are a new user it is worth having a hunt around for general comments about an application you are considering using as some create quite sloppy or bloated (large) HTML code. This driver doesn't care which program you use to create your web pages. The only requirements are that you can not use sub directories more than 1 level deep (only files from the main directory and any sub directories directly off that directory can be used) and the only 'server side' processing you may use is the dynamic data function provided by the driver.

## INCLUDING DYNAMIC CONTENT IN YOUR WEB PAGES

To allow you to be able to display text or items in a web page that are selected as the page is served to a browser the driver includes a powerful dynamic content feature. As the driver reads each .htm HTML file from memory and outputs it into a TCP packet, it checks every byte to see if it is the tilde '~' character. When a tilde character is found it is not outputted as part of the HTTP page but instead the driver continues reading and storing subsequent characters as a variable name until it finds the hyphen '-' character. As soon as the hyphen character is found the driver calls the function defined by HTTP_DYNAMIC_DATA_FUNCTION in the eth-http.h header file.

The function is called by the driver with the variable found, together with the TCP socket ID of the client computer the page is being sent to (in case its of use for example to identify a user by their unique MAC or IP address). Your function can then compare the variable name against a list of possible variable names you included in your web pages and return a string containing whatever you wish (maximum 100 characters) which the driver will output before continuing to send the rest of the page.

There are no restrictions or requirements on this. Anywhere you place the following:

~my_variable_name-

in your .htm files will cause your function to be called each time with the driver replacing the .htm file content from the tilde character to the hyphen character (inclusive) with the string you return from your function (excluding the 0x00 null termination). This means you can include anything from dynamically generated text to dynamically selected images or links.

The tilde character was selected as it's a character that is very rarely used in web pages. The use of html comments was considered (<!– –>) but these can cause strange preview results in many web page creation programs when used inside certain special tags. Should you need to include the tilde character in your web page to be displayed as a tilde character then you can simply use the html character code for it instead in the html source:

&#126 in html source will display the tilde character ~

## SENDING DATA TO YOUR EMBEDDED HTTP SERVER

Being able to provide web pages that include dynamically generated content is very useful but it doesn't allow a user to send data to your embedded device using the web interface. You may want to provide the means for users to turn on and off simple options, adjust numeric or textual parameters or even upload files. This driver provides the common HTTP server form input methods, optimised for embedded use. Typically these can be implemented on your web page by inserting a standard form and the individual form components using your web design application.

### GET Requests

The most simple of all input methods is to use the standard GET request. This is exactly the same request that is used by a browser to retrieve web pages and resources such as images, but with input data being sent back by the browser tagged onto the end of the requested filename. You've probably noticed this input method before when using all sorts of internet sites. All that has to be done is to follow the filename the browser is GET'ing with a '?' character and then one or more input names and values. For example the following is a GET request with inputs:

/index.htm?variable1=25&varaible2=1280&string1=Hello+World

The filename being requested is immediately followed by the '?' character, which signifies that inputs follow. Each input is then added to the end formatted as follows:

input_name=input_value

with a '&' character between each input. As spaces are not permitted, if an input value contains spaces they are converted to the '+' character by the browser before being added, and converted back to spaces by the HTTP server as they are received. If a special character needs to be sent (for instance a '?', '=', '&', '+', '%') it may be sent by using the '%' character followed by the 2 character hexadecimal code which represents the character. As this special 3 character sequence is received it is decoded into the required character by the driver.

The first space character found after the start of the filename marks the end of the inputs.

When your browser sends this type of GET request, which is typically generated when a form is submitted but can also simply be the URL of a hyperlink, the data being submitted is displayed in the browsers address bar exactly as it is sent to the HTTP server. The typical way to cause a browser to include inputs with a GET request is to include a form (using GET) on your html page with one or more inputs. When the form submit button is pressed the GET request is sent together with all of the form controls as input values.

This driver provides a very simple and memory efficient method of passing received input values to your main application. You define a function in the eth-http.h file to be called when an input value is received and the HTTP server will automatically call it every time it encounters an input value. The call includes the input name string that was sent by the browser (which will typically be the name you gave the form object when designing the html form), the input value string (fully decoded to match what the user entered or selected), plus the filename being requested and the TCP socket ID (in case these are required by your application to uniquely identify input values against particular files or clients. How you process the input data is entirely up to you, with the HTTP server simply acting as a relay of the values to your applications function. As a pointer to the requested filename is passed to your function you can also use this to change the filename that will be returned to the user once the inputs have been processed, by modifying it. This can be useful if you are, say, checking for a valid password entry to re-direct the user from a default 'bad password' page to a 'you have logged in' page.

Any input data is provided to your applications function before the filename the browser requested is returned by the HTTP server, allowing a new web page to include dynamic content retrieved from your application based on the data just submitted (useful when you want the next web page to confirm the values just entered).

The GET input method is an excellent method of passing small amounts of input data whilst at the same time requesting a new html page. Its only significant drawbacks are that the input data is visible in the browsers address bar (and may be cached by some applications or hardware which could be a concern for sensitive information) and that it only supports ASCII text.

The included sample project web page 'Setup POP3' section is a working example of a GET form.

## POST Requests – application/x-www-form-urlencoded

The basic POST request with encoding type "application/x-www-form-urlencoded" is very similar to the GET request method of providing input values, except that the input values are contained within the message body of the TCP transmission and not after the filename. This means they can only be viewed by a network analyser and they can also be longer if required. When posting an html form the input values are formatted and passed in exactly the same way as the GET method, so the message data area of a POST request might contain the following:

```
variable1=25&varaible2=1280&string1=Hello+World
```

The driver passes POST request input values to your application by calling the same function as is used for GET input values (defined in eth-http.h). This will be called in exactly the same way as a GET request, with a separate call for each input value and with the call including the input name string, the input value string, the filename being requested and the TCP socket ID. How you process the input data is entirely up to you, with the HTTP server simply acting as a relay of the values to your applications function. As a pointer to the requested filename is passed to your function you can also use this to change the filename that will be returned to the user once the inputs have been processed by modifying it. This can be useful if you are, say, checking for a valid password entry to re-direct the user from a default 'bad password' page to a 'you have logged in' page.

The POST input method is an excellent way to pass input values that you don't want a user to see or larger input values.

The included sample project web page 'Setup SMTP' section is a working example of a POST "application/x-www-form-urlencoded" form.

**Driver Limitations**

To avoid the need for large ram buffers the driver supports the POST application/x-www-form-urlencoded method using a single packet for the input data. The input data may be located in the first packet message body after the HTTP headers or in a second subsequent packet immediately after the HTTP headers (browsers may do either). This restriction avoids the need for the driver to deal with inputs that span packet boundaries, which can happen in TCP transfers with a large block of data. In normal use this will not be an issue because as long as the total quantity of input names and data bytes does not exceed approximately 1000 bytes then it will fit within a single TCP packet even if included by the browser with the HTTP headers. However if this is an issue then use the POST multipart/form-data method below.

## POST Requests – multipart/form-data

The content type “application/x-www-form-urlencoded” is inefficient for sending large quantities of binary data or text containing non-ASCII characters. It is not possible to label the enclosed data with content type, apply a charset, or use other encoding mechanisms. The content type “multipart/form-data” should be used for submitting forms that contain non ASCII data and is also the standard method used to allow files to be uploaded from a browser.

The driver provides simple but effective handling of this type of POST request, and decodes the multipart data before passing it to the user application, so that the application gets the data exactly as submitted by a user. Instead of providing the data as complete strings, the driver passes it byte by byte, allowing binary data and data spanning many TCP packets to be handled.

The included sample project web page ‘Upload File’ section is a working example of a POST “multipart/form-data” form.

Limitations

> The driver supports uploading single files per input using a POST request. Multiple selected files in 1 form input (“multipart/mixed”) are not supported (and are not generally required).
>
> To avoid the need for large ram buffers the driver supports only a single HTTP client actively using the POST method at a time. As typical POST requests will typically only take one or a very few TCP packets to complete the upload of the input data this is typically not an issue as two HTTP clients would have to try and POST at exactly the same time for one of them to be rejected by the driver. However when uploading large files this will block any other HTTP clients being able to use the POST method until the upload is complete. Should a client be blocked from using POST it will receive a ‘503 Service Unavailable’ response, which is a general server busy message advising the browser to try again after a delay.

## PUT Requests

PUT requests provide the most simple method of uploading files to a HTTP server. Although part of the HTTP specification, PUT is not generally supported by the main browsers. To utilise a PUT request you have to use script in your web page. However due to constant security concerns and vulnerabilities script that is cable of accessing client side files is often blocked by browsers and operating systems. Therefore, whilst more complex to handle, POST requests are implemented by this driver to handle file upload instead of PUT requests. Due to the widespread adoption of POST as the method to use for file uploads you can be confident that it will just work with the various mainstream browsers and operating systems available.

## HTML Forms For Input Data

The typical method of sending input data with a GET or POST request is by using one or more forms on your web page. A form always has three elements: <form></form> tags that define the start and end of the form, one or more controls that allow the user to provide data to the server and a submit button. An example of the HTML code for a form:

```
<form action=″index.htm″ method=″get″>

<input type=″text″ name=″max_level″ maxlength=″3″ value=″50″>

</form>
```

action

> Names the URL where the browser will submit the form data to when the submit button is clicked. This is simply the filename that will be requested when the form data is sent. Once all of the form input data has been sent to your function this is the file that will be returned by the HTTP driver. The driver will pass the filename to your function that processes each of the input values in case it is helpful. If you wish you may change the filename at any time while processing the input data in your function allowing, for instance, a log in success page to be returned if a user enters the correct log in password.

method

Specifies whether the browser will use a HTML GET or POST request to send form data to the server (see above for details of each type)

enctype

Specified for POST forms to select either the “application/x-www-form-urlencoded” or “multipart/form-data” method.

input

Start of a user input control

type

The type of user input control. Text displays a single line text box. Also available are check boxes, radio buttons, passwords, multiline text boxes, hidden fields, etc.

name

Unique within the form to identify the control.

maxlength

Maximum number of characters the user may enter

value

The default value to display (for a user adjustable embedded parameter this would typically be dynamically retrieved from the embedded application as the web page is transmitted so the user is shown the current value).

**Tips for creating forms**

Where your site will have several forms it’s often a good idea to give form elements unique names so that when processing an input value in your applications function you can check just the value name and without having to also check the filename to know what the input value is for.

Keeping input value names short will reduce your program memory space and speed up the time it takes to compare received value names with all of the possible names.

Where you have several input values that are related consider using a common name followed by an index number when naming them. For instance when providing an option to set a MAC address you might have 6 input boxes which you could call:

mac1, mac2, mac3, mac4, mac5 & mac6

Your function which handles the input values could then simply check to see I the input name contains ‘mac’ and if it does then check which indexed value of the mac address (1 – 6) it is.

## USEFUL HTTP DESIGN NOTES

### Refreshing a page automatically

Once loaded an HTML page will not be re-loaded unless you click the browsers refresh button. You can cause a page to automatically re-load periodically using the following HTML code in a web pages <head></head> section:

```
<meta http-equiv=”Refresh” Content=”30”>
```

This will cause the page to be re-loaded every 30 seconds in most browsers.

## Limited Processing Power

If your embedded microcontroller or processor is not particularly fast then you can speed up web server speed by limiting the number of individual files to be served, limiting the length of filenames and any sub directories and keeping file sizes small where possible.

## Included resources

Remember that a single image file can be included in many different pages. Where memory space is tight (and to improve page loading speed) try to use generic image files that may be included in different places and which the browser will only need to request once.

Resources that are included in a web page, such as images, don’t have to be stored on the same device as the HTML web page. If you are designing with an embedded device that has limited memory there is nothing to stop you including resources such as images that are located on a different web server. When a client requests the web page it will request any included resources from whatever address is specified in the HTML code. Remember of course that the client must be able to connect to the other web server to be able to get the resources.

## File Extensions

The driver and web pages converter application processes all file extensions as 3 characters, but will correctly convert file extensions of .html to .htm

## Caching

A client browser will often cache retrieved resources to avoid re-loading them unnecessarily. If this causes your application problems there are approaches to avoid this occurring which you can search online for. Also bear in mind that if your browser is not displaying an updated image file, for instance, it may be because its not actually requesting it. In this instance waiting a period of time or changing the filename will force it to get a newly updated file. Also bear in mind that if testing across the internet, internet service providers (ISP’s) also cache to reduce bandwidth and demands on their networks, and again you may know that you’ve changed the content of a file such as an image but be baffled as to why the browser isn’t showing it, even though in this case you may see in Wireshark the file actually being downloaded! Waiting or changing the file name will again solve this.

# WEB PAGES CONVERTER PC APPLICATION

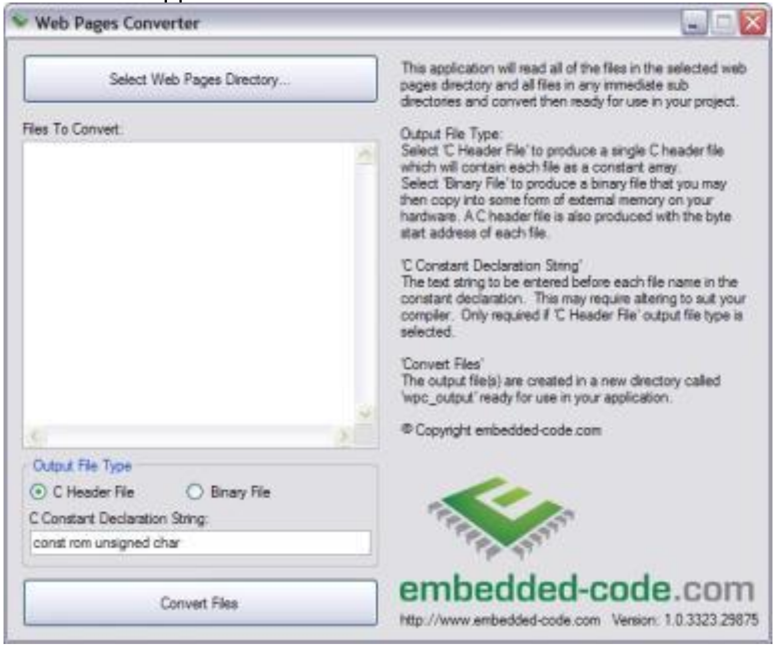
## USING THE APPLICATION

The included Web Pages Converter Application will read all of your web site files and convert them into the selected format ready for use in your project.

Your file and directory names must use C compliant names, as they are used in the C header file to point to that file.

Ensure that if you sites organisation uses sub directories (for instance if you've located your image files in a sub directory called \images) you are only using 1 level of subdirectories, as this application will not include files nested any deeper than this.

1. Run the Web Pages Converter Application:



This application will read all of the files in the selected web pages directory and all files in any immediate sub directories and convert then ready for use in your project.

Output File Type:
Select 'C Header File' to produce a single C header file which will contain each file as a constant array.
Select 'Binary File' to produce a binary file that you may then copy into some form of external memory on your hardware. A C header file is also produced with the byte start address of each file.

'C Constant Declaration String'
The text string to be entered before each file name in the constant declaration. This may require altering to suit your compiler. Only required if 'C Header File' output file type is selected.

'Convert Files'
The output file(s) are created in a new directory called 'wpc_output' ready for use in your application.

© Copyright embedded-code.com

Output File Type: [✓ C Header File] Binary File

C Constant Declaration String: const rom unsigned char

http://www.embedded-code.com Version: 1.0.3323.29875

2. Select the directory that contains the root file(s) that make up the web content / web site.

3. Select the output file Type:

> Select 'C Header File' to produce a single C header file which will contain each file as a constant array.
>
> Select 'Binary File' to produce a binary file that you may then copy into some form of external memory on your hardware. A C header file is also produced with the byte start address of each file.

4. If you have selected a 'C Header File' then check the 'C Constant Declaration String' is correct for your compiler. This is the text string that will be inserted before each file name in the constant declaration.

5. Press 'Convert Files' and the output file(s) are created in a new sub directory called '\wpc_output'. Copy the .h file to your projects main directory and its ready for use in your application. If you created a C Header File you just need to compile and the content will be available. If you selected Binary File you just need to compile and copy the binary file into your memory device before the content will be available.

# INFORMATION

## FREQUENTLY ASKED QUESTIONS

### This driver seems harder to use than some other drivers?
Different TCP/IP stacks take different approaches and our driver has been deliberately designed to give programmers low level access to all functionality.  By their nature TCP/IP stacks are complex and when implementing them on 8, 16 and 32 bit microcontrollers we prefer to be able to program “to the metal” to obtain optimum performance and versatility.  This means that to carry out an operation like sending a UDP packet for instance, you may need to do slightly more work than if using some other drivers, but with the advantage that you know exactly what is happening, rather than it being hidden “under the hood”, and your not wasting memory or resources using a higher level function designed to cover all sorts of eventualities not relevant to your application.

### Can a low end microcontroller really provide full TCP/IP functionality?
Yes. The real issue around TCP/IP functionality for embedded applications is typically not the capability of the microcontroller to provide the required functionality in a timely manor, but is the code space required for it. As microcontroller memory sizes get bigger and bigger this is no longer an issue even for really low cost devices. Here at embedded-code our engineers have designed many devices that use a simple 8 bit microcontroller and communicate using TCP/IP without any problems.

### Is TCP/IP communications hard?
Absolutely not. Getting a communication link between two devices or PC’s using UDP is incredibly simple and very much like getting a communication link between two devices using RS232 serial ports. There’s a couple more things to setup and understand but other than that its simply a case of outputting some data byte by byte and at the far end receiving it byte by byte. However unlike RS232 your communicating over Ethernet, which means your two devices or PC’s can be in the same room or across the world from each other with no additional work required by you (well OK, you’ll have to setup your router to allow the incoming connection, but other than that there’s nothing more you have to do!).

### TCP / UDP – Which Is Best?
You’ll probably come across programmers who say this isn’t even a question – use TCP because all the hard work is done for you. In many applications this is true and using TCP can make your life as a programmer very easy. However TCP has one very big drawback in embedded applications – when a packet gets lost or a remote device doesn’t respond you have to wait for the TCP stack to retry the communication several times, each after successively longer delays before it finally gives up. This is TCP’s big advantage and drawback at the same time. With UDP there is no re-try – if a packet gets lost you need to notice that you’ve not had a reply and send it again. To the hardened PC programmer this is task often not worthy of their genius, but to an embedded programmer this is something that can make UDP incredibly powerful and fast in time critical applications. Was it ever really a problem having to notice that a remote device hadn’t responded when talking to it using RS232? Well its no different when talking using UDP and you can use the same simple approaches to defining your communications link. UDP also has one other great advantage – it allows you to broadcast packets so that every device on the local network or in the devices sub net receives them – you can’t do that with TCP.

So the answer is, it depends on what you are doing. In terms of how the packets of data travel over the Ethernet links and Internet there is no difference – both get routed in exactly the same way. TCP is fantastic at offloading work from you as a programmer for many applications, but UDP is the obvious choice in a great many other applications where you need that raw control of what you are sending and receiving.

### Can I Open And Use Multiple Sockets At The Same Time?
Yes. You specify the maximum number of UDP and TCP sockets that the driver will provide

### Is the driver suitable for 64 bit architectures?
The driver uses structures to efficiently match the layout of several TCP/IP packet headers. The contents of all the structures that can be read or written using pointers by the driver are correctly aligned for 8, 16 and 32 architectures, except the ARP packet structure (which the driver reads and writes per variable and avoids

using pointers or sizeof() for 32 bit compilers that typically insert padding). The structures used are not however safe for 64bit architectures unless you compiler provides the option to force structures to not be padded. This is not an ANSI requirement, however many compilers will provide this option.

### What's Difference The Between Ethernet Switches & Hubs

First there we're Ethernet hubs, then came expensive switches and now you are hard pushed to find a hub! Ethernet hubs are simple dumb devices. A transmission from a device on any of its ports is 'seen' by every other device. This is fine as Ethernet is designed based on this, with your nic dealing with detecting packet collisions etc. However it can start to be inefficient if you have a lot of devices all connected by hubs. Ethernet switches on the other hand are intelligent. A switch learns (or is configured to know in the case of managed switches) what devices are connected to each of its ports and when a device transmits if the switch knows on which of its ports the device that the packet is intended for is, it only sends the packet to that port. This means that all the other ports are not blocked because one device is transmitting and they can transmit themselves at the same time with the switch handling buffering of packets if need be. In terms of improving a networks bandwidth switches are great. However they have two drawbacks.

> Firstly when debugging you may want to monitor the traffic being transmitted between other devices. You can't do this using a switch, unless the packets are broadcast UDP packets. However with a hub every device see's everything being sent so any device can monitor the traffic. Incidentally this is why many hotels are a hackers paradise as often cheaper or older hubs are installed so connecting a sniffer in any room allows you to see everyone else's traffic. The same is true of public WiFi hotspots and its why if you are doing anything at all sensitive on public Internet connections (even just checking your email) you should use https or a VPN service to encrypt your traffic.
>
> Secondly switches often have to buffer packets. This means that an element of uncertainly can be introduced in very time critical applications as a busy switch may buffer a packet for an unknown amount of time before it is finally passed to the remote device.

## SPECIFICATIONS

### Using The Driver With a RTOS or Kernel

The stack / driver is implemented as a single thread so you just need to make sure it is always called from a single thread (it is not designed to be thread safe).

## CODE AND DATA MEMORY REQUIREMENTS

### Ram Space

The amount of static ram memory used by the driver will vary greatly depending on the individual stack components that are selected. Below is an approximate guide for each of the stack components:-

General Stack and nic 128 bytes

UDP

> 5 bytes
> Plus 20 bytes per socket

TCP

> 19 bytes
> Plus 38 bytes per socket

HTTP

> 11 bytes
> Plus 14 bytes per socket if web files converted to a C header file, or 17 bytes per socket otherwise.
> Plus 149 bytes if POST inputs are enabled (the GET input method is still available if not)

DHCP

29 bytes

NetBIOS

55 bytes

DNS

83 bytes

POP3

142 bytes

SMTP

52 bytes

ICMP

0 bytes

SNTP

14 bytes

The driver requires a small to moderate amount of temporary variable storage space from the stack for its functions.

## Program Memory Space

The amount of program memory used by the driver will vary greatly depending on the individual stack components that are selected. The following program memory sizes are based on compiling with optimisations turned off, and therefore would obviously be reduced, significantly in many cases, with compiler optimising enabled. We present them in this un-optimised state so that you can gauge for yourself the likely program memory size for your specific application, without worrying that our figures have been made under perfect optimising conditions as a sales technique!
Using the Microchip C18 Compiler for PIC18 8 bit microcontrollers – Optimisations off

General Stack and nic

11280 program memory bytes (5640 x 16 bit instructions)

UDP

4548 program memory bytes (2274 x 16 bit instructions)

TCP

13970 program memory bytes (6985 x 16 bit instructions)

HTTP

8366 program memory bytes (4183 x 16 bit instructions)
3530 program memory bytes (1765 x 16 bit instructions) additional if POST inputs are enabled

DHCP

3948 program memory bytes (1974 x 16 bit instructions)

NetBIOS

1252 program memory bytes (626 x 16 bit instructions)

DNS

3216 program memory bytes (1608 x 16 bit instructions)

POP3

4280 program memory bytes (2140 x 16 bit instructions)

SMTP

8801 program memory bytes (4402 x 16 bit instructions)

ICMP

916 program memory bytes (458 x 16 bit instructions)

SNTP

2278 program memory bytes (1139 x 16 bit instructions)

Using the Microchip C30 Compiler for PIC24 & dsPIC 16 bit microcontrollers – Optimisations off

General Stack and nic

5490 program memory words (2745 x 24 bit instructions)

UDP

1608 program memory words (804 x 24 bit instructions)

TCP

6738 program memory words (3369 x 24 bit instructions)

HTTP

4526 rogram memory words (2263 x 24 bit instructions)
2396 program memory words (1198 x 24 bit instructions) additional if POST inputs are enabled

DHCP

2036 program memory words (1018 x 24 bit instructions)

NetBIOS

634 program memory words (318 x 24 bit instructions)

DNS

1450 program memory words (725 x 24 bit instructions)

POP3

2214 program memory words (1107 x 24 bit instructions)

SMTP

4512 program memory words (2256 x 24 bit instructions)

ICMP

266 program memory words (133 x 24 bit instructions)

SNTP

914 program memory words (457 x 24 bit instructions)

# How The Driver Works

## How The Basic Driver Works

***Note – this section of the manual is for information only. You do not need to read and understand this large and in depth section to use the driver! However you may want to if you wish to gain an understanding of how each of the driver components works.***

The eth-main.c and eth-main.h files are the central files of the TCP/IP driver. The central stack operations and definitions are provided in these files. If you want to get a feel for how the stack works at the basic level have a quick scan through the tcp_ip_process_stack() function.

The tcp_ip_initialise() function initialises the driver and automatically calls any other driver initialisation functions required by the selected driver / stack components. This function also calls the nic_initialise() function to initialise the nic (network interface controller) IC, provide it with its MAC address and enable the receiving of packets.

The user application then calls the tcp_ip_process_stack() function regularly (typically as part of an applications main loop) and this function checks for packets received by the nic, for responses that need to be sent and calls any background task functions that need to carry out periodic tasks.

The stack is designed as a single thread and no stack component is permitted to halt operation in wait states. Every operation that may require a delay or wait is handled by simple state machines so that completion of tasks or delays is checked for and handled each time the tcp_ip_process_stack() function is called. Therefore the stack cannot cause the user application to become halted.

When a packet is received the tcp_ip_process_stack() function starts the reading of the packet data from the nic, reading the Ethernet header which contains the senders and destination MAC address and the packet type code. It then branches depending on the packet type, and if the packet is IP then branching again for the IP packet type. At this stage this function passes the packet for processing to the individual driver / stack components that can handle it, or discards it if there is no facility to process it. Once the packet has been passed it is then down to the relevant driver component to continue processing the packet and finally discard it.

In addition to checking for received packets the tcp_ip_process_stack() function then calls each of the driver / stack components being used, allowing them to process packets passed to them and carry out any other operations, such as background tasks and transmitting packets.

## How The IP Driver Works

IP is the primary protocol in the Internet Layer of the Internet Protocol Suite. Its header contains the source and destination IP addresses of the packet, the protocol of the data within the IP packet (e.g. UDP, TCP, ICMP), the time to live for the packet (a value that is read and modified by individual routers to allow them to dump the packet if it takes too many hops to reach its destination) ,the packets length and a few other less important fields. This header contains all the information required by routers to deliver the packet to its destination irrelevant of what is contained within the data area of the IP packet. Although the header includes a checksum, this is only a checksum of the IP header itself and not the whole packet, allowing routers and devices to verify the integrity of the header information without wasting time checksumming the entire packet. Typically the protocol being transported by the IP packet (e.g. UDP) will use its own separate header which will be verified by the device the packet is being sent to, but is not important to all the devices the packet may pass through on its way.

This driver uses Internet Protocol Version 4 (IPv4) which is the dominant protocol of the Internet. Although its successor Internet Protocol Version 6 (IPv6) is now being actively deployed the abundance of IPv4 devices means that it isn't going away anytime soon, if ever.

As a TCP/IP packet is received by the driver it looks at the start of the packet to determine the packet type. If the packet is IP it then looks within the IP header to determine which protocol the IP packet contains and passes it on to the specific protocols driver to handle. The IP part of the overall TCP/IP driver is relatively simple. It provides IP headers for outgoing packets and reads the IP headers of incoming packets. Each packet is then created or processed by the other individual stack components

For most users there will be no need to directly use IP (Internet Protocol), as each of the TCP/IP stack components sits on top of the IP layer and therefore sends and receives the IP packets automatically (UDP, TCP and ICMP packets are actually contained within an enclosing IP packet). However the following usage example is given for users who want to send and receive their own IP packets, for instance if implementing a special protocol.

```
	//----- TRANSMIT AN IP PACKET -----
DEVICE_INFO *remote_device_info

	//setup transmit
	if (!nic_ok_to_do_tx()) //Exit if nic is not currently able to send a
return(0);		//new packet

	if (!nic_setup_tx())	//Setup the nic ready to tx a new packet
		return(0);	//Nic is not ready currently to tranmit a new
packet

	//WRITE THE IP HEADER
	ip_write_header(remote_device_info, ip_protocol);

	//WRITE THE DATA
	//Write each byte of the packet using:
	nic_write_next_byte(0x00);
	//or
	nic_write_array (my_array, sizeof(my_array))

	//TRANSMIT THE PACKET
	ip_tx_packet();
```

When transmitting a packet the total packet length does not need to be known as the length is automatically calculated by each of these functions and written when the ip_tx_packet() function is called.

```
	//----- TO RECEIVE AN IP PACKET -----

	//AsReception of IP packets is automatically dealt with by the stack, with
the different packet types then handled as required, to receive bespoke packet
you need to add your own handler to the following function in eth-main.c:
	tcp_ip_process_stack()

	//in the following section of its state machine:
	case SM_ETH_STACK_IP:

	//The contents of the received packet may be read using the following
functions:
	nic_read_next_byte()
	nic_read_array()

	//And finally the following function needs to be called to discard the
packet and allow the TCP/IP driver to continue its other processes:
	nic_rx_dump_packet()
	//The packet may be dumped without reading all of its contents if desired.
```

## How The UDP Driver Works

UDP, which is defined in RFC768, does about as little as a transport protocol can. Apart from adding the concept of local and remote 'ports', from and to which packets are sent, and optional checksumming of the UDP data it doesn't add anything to IP. When using UDP your application is able to transmit and receive packets in a completely raw fashion, in much the same way as sending and receiving packets of data over a serial link such as RS232. If error checking is enabled then packets will be automatically checksummed to detect errors, but no connection checking or automatic re-sending is undertaken. You send a packet and hopefully the device or devices you want to receive it will actually receive it. Your application needs to provide acknowledgements etc if you need to know you are connected or data has been received.

UDP's great advantages over TCP are that you don't ever have to wait on the stack, you can transmit to multiple devices at a time if you wish and its much lower processing overhead. In time critical applications TCP's automatic retry can effectively temporarily lock up your socket if a packet gets lost. With UDP you decide how to handle lost packets if you need to. TCP's great advantage over UDP is that connections are automatically made and lost packets are automatically re-transmitted.

UDP, like all of the stack process can only do one thing at once. When a UDP packet is received the application process that is using the socket must process and dump the packet, responding with a UDP tx packet if it wishes. Only 1 UDP packet may be received at any one time (other packets may be queued behind it in the nic receive buffer) and when transmitting a UDP packet it must be sent as a single process. This isn't as limiting as it sounds, it simply means that any application process that has opened a socket must deal with packets received to that socket as soon as they arrive and when transmitting a packet must transmit a packet in one go. Multiple sockets may all have different processes running concurrently and the nic's (network interface controller) receive buffers will automatically queue received packets for each socket.

UDP sockets are used with each assigned to a unique local port number. This allows multiple UDP sockets to be opened (total number available is defined in eth-udp.h) by the application and each socket used either for a brief exchange and then closed or just left open waiting for any UDP received to that socket (set with a broadcast IP address if required to receive packets from anyone).

## How The TCP Driver Works

TCP (specified in RFC 793) is the most complex of all of the components of this driver. It is a connection based protocol, whereby you create one or more client or server connections to a single other device through which you send and receive data. The protocol automatically handles acknowledging of data transferred and resending of data should packets be lost. From an embedded point of view this is quite a challenge as apart from requiring relatively large amounts of code space you can potentially require large ram buffers also.

This driver takes a simplistic and lowest 'cost' view of providing TCP functionality, to allow TCP to be provided using limited resource processors / microcontrollers or to avoid large amounts of resources having to be provided to the driver. However the driver is also designed to provide very versatile TCP communications with multiple client and server sockets able to opened and used all at the same time. In order to allow the TCP driver to be used as needed for each specific end users application quite a low level interface is provided when creating and using client or server sockets.

TCP, like all of the stack process can only do one thing at once. When a TCP packet is received the application process that is using the socket must process and dump the packet, responding with a TCP tx packet if it wishes. Only 1 TCP packet may be received at any one time (other packets may be queued behind it in the nic's (network interface controller) receive buffer but won't be read until the previous packet is processed and dumped). When transmitting a TCP packet it must be sent as a single process. This isn't as limiting as it sounds, it simply means that any application process that has opened a socket must deal with packets received to that socket as soon as they arrive and when transmitting a packet must transmit a packet in one go. Multiple sockets may all have different processes going on concurrently the nic's receive buffers will automatically queue received packets for each socket.

TCP sockets are used to allow multiple (user defined in eth-tcp.h) sockets to be opened by application processes. A socket may act as a server waiting for an incoming connection, or as a client to connect to a remote device.

Sockets are opened on a specified local TCP port, and multiple sockets may be opened on the same port if required to allow connections to that port from multiple remote devices.

This TCP driver deals with much of the TCP background tasks automatically. TCP is a connection based protocol with automatic acknowledgement of communications and automatic timeout re-tries. The driver has one significant limitation however which is intentional to allow the driver to be used with limited resource processors or in applications where you don't want to designate large buffers of memory to the driver. When a TCP packet is sent to a remote device but not received by the remote device for some reason, it needs to be resent by the TCP driver. However a copy of the transmitted packet is not automatically saved by the TCP driver. This approach allows this situation to be dealt with on a per application basis. In applications with insufficient memory to buffer every packet sent until an acknowledge is received the application processes using each socket can maintain a method of re-generating the last packet sent on a particular socket (i.e. do exactly what it did to send the packet the first time again), or decide to simply dump the last packet sent in this situation and allow for this in the

applications communication protocol. Alternatively if the application does have sufficient memory then it can copy each packet sent to memory ready to be sent again should it be needed (this would require a buffer of 1460 bytes per TCP socket). This drivers approach allows either method to be used.

A third approach could have potentially been used whereby the memory of the nic (network interface controller IC) could have been used to keep a copy of previously sent packets until an acknowledge was received from the remote device. However nic's don't tend to have very large memory buffers and with potentially long timeout times for each socket, especially if communicating over the internet, this approach could effectively lock up the Ethernet interface due to a lack of available memory, potentially resulting in lost received packets and long wait times for outgoing packets on other sockets. Therefore this approach was deemed unsuitable for this TCP driver.

## Detailed TCP Driver Notes

Sequence Number

'Sequence Number' or 'SEQ' is the position (start address) of the current data block in the packet being sent within the overall data stream. If the value is 0 for the first packet which has 10 TCP data bytes then the sequence number for the next packet would be 10. The sequence number does not necessarily start from zero though and the 32bit value is used compared to the last value received to calculate the difference and therefore the position of the new data packet. The value wraps around when it reaches 0xFFFFFFFF but as this equals 4.3GB of data this is not an issue. This is why the sequence number is dealt with as a relative value (i.e. from last checkpoint) rather than absolute (i.e. from start of transfer).

The Sequence Number is sent by a device (server or client) to tell the other device the relative position of the packet of data being sent. Each device has its own Sequence Number and each device maintains a record of the last value received from the other device.

If a packet needs to be re-transmitted then the packet must be sent with the same Sequence Number as before.

Acknowledgement Number

'Acknowledgement Number' or 'ACK' indicates the total amount of data received. It is sent by a device (server or client) to tell the other device to total amount of data it has received from it. Each device has its own Acknowledgement Number and each device maintains a record of the new value to be sent to the other device. The Acknowledgement Number is valid when the ACK flag is set.

If the sequence number of a received packet is 200 and 10 bytes of data we're received in the packet then if the device was immediately responding it would return an Acknowledgement Number of 210.

TCP connection

To open a TCP connection a TCP packet is sent with the SYN flag set. The packet does not contain any data. However TCP counts the SYN marker as one byte so the Sequence Number is incremented by one. To close a TCP connection a TCP packet is sent with the FIN flag set. The packet does not contain any data. However TCP counts the FIN marker as one byte so the Sequence Number is also incremented by one. Packets are acknowledged by the other device, including SYN and FIN packets, with the return of a packet at some point with the correct Acknowledgement Number value.

A packet may be sent at any stage of a transfer whether any data has been received or not, but a packet with its Acknowledgement Number must be sent when the quantity of unacknowledged data approaches the Window size or if it appears that no more data will be received for a while. An Acknowledgement packet may also contain data if the acknowledging device has data to send to the other device.

Window Size

This is the limit specified by each device as to how much data the other device may send before an acknowledgement is returned. In this driver it is set to the size of 1 Ethernet packet to effectively remove the problems of receiving multiple packets out of order and having to re-assemble them. This slows down the transfer of bulk data, but greatly simplifies the driver and the need for large hardware memory resources or complex application receive functions.

## HOW THE HTTP SERVER DRIVER WORKS

An HTTP server (RFC 2616 defines HTTP/1.1 which is the version of HTTP in common use) is provided as part of the TCP/IP driver / stack for inclusion if your device needs to provide a web interface. The HTTP server in its basic form is relatively straightforward. However a useful HTTP server needs to be able to receive input from users and generate dynamic content as pages are served and the functionality to provide this makes the driver quite complex in places.

HTTP works using TCP. The HTTP server opens one or more TCP server ports on TCP port 80 and waits for clients to connect. When a client connects, by a user entering the IP address or NetBIOS name into their browser, the TCP driver acknowledges the connection and the HTTP driver waits for the client to send a request. There are three basic HTTP requests and this driver supports each of them:- GET, HEAD and POST. Whilst the HTTP specification includes other requests types, these are the three that provide most of the functionality for browsing the web, with other request types often not well supported by browsers or servers and not required to be implemented.

Once the TCP connection is made the client browser then typically sends a GET request to the HTTP server. This is simply a TCP packet containing ASCII text formatted as defined by the HTTP specification. If you use Wireshark to capture the HTTP packets sent and received they are all human readable text, for instance a typical GET request looks like:

GET /index.htm HTTP/1.0 <CR><LF>

User-Agent:Mozilla/4.5

Host: www.embedded-code.com<CR><LF>

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8<CR><LF>

Accept-Language: en-gb,en;q=0.5<CR><LF>

<and so on…>

The headers section must end with a blank line, so the last 4 bytes will always be:

<CR><LF><CR><LF> (<CR><LF> means the two bytes 0x0D, 0x0A)

After the message header is an optional message data section.

This driver takes a lean approach processing of HTTP request headers and will ignore many of the general headers. As it is generally safe to assume that the server will be used to send files that are compatible with all popular browsers the reading and storing of most header data is unnecessary.

When the HTTP driver receives the request, it looks to see if the file being requested is actually available and if it is it returns it. For an .htm HTML file this again is an ASCII text file that the HTTP server sends using one or more TCP packets. Typically an HTML file will typically include references to other files, such as images, that the clients browser then needs to also retrieve to add to the page that will be displayed on the browser. So the browser sends new requests to retrieve each of these and if they are available the HTTP driver also sends them in one of more TCP packets. In order to improve speed browsers will often attempt to open more than one TCP connection to a HTTP server when retrieving web pages so that multiple files may be transferred over the internet at the same time. If you define the HTTP driver to have multiple TCP sockets (in eth-http.h) it will allow the client to connect to as many of them as it wants and this can help improve overall delivery speed. If a browser tries to open more TCP connections than the HTTP server has the TCP driver will simply refuse the request and the browser will wait to download the additional files once already in progress files have completed downloading.

As the HTTP driver is sending HTML files it checks each byte looking for the special dynamic data marker described in the earlier HTML sections of this manual. Each time it finds this it reads the variable name and then calls the user application, allowing it to provide a dynamically generated string to be sent in place of the variable.

In addition to making requests the browser is also able to send inputs to the HTTP server, for instance in response to a user submitting a form or selecting an option. Responses may be sent as part of a normal GET request or using a POST request. In both instances the client browser specifies a file to be retrieved or activated but with data attached to the request to be processed first. The HTTP driver deals with these inputs in turn, passing each one fully decoded to the user application for it to store or process as desired. In the case of multipart form data and file uploads the data is passed to the user application byte by byte to allow for large data sizes and files.

### Error Responses

If the HTTP server receives an invalid request or cannot return the information requested it returns a standard HTML error response, which in its basic form (and as implemented in this driver) is simply plain ASCII text indicating the error type.

### HTTP Versions & RFC's

HTTP version 1.1 is specified in RFC2616. RFC1945 specifies the previous versions HTTP 1.0 and HTTP 0.9.

Version 1.1 adds capabilities for conserving network bandwidth, improving security and error notification plus some other more minor things. Typically embedded systems serve small, simple web pages and therefore gain little benefit from supporting HTTP 1.1. They therefore may use HTTP 1.0 for simplicity. A browser that supports HTTP 1.1 would have no trouble communicating with an HTTP server that supports HTTP 1.0. However one of the advantages of HTTP 1.1 is that it allows persistent connections. With HTTP 1.0 each request requires a new connection, so if a client requests a web page that has several links to images it has to be make a new connection for every page and every image, with all the handshaking overhead required by this. HTTP 1.1 on the other hand has a default behaviour of persistent connections where a TCP connection is left open until either the client or server decides that the connection is complete or the server chooses to close it after a period of no activity.

This driver supports HTTP 1.1 with support for the following methods:-

GET, HEAD, POST.

HTTP 1.1 includes some other less used methods but these are not required to be supported by a HTTP 1.1 server (only GET and HEAD must be supported).

## HOW THE ARP DRIVER WORKS

ARP (specified in RFC 826) is incorporated as part of the basic TCP/IP driver. ARP packets are specified by the ARP type code in an Ethernet packet header. Whenever the driver receives an ARP packet it automatically checks to see if the target IP address is the same as the drivers IP address. If it is and it is an ARP request the driver automatically sends an ARP response. If it is a response the driver stores the responding devices MAC address ready for the driver or user application function that requested ARP.

To trigger an ARP request the arp_resolve_ip_address() function is called with the IP address to be resolved. The calling function should then call the arp_is_resolve_complete() function periodically to see if the response has been received, implementing its own timeout should no response be received.

## HOW THE DHCP DRIVER WORKS

DHCP (specified in RFC 2131) allows the driver to automatically obtain the settings it needs on a network from a DHCP server. Large networks may include one or more dedicated servers that provide DHCP. Small home and many simple business networks will typically have DHCP provided by a broadband DSL router or by a PC that is providing internet connection sharing. Using DHCP in your embedded device allows it to be connected to a network and without any user configuration to obtain its settings.

DHCP operates automatically and the only intervention the user application needs to make is to set it as active or not on power up (and at any time later). If disabled the user application instead specifies the manual IP address, subnet mask and gateway address to be used, allowing the products user interface to provide the option of automatic or manual network settings if desired. DHCP may be enabled or disabled at any time and the driver will respond accordingly.

When enabled the DHCP driver will attempt to contact a DHCP server on the local network by broadcasting a DHCP discover packet periodically using UDP. If there is a DHCP server on the network it will respond to the discover packet with a DHCP offer packet, broadcasting the response using UDP (it can't sent the response directly as the driver has not yet been issued with an IP address). The offer packet will contain various fields, of which the following are read by the driver:

> Client IP address, which is the IP address being offered by the DHCP server
>
> Lease Time, which is the time period the driver may use this IP address for before it has to contact the DHCP server again with a new request.
>
> Subnet Mask
>
> Gateway, which is the IP address of the device on the local network that provides a connection onward to the next network (for instance the internet or the next level up of a corporate network). In the case of a router that is providing the DHCP server service this will be the same address as the DHCP server.

The driver stores all of these fields and the DHCP servers IP address and responds with a DHCP Request packet, again sent using a broadcast UDP packet. If defined the driver will also include an ASCII name field within the DHCP Request packet, which some DHCP servers will store for their name services.

The DHCP server finally confirms the process by returning a DHCP ACK packet, to the now assigned client IP address.

The DHCP driver has nothing else to do so sits idle, except for checking two timers that we're started when the DHCP offer was last received. The eth_dhcp_1sec_renewal_timer variable is set to expire half way through the DHCP servers issued lease time. Once this timer expires (typically after several hours or days) the DHCP driver starts attempting to contact the DHCP server again to repeat the original process and renew is settings. If for some reason the DHCP server doesn't respond after the many attempts that will have been made to contact it and the eth_dhcp_1sec_lease_timer variable expires, DHCP will change the status of the our_ip_address_is_valid variable to indicate that the IP settings are no longer valid and the TCP/IP driver must therefore remain offline until the DHCP server responds to the DHCP requests (that will continue to be sent reguarly).

## HOW THE NETBIOS DRIVER WORKS

The NetBIOS protocol (specified in RFC 1001 and RFC 1002) provides a simple method of searching for device by its name (up to 15 characters) on a local network. NetBIOS requests for the address of a queried name are sent as a broadcast UDP packet, and are therefore not passed through routers (routers do not forward broadcast packets).

The NetBIOS driver simply opens a UDP port and listens for broadcast packets received from anyone to its local port (the NetBIOS Nameservice Client Port 137). When a request packet is received the driver checks the query section to see if it contains a name which matches the name that has been set as the device name. If it does it replies to the sender with a NetBIOS response which includes the IP address.

## How The DNS Driver Works

DNS is specified RFC 1034 and 1035. RFC 1034 is more conceptual in nature and RFC 1035 is the one for programmers to read to understand the format of DNS communications. Its basic function is to translate domain names meaningful to humans into numerical IP addresses. The Domain Name System also stores other types of information, such as the list of mail servers that accept email for a given domain.

A DNS query is started by calling the do_dns_query() function with the domain name requiring resolving and the dns type (host or MX mailserver). As long as the DNS driver is not busy carrying out a prior DNS operation it will setup the DNS state machine to send a DNS request.

The driver sends an ARP request to retrieve the MAC address of the network gateway device (which will have been learnt via DHCP or manually set by the user). Once the MAC address is retrieved the driver then attempts to open a UDP socket and send a DNS request packet.

The request specifies the domain name that is to be resolved and includes the recursion desired bit to indicate to the gateway device that it needs carry out a recursive query, where it will fully answer the query rather than simply return a partial answer.

The DNS driver then waits for a response (checking for a possible timeout error). As the DNS response can use compression (compression of names by repeated names or sections of names being re-used later on in the packet) the driver doesn’t check the response names (as this is unnecessary and would involve a great deal of complexity and ram space). Instead it checks the answer and additional sections of the response looking for the IP address.

DNS may return several IP addresses (for instance for large sites) and in this case the driver selects the first one.

The function that requested the DNS resolve should periodically call the check_dns_response() function to determine if DNS has completed (or failed).

## How The POP3 Driver Works

The POP3 protocol (specified in RFC 1939) is a simple text-based protocol which is designed to support users with intermittent connections as well as permanent connections, so it is a perfect choice for many embedded devices requiring the ability to receive email. POP3 works over a TCP/IP connection using TCP server port 110.

When the process of retrieving email is started, by calling email_start_receive(), the POP3 state machine is setup to first take the URL of the POP3 server (defined as either a constant or variable string) and perform a DNS query, using the DNS driver.

Once the IP address of the POP3 server is returned the driver then sends an ARP request to retrieve the MAC address to communicate with. As a POP3 server will typically be on the internet and connected to via a gateway router of some form, the MAC address returned by the ARP driver will typically be the MAC address of the gateway, which will deal with forwarding communications on.

Once the MAC address is retrieved the driver then attempts to open a TCP connection with the POP3 server. This may take some time, if for instance the server is very busy.

Once the TCP connection is opened the server sends a greeting message. This is a text string that must start with “+OK” to be valid (the remainder of it can be anything). For example:

> “+OK Hello there”

The driver looks for the leading “+” and if its there it continues on. In order to tell the POP3 server which mailbox is to be accessed and to start the authentication process the driver sends the “USER” command followed by a space character and then the username (which is typically the email address of the mailbox). The line is terminated by a carriage return and line feed. For example:

> “USER me@yahoo.com<CR><LF>”

The server should then respond with a string that starts with “+OK” to indicate the mailbox is valid. For example:

> “+OK Password required. “

The driver looks for the leading “+” and if its there it continues on. The password command is now sent with the mailbox password. For example:

> “PASS mypassword <CR><LF>”

The server should again respond with a string that starts with “+OK” to indicate the password is accepted. For example:

> “+OK logged in. “

The driver looks for the leading “+” and if its there it continues on. Next the driver uses the STAT command to retrieve the number of emails that are in the mailbox. For example:

> “STAT<CR><LF>”

The server should respond with a fixed format string starting with “+OK”, then a single space, then the number of messages in the mailbox, a single space and finally the size of the mailbox in bytes. There may be additional characters after this but these are optional and are not required to follow a set format. For example:

> “+OK 2 3530”

The driver looks for the leading “+OK ” and then stores the following number of messages value. Each message is now retrieved as follows:-

> The driver sends a RETR command with the message number, for instance:
>
> > “RETR 1<CR><LF>”
>
> The server will then respond with the requested message. This will consist of one or more TCP packets which will contain a continuous block of data which starts with a block of message headers, a blank line and then the email body. The headers contain all sorts of information of which only 2 fields are of particular interest – the email sender (the reply address) and the email subject. When the subject is encountered the driver calls a user application function (defined by POP3_PROCESS_RECEIVED_EMAIL_LINE_FUNCTION), sending it the subject string (and also the senders email address). This indicates to the user application that this is the start of a new message being retrieved. The driver continues reading each of the headers and once it reaches the blank line that marks the transition from headers to email body it then starts calling the user application function with each line of the email body retrieved, for the user application to parse and process as required. Note that whilst the body may be plain text, much email is actually MIME formatted. If decoding of MIME formatting or reception of MIME encoded file attachments (which are included in the email body) is desired by the user application this must be provided by it. This driver simply passes each line as it is retrieved. Often for an embedded application all a user wants to do is

pass simple information such as configuration settings or commands and if the email is sent as plain text, even though it may still actually be MIME formatted, each line of the email text will be passed to the user application and can be parsed as desired.

The end of the email is indicated by:

“<CR><LF>.<CR><LF>”

(This 5 byte sequence is guaranteed not to appear anywhere within the contents of the email). One final call to the user application function is made by the driver indicating the end of the message. The return value from the function allows the user application to indicate to the driver if the message should be deleted or not.

If the message is to be deleted the driver sends the DELE command with the message number, for example:

“DELE 1<CR><LF>”

The driver looks for the leading “+OK ” in the response and then loops back to retrieve the next message.

Once all of the messages have been retrieved and optionally deleted the driver sends the QUIT command as shown:

“QUIT<CR><LF>”

It waits for the “+OK” response before then closing the TCP connection to the POP3 server and returning the POP3 driver state machine to the idle state.

Should an error occur at any point, for instance due to a timeout or unexpected response, the TCP connection is closed and the POP3 driver state machine is returned to the idle state.

If the user application wants to follow the progress of the POP3 process it can either monitor the state of the sm_pop3 variable or an optional string pointer is provided (selected using the DO_POP3_PROGRESS_STRING define) which is loaded with a user changeable string at each stage of the email receive process. This may be used for example to display progress on a devices screen.

## How The SMTP Driver Works

The SMTP protocol (first specified in RFC 821, most recently in SMTP 5321) is a simple text-based protocol which is designed to transfer email across IP networks. Although used by servers to send and receive messages, client devices typically only use SMTP for sending messages and use POP3 or IMAP to retrieve messages from a mail server.

SMTP works over a TCP/IP connection using TCP typically on server port 25.

When the process of sending email is started, by calling email_start_send(), the SMTP state machine is setup to first take the URL of the SMTP server (defined as either a constant or variable string) and perform a DNS query, using the DNS driver.

Once the IP address of the SMTP server is returned the driver then sends an ARP request to retrieve the MAC address to communicate with. As a SMTP server will typically be on the internet and connected to via a gateway router of some form the MAC address returned by the ARP driver will typically be the MAC address of the gateway, which will deal with forwarding communications on.

Once the MAC address is retrieved the driver then attempts to open a TCP connection with the SMTP server. This may take some time, if for instance the server is very busy.

Once the TCP connection is opened the server sends a greeting message. This is a text string that must start with 3 digits to be valid (the remainder of it can be anything). For example:

“220 some.server.name ESMTP”

The leading 2 character is the important one and it indicates the server is happy (3 is used for data transfers, 4 & 5 indicate an error). Once received the driver sends one of two commands, followed by a domain identifier string and then followed by a carriage return and line feed as shown:

“HELO EMB<CR><LF>”

or

“EHLO EMB<CR><LF>”

HELO is the normal command.

EHLO is the alternative command that requests an authenticated login. You may select either when calling the email_start_send() function, with an authenticated login generally required when using a SMTP server that is located remotely from the local internet connection (to protect against spam use). The domain identifier string can be anything.

The server should respond with a string starting with a 3 digit value. The first character should be “2” to indicate that the server is happy. If the 3 digit number is followed by a hyphen “-“ character then there are more lines to be read, each of which will start with the 3 digit number. Each line will be indicating a capability of the server and the driver reads each line until the final line, indicated by no trailing “-“ is read. For example:

“250-the.server.name<CR><LF>”

“250-AUTH=LOGIN CRAM-MD5 PLAIN<CR><LF>”

“250 8BITMIME<CR><LF>”

The driver will now jump over the following login, username and password stages if an authenticated login has not been selected, or if it has it will proceed as follows.

The driver first sends the AUTH LOGIN command, as shown:

“AUTH LOGIN<CR><LF>”

It waits to receive the response string, which should start with a “3” to indicate the server is in data transfer mode (4 & 5 indicate an error). For example:

“334 VXNlcm5hbWU7<CR><LF>”

The driver then sends the username encoded using BASE64. For example:

“ZGV2ZWxALnVrWHX<CR><LF>”

It waits to receive the response string, which should start with a “3” to indicate the server is still in data transfer mode and happy. For example:

“334 UGFzc3YAq8<CR><LF>”

The driver then sends the password encoded using BASE64. For example:

“C2VJlNkj3FuQ<CR><LF>”

It waits to receive the response string, which should start with a “2” to indicate the server is happy. For example:

“235 go ahead<CR><LF>”

Now that the login has been completed the driver continues on in the same way if either authenticated login was select or not. The driver starts the send process with the MAIL FROM command, followed by a space character and then the email address the email is being sent from enclosed in angle brackets, for example:

“MAIL FROM: <embedded-device1@my-domain.com><CR><LF>”

It waits to receive the response string, which should start with a “2” to indicate the server is happy. For example:

“235 ok<CR><LF>”

The driver now sends the RCPT TO command, followed by a space character and then the destination email address enclosed in angle brackets, for example:

“RCPT TO: <john@yahoo.com><CR><LF>”

It waits to receive the response string, which should start with a “2” to indicate the server is happy. For instance:

“235 ok<CR><LF>”

The driver now sends the DATA command:

“DATA<CR><LF>”

It waits to receive the response string, which should start with a “3” to indicate the server is now in data transfer mode. For example:

“354 go ahead<CR><LF>”

The email message is now transferred to the server as a continuous block of data until it is complete. No response is given by the server until the transfer is complete (apart from TCP acknowledgements).

The first section of the message is the headers, which includes the FROM email address, which is the senders email address again (these fields will be read and used by the receiver of the email), the TO email address, which is the recipients email address again, and the email subject.

After this the email body is sent, the start of which is indicated by a blank line. As this driver allows a file to be attached emails are sent MIME encoded. Therefore first of all a constant MIME header block is sent to indicate the start of the text portion of the email. Once this has been sent a special user application function is called to request each byte of the email text. As TCP packets may need to be resent should a packet be lost, a variable is passed to the user application which is normally zero, but if not indicates that the application needs to be move back the specified number of bytes as the driver is resending the previous packet. Once the user application has passed all of the bytes of the email text it returns a value to indicate that it is complete.

If the user flagged that the email would include a file attachment when email_start_send() was originally called the driver now sends a MIME header for the file attachment portion of the email body. Once this has been sent the special user function is now called to request each byte of the file. Any type of file may be sent and the driver automatically BASE64 encodes the file so that binary data is suitable for sending via SMTP. Again, as TCP packets may need to be resent should a packet be lost, a variable is passed to the user application which is normally zero, but if not indicates that the application needs to be move back the specified number of bytes as the driver is resending the previous packet of the file data. Once the user application has passed all of the bytes of the file it returns a value to indicate that it is complete.

The driver sends the final MIME block and completes the email with the <CR><LF>.<CR><LF> marker.

It waits to receive the response string, which should start with a “2” to indicate the server is happy. For instance:

“250 ok<CR><LF>”

Finally the driver sends the QUIT command:

“QUIT<CR><LF>”

It waits for the response before then closing the TCP connection to the SMTP server and returning the SMTP driver state machine to the idle state.

Should an error occur at any point, for instance due to a timeout or unexpected response the TCP connection is closed and the SMTP driver state machine is returned to the idle state.

If the user application wants to follow the progress of the SMTP process it can either monitor the state of the sm_smtp variable or an optional string pointer is provided (selected using the DO_SMTP_PROGRESS_STRING define) which is loaded with a user changeable string at each stage of the email send process. This may be used for example to display progress on a user screen.

## HOW THE ICMP DRIVER WORKS

ICMP (which is specified in RFC 792) is a core protocol of the Internet Protocol Suite. Whilst the protocol provides such functions as error reporting, for an embedded device the only functionality that is typically required is responding to ping ICMP Echo requests. ICMP packets are marked with their own IP protocol value and when one is received the driver checks to see if it is an Echo request sent to its IP address. If it is the driver automatically responds with an ICMP reply, returning the same block of data received.

## HOW THE SNTP DRIVER WORKS

SNTP (which is specified in RFC 4330) is a simple protocol which allows devices to request the current time from a public NTP server. It is a less complex form of NTP and does not require the storing of information about previous communications so is ideal for embedded devices in applications where high resolution / accuracy timing is not required.

The response from a SNTP / NTP server is a 32 bit value which is the number of seconds since 00:00:00 on 1 January 1900. This value can then be used by the application and a typical example of usage is given in the ‘Adding SNTP Functionality’ section earlier in this manual.

To send a SNTP request the user application simply needs to call the sntp_get_time() function. The SNTP driver will then automatically use DNS to lookup the IP address of the SNTP server. If you are using a NTP service such as pool.ntp.org DNS will often retrieve a different IP address on successive calls, as these public NTP services typically distribute requests around a pool of NTP servers. Once the IP address is retrieved the driver will send an ARP request to obtain the MAC address for the server, which will typically return the MAC address of the local network gateway device that will forward the communications on to the actual SNTP server.

The driver then opens a UDP socket and transmits a SNTP request packet, followed by an immediate call of the user application function defined by SNTP_USER_AP_FUNCTION in eth-sntp.h. This marks the exact moment the request is sent to the SNTP server and allows the user application to start a timeout timer if desired so that should the response not be received quickly enough, due to internet delays, it can determine this. The driver then waits for the response and as soon as it is received it again calls the SNTP_USER_AP_FUNCTION function with the received time value.

The SNTP driver incorporates timeout timers for each of the steps of the SNTP process and the user application function is notified if a timeout should occur and the SNTP driver has given up.

If higher accuracy timing is required then it is a good idea to select a SNTP / NTP server to send the request to based on geographical location, so that less time is taken for the request and response to travel across the internet.

# TROUBLESHOOTING

## GENERAL TROUBLESHOOTING TIPS

Some common troubleshooting tips:

Double check the IO pin definitions in the drivers nic.h header file are correct

Verify with a scope that all of the control and data pins to the nic are working correctly.

Check that no other device on a serial or parallel data bus to the network interface controller IC is outputting while the driver is trying to communicate with the nic.

Check that your microcontroller is not resetting due to a watchdog timer timeout.

See the:

‘Signal Noise Issues With MMC & SD Memory Cards (& Clocked Devices In General)’

page in the resources area of our web site for details of a common signal noise problem experienced when using clocked devices such as network interface controllers.

If you are using a 32bit device ensure that for the driver files WORD = 16 bits and DWORD = 32 bits.

If you suddenly find that you don’t get a response from a remote server or router, for instance when trying to log onto a POP3 or SMTP server or requesting a DNS response from your networks router, bear in mind that you may deliberately not be getting a response. While debugging you can easily upset servers by, for instance, break pointing code and not letting a TCP connection be properly closed at the end of an operation. Alternatively your router may have built in protection mechanisms enabled to protect against denial of service (DOS) attacks which cause it to start ignoring repeated requests for the same thing. Typically waiting a little while will solve the problem as these types of protection mechanisms are automatically reset by a timer. Alternatively you could try hard resetting your router or temporarily using an alternative POP3 or SMTP server.

# REVISION HISTORY

## CHANGES TO THE TCP/IP DRIVER FILES

### V1.00

First released version.

### V1.01

ARP handling minor error fix

In eth-arp.c file, arp_process_rx function

All of the initial if(! checks should return '1'. Previously 2 off returned '0'.

In eth-main.c, tcp_ip_process_stack function.

Locate case SM_ETH_STACK_ARP:

And add this before the break; statement:

nic_rx_dump_packet(); //Ensure packet has been dumped

NetBIOS handling bad initialisation fix

In eth-netbios.c, process_netbios_nameservice function

Change the following line:

static BYTE our_udp_socket;

to this:

static BYTE our_udp_socket = UDP_INVALID_SOCKET;

All sizeof() references to struct definitions changed to use a defined length constant. All array read and writes from and to struct's changed to individual read and writes to the struct members. This is to deal with 32bit compilers which may add pad bytes within a struct (for instance sizeof(MAC_ADDR_LENGTH) will often be 8 not 6 due to the addition of pad bytes).

DHCP timeouts (DHCP_DISCOVER_TIMEOUT and DHCP_REQUEST_TIMEOUT) changed from 4 seconds to 10 seconds to deal with routers / DHCP servers that may be too busy to respond immediately.

CrossWorks ARM compiler with NXP LPC2000 series 32bit ARM microcontroller support added.

### V1.02

Added HTTP client functionality.

Added PIC32 built in Ethernet peripheral support.

At the end of the six email_return_smtp_ functions in eth-smtp.c and three email_return_pop3_ functions in eth-pop3.c changed the end of the do loop from:

```
    while (p_source_string++ != 0x00);
```

to

```
    while (*p_source_string++ != 0x00);
```

Correction for NXP LPC2000 nic.c driver file. The interrupt line from the KSZ8001 phy used in the sample project would not be cleared. Corrected in nic_check_for_rx() by replacing the following line:

```
    nic_write_phy_register(NIC_PHY_IRQ_CTRL_STATUS_REG, (NIC_PHY_IRQ_CTRL_STATUS_VALUE |
0x00ff));
```

with

```
    data = nic_read_phy_register (NIC_PHY_IRQ_CTRL_STATUS_REG);
```

In process_http() added check for a client disconnecting a socket part way through downloading a file (sometimes a browser may close a socket to stop downloading any further data).

Change this:

```
			//----- PROCESS EACH SOCKET -----
			switch (http_socket[socket_number].sm_http_state)
```

To this:

```
			//----- PROCESS EACH SOCKET -----
			if(!tcp_is_socket_connected(http_socket[socket_number].tcp_socket_id))			//If
a socket has disconnected during a transfer reset its state
				http_socket[socket_number].sm_http_state =
HTTP_WAITING_FOR_CONNECTION;

			switch (http_socket[socket_number].sm_http_state)
```

Minor improvements to the DNS functions.

Fixed TCP issue whereby if server sends FIN ACK with data the TCP stack was not dealing with the FIN. It now closes the connection as it should.

Added check for LPC2000 driver KSZ8001 PHY link is down when link status changes to avoid long delay before checks timeout when cable is disconnected.

Changed all .val to .Val to provide easier compatibility with Microchip C32 compiler which now clashes with .val

http_transmit_next_response_packet() function updated with new file_offfset variable to fix bug when using HTTP_USING_BINARY_FILES or HTTP_USING_FILING_SYSTEM defines.

## V1.03

Minor correction of ".val" usage to ".Val"

## V1.04

In eth-http.c corrected issue where http_setup_response would be called twice by modifying this:

```
			if (http_post_content_bytes_remaining)
			{
				//----------------------------------------------------------------------------------------------------
				//----- THERE IS MORE CONTENT TO BE RECEIVED FOR THIS POST REQUEST
IN SUBSEQUENT TCP PACKETS -----
				//----------------------------------------------------------------------------------------------------
```

```
            //Exit now and flag new socket state as still receiving a post request (this will also
block other post requests until its complete)
            http_socket[socket_number].sm_http_state = HTTP_PROCESSING_POST;
            tcp_dump_rx_packet();
            return;
        }

    }
#endif
```

to this:

```
        if (http_post_content_bytes_remaining)
        {
            //-----------------------------------------------------------------------------------------------
            //----- THERE IS MORE CONTENT TO BE RECEIVED FOR THIS POST REQUEST
IN SUBSEQUENT TCP PACKETS -----
            //-----------------------------------------------------------------------------------------------
            //Exit now and flag new socket state as still receiving a post request (this will also
block other post requests until its complete)
            http_socket[socket_number].sm_http_state = HTTP_PROCESSING_POST;
            tcp_dump_rx_packet();
        }
        return;
    }
#endif
```

Fixed tcp client issue in eth-http-client.c

Customer provided driver for the Stellaris LM3S6965 NIC added.

Designed by:

IBEX UK Limited
32A Station Road West
Oxted
Surrey
RH8 9EU
England
Tel: +44 (0)1883 716 726
E-mail: info@ibexuk.com
Web: www.ibexuk.com