Development Kit For the PIC® MCU

Exercise Book

3.3V Ethernet Controller

March 2010

CCS Inc.

Custom Computer Services, Inc.
Brookfield, Wisconsin, USA
262-522-6500

Recognized Microchip Third-Party Tool Provider

Custom Computer Services, Inc. proudly supports the Microchip brand with highly optimized C compilers and embedded software development tools.

Inventory

❑ Use of this kit requires a PC with Windows 95, 98, ME, NT, 2000 or XP. The PC must have a spare 9-Pin Serial or USB port, a CD-ROM drive and 75 MB of disk space.

❑ The diagram on the following page shows each component in the 3.3V Ethernet Controller Kit. Ensure every item is present.

Software

❑ Insert the CD into the computer and wait for the installation program to start. If your computer is not set up to auto-run CDs, then select **My Computer** and double-click on the CD drive.

❑ Click on **Install** and use the default settings for all subsequent prompts by clicking NEXT, OK, CONTINUE…as required.

❑ Identify a directory to be used for the programs in this booklet. The install program will have created an empty directory **c:\program files\picc\projects** that may be used for this purpose.

❑ Select the compiler icon on the desktop. In the PCW IDE, click **Help>About** and verify a version number is shown for the IDE and PCM to ensure the software was installed properly. Exit the software.

Hardware

❑ Connect the PC to the ICD(6) using the USB cable.[1] Connect the prototyping board (9) to the ICD using the modular cable. Plug in the DC adaptor (8) to the power socket and plug it into the prototyping board (9). The first time the is connected to the PC, Windows will detect new hardware. Install the USB driver from the CD or website using the new hardware wizard. The driver needs to be installed properly before the device can be used.

❑ The LED should be red[2] on the ICD-U to indicate the unit is connected properly.

❑ Run the Programmer Control Software by clicking on the CCSLOAD icon on the desktop. Use CCSLOAD Help File for assistance.

❑ The software will auto-detect the programmer and target board and the LED should be illuminated green. If any errors are detected, go to Diagnostic tab. If all tests pass, the hardware is installed properly.

❑ Disconnect the hardware until you are ready for Chapter 3. Always disconnect the power to the Prototyping board before connecting/disconnecting the ICD or changing the jumper wires to the Prototyping board.

[1] ICS-S40 can also be used in place of ICD-U. Connect it to an available serial port on the PC using the 9 pin serial cable. There is no driver required for S40.

[2] ICD-U40 units will be dimly illuminated green and may blink while connecting.
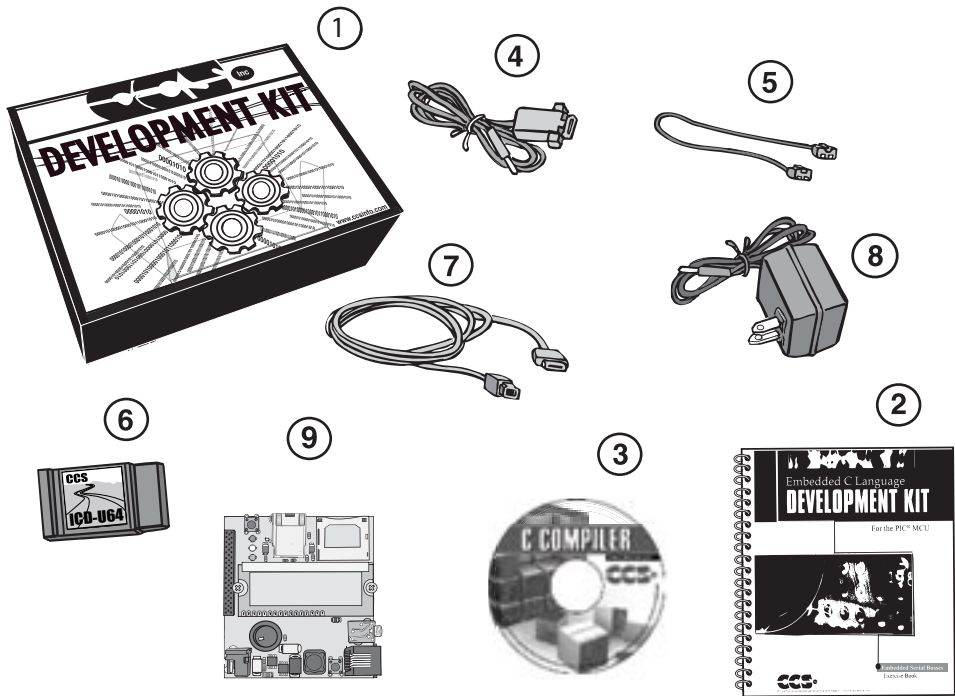
(1) Storage box
(2) Exercise booklet
(3) CD-ROM of C compiler (optional)
(4) Serial PC to Prototyping board cable
(5) Modular ICD to Prototyping board cable
(6) ICD unit for programming and debugging
(7) USB (or Serial) PC to ICD cable
(8) **A**C Adaptor (9VDC)
(9) 3.3V Ethernet Controller Prototyping Board with LCD Display
   (See inside front and back cover for details on the board layout and schematic)

2 USING THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Editor

❑ Open the PCW IDE. If any files are open, click **File>Close All**

❑ Click **File>Open>Source File**. Select the file: **c:\program files\picc\examples\ex_ stwt.c**

❑ Scroll down to the bottom of this file. Notice the editor shows comments, preprocessor directives and C keywords in different colors.

❑ Move the cursor over the **Set_timer0** and click. Press the F1 key. Notice a Help file description for set_timer0 appears. The cursor may be placed on any keyword or built-in function and F1 will find help for the item.

❑ Review the editor special functions by clicking on **Edit**. The IDE allows various standard cut, paste and copy functions.

❑ Review the editor option settings by clicking on **Options>Editor Properties**. The IDE allows selection of the tab size, editor colors, fonts, and many more. Click on **Options>Toolbar>Keyboard Setup** to select which icons appear on the toolbars.
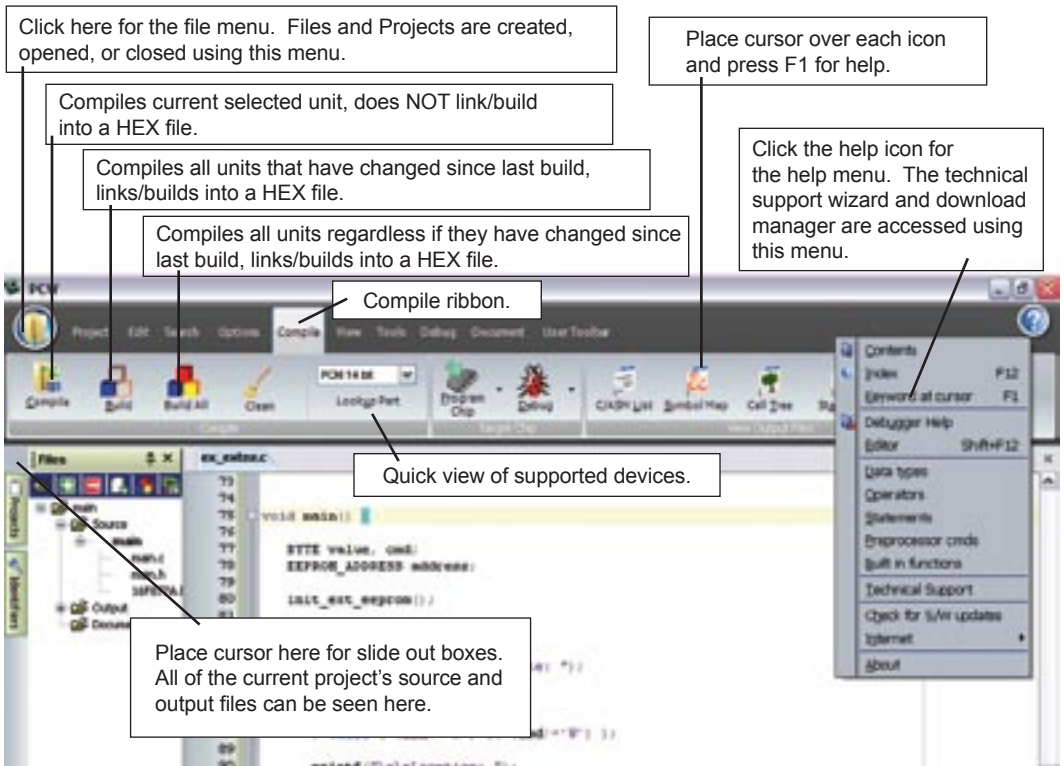
Compiler

❑ Use the drop-down box under Compile to select the compiler. CCS offers different compilers for each family of Microchip parts. All the exercises in this booklet are for the PIC18F67J60 chip, a 16-bit opcode part. Make sure **PCH 16 bit** is selected in the drop-down box under the **Compiler** tab.

❑ The main program compiled is always shown in the bottom of the IDE. If this is not the file you want to compile, then click on the tab of the file you want to compile. Right click into editor and select **Make file project**.

❑ Click **Options>Project Options>Include Files…** and review the list of directories the compiler uses to search for included files. The install program should have put two directories in this list to point to the device: *.h files and the device drivers*.

❑ Normally the file formats need not be changed and global defines are not used in these exercises. To review these settings, click **Options>Project Options>Output Files** and **Options>Project Options>Global Defines**.

❑ Click the compile icon to compile. Notice the compilation box shows the files created and the amount of ROM and RAM used by this program. Press any key to remove the compilation box.

Viewer

❑ Click **Compile>Symbol Map**. This file shows how the RAM in the microcontroller is used. Identifiers that start with @ are compiler generated variables. Notice some locations are used by more than one item. This is because those variables are not active at the same time.

❑ Click **Compile>C/ASM List.** This file shows the original C code and the assembly code generated for the C. Scroll down to the line:

```
int_count=INTS_PER_SECOND;
```

❑ Notice there are two assembly instructions generated. The first loads 4C into the W register. INTS_PER_SECOND is #defined in the file to 76. 4C hex is 76 decimal. The second instruction moves W into memory. Switch to the Symbol Map to find the memory location where int_count is located.

❑ Click **View>Data Sheet**, then **View.** This brings up the Microchip data sheet for the microprocessor being used in the current project.

Click here for the file menu. Files and Projects are created, opened, or closed using this menu.

Place cursor over each icon and press F1 for help.

Compiles current selected unit, does NOT link/build into a HEX file.

Click the help icon for the help menu. The technical support wizard and download manager are accessed using this menu.

Compiles all units that have changed since last build, links/builds into a HEX file.

Compiles all units regardless if they have changed since last build, links/builds into a HEX file.

Compile ribbon.

Quick view of supported devices.

Place cursor here for slide out boxes. All of the current project's source and output files can be seen here.

❑ Open the PCW IDE. If any files are open, click **File>Close All**

❑ Click **File>New>Source File** and enter the filename **EX3.C**

❑ Type in the following program and **Compile.**

```
#include <18F67J60.h>
#fuses  HS,NOWDT
#use delay(clock=25000000)

#define GREEN_LED PIN_A5

void main () {
    while (TRUE) {
        output_low (GREEN_LED);
        delay_ms (1000);
        output_high (GREEN_LED);
        delay_ms (1000);
    }
}
```

N O T E S

- The first four lines of this program define the basic hardware environment. The chip being used is the PIC18F67560, running at 25MHz with the ICD debugger.

- The #define is used to enhance readability by referring to GREEN_LED in the program instead of PIN_A2.

- The "while (TRUE)" is a simple way to create a loop that never stops.

- Note that the "output_low" turns the LED on because the other end of the LED is +5V. This is done because the chip can tolerate more current when a pin is low than when it is high.

- The "delay_ms(1000)" is a one second delay (1000 milliseconds).

- Do not add device ICD=true if you are going to generate a stand-alone program that does not need an ICD for debugging.
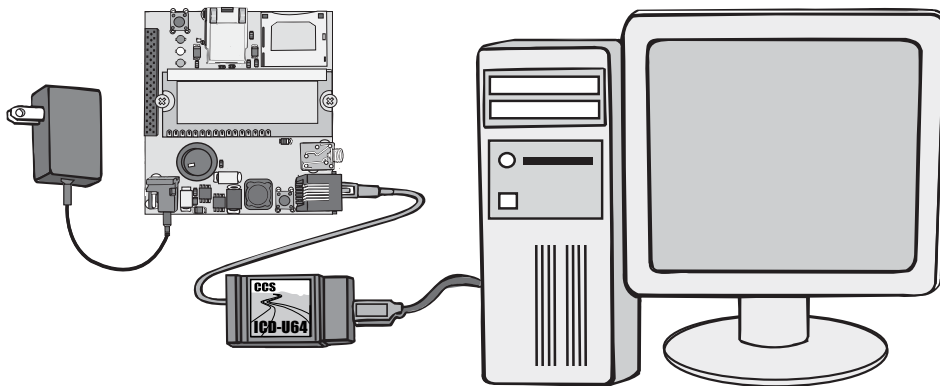
☐ Connect the ICD to the Prototyping board using the modular cable, and connect the ICD to the PC. Power up the Prototyping board.

☐ Click **Debug>Enable Debugger** and wait for the program to load.

☐ If you are using the ICD-U40 and the debugger cannot communicate to the ICD unit go to the debug configure tab and make sure ICD-USB from the list box is selected.

☐ Click the green go icon:

☐ Expect the debugger window status block to turn yellow indicating the program is running.

☐ The green LED on the Prototyping board should be flashing. One second on and one second off.

☐ The program can be stopped by clicking on the stop icon:

FURTHER STUDY

A *Modify the program to light the green LED for 5 seconds, then the yellow for 1 second and the red for 5 seconds.*

B *Add to the program a #define macro called "delay_seconds" so the delay_ms(1000) can be replaced with : delay_seconds(1); and delay_ms(5000) can be: delay_seconds(5);.*

Note: *Name these new programs EX3A.c and EX3B.c and follow the same naming convention throughout this booklet.*

4 ▸ NETWORKING OVERVIEW

❑ TCP/IP is the foundation on which many networks, such as the Internet, operate. However TCP/IP is not just one protocol, but a combination of several protocols stacked on top of each other. Just by the name TCP/IP, it is inferred that there is a TCP protocol that operates on top of the IP protocol. Below is an example of a stack of protocols that make up HTTP (is the protocol used to send and receive web pages) over an Ethernet network connection or a modem network connection:

Stack Component	Ethernet	Modem
Application	HTTP	HTTP
Transport & Session	TCP	TCP
Network	IP	IP
Data Link	Ethernet	PPP
Physical Link	10Base-T	Phone Line

❑ The physical link defines the physical connection and properties that connect to a network. Common properties are cable type, pin-outs, data rates, max distances, and so on. In Ethernet networks the physical network is the twisted pair cabling. When using a modem to dial an Internet Service Provider (ISP), the physical link is phone line.

❑ The data link defines the protocol used to maintain the physical link, which may include: data framing, checksum/CRC, and collision detection. The data link layer is divided into two parts: media access control (MAC) and link layer control (LLC). MAC controls access and encodes signals to a valid format. LLC creates a link to the network via low level link negotiation.

❑ The network layer provides address and routing information for the packet. The network protocol primarily used on the Internet is IP. Addresses are provided in IP via a 4-byte denomination for example 192.168.100.1. IP can route incoming and outgoing packets by inspecting the IP address in the protocol and determine the best route for such packets. Another network layer often seen in Ethernet is ARP, which is used to resolve MAC addresses with IP address.

❑ The transport and session are actually separate layers, but for simplicity, shall be combined in this tutorial. The transport layer provides for error checking, error recovery and data flow control. The session layer provides a method for creating a communication session between two points, and may include security and authentication.

❑ The application layer employs user defined data and protocols. In the case of the web, asking for and transmitting web pages are done through a protocol called HTTP, which sits in the application layer. Other protocols, such as TELNET and FTP, also sit in the application layer.

❑ As a packet progresses through the a network, such as the Internet, all machines involved in the routing from point A to point B may change the contents of the link layer and network layer as needed. For example, a PC may be connected to the Internet via a modem with a PPP connection, and therefore, all packets originating from that PC will use PPP for the link layer. However, not all machines are connected to the Internet using PPP, so as a packet is sent through the network each unit will use their own link layer protocol.

❑ Over the next few chapters of this tutorial, these layers and how to implement them in firmware will be discussed.

❑ A great way to learn to TCP/IP, and the underlying foundation of all layers involved, is to use a packet sniffer. A packet sniffer can inspect packets as they enter and leave a PC. A free, open-source, packet sniffer in software is available called Wireshark, and can be downloaded at http://wireshark.org. Below is a screen-shot of Wireshark inspecting an ARP packet on Ethernet:
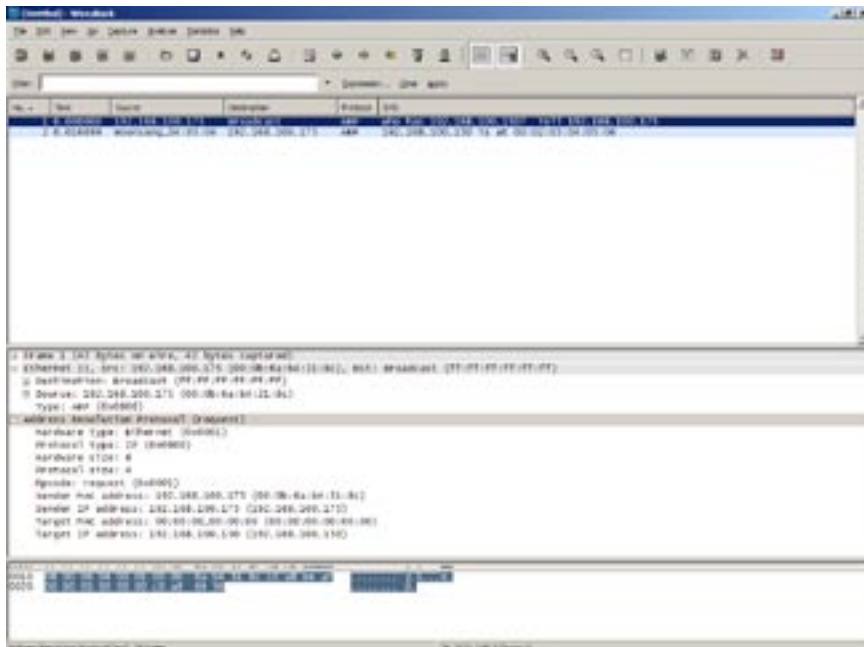
5 ▸ COOPERATIVE MULTITASKING

❑ When implementing TCP/IP and other networking protocols, portions of the code will require waiting for a connection, response, or acknowledgement. Often there are many network requests that need to be handled; sitting in a loop waiting for a response may not be acceptable. For this reason, a multitasking scheme must be employed when implementing a TCP/IP stack on the PIC. The easiest multitasking scheme to implement on a microcontroller, such as a Microchip PIC MCU, is a cooperative multitasking scheme.

❑ Cooperative multitasking is where processes must give control back to other processes. It is called cooperative because all tasks and processes must cooperate and give back control for other processes.

❑ The following is an example program that is not cooperative:

```
void blink _ leds(void);
void handle_input(void);

void main(void) {
    while(TRUE) {
        blink_led();
        handle_input();
    }
}

void blink_led(void) {
    output_low(PIN_LED);
    delay_ms(1000);
    output_high(PIN_LED);
    delay_ms(1000);
}

void handle_input(void) {
    //get input from user and act upon it
}
```

❑ blink_leds() is not a routine that is compatible with cooperative multitasking schemes because it will take over the microcontroller for two seconds. During that two seconds, other tasks are not running, including the function handle_input(). If a button is pressed in the middle of the blink_leds() routine, it may take up to two seconds before seeing the microcontroller act upon that input.

❑ The following is a replacement for blink_leds() that is cooperative multitasking friendly:

```
void blink _ led _ task(void) {
    static enum {LED_TASK_TOGGLE, LED_TASK_WAIT} state=LED_TASK_TOGGLE;
    static TICKTYPE last_counter;

    switch(state) {
        case LED_TASK_TOGGLE:
            output_toggle(PIN_LED);
            state=LED_TASK_WAIT;
            last_counter=TickGet();
            break;

        case LED_TASK_WAIT:
            if (TickGetDiff(TickGet(), last_counter) > TICKS_PER_SECOND) {
                state= LED_TASK_TOGGLE;
            }
            break;
    }
}
```

❑ Unlike blink_leds(), blink_leds_task() has no delay() routines, so it operates very fast. Since it operates fast, it is cooperative with other tasks in the system. Updating blink_leds() to blink_leds_task() does involve more overhead, as more RAM is needed to save the previous state of the task and more code space is needed to continue from the previous state of the task. Note that other tasks in the program, such as handle_input() in the example, have to be cooperative also.

❑ TICKTYPE, TickGet() and TickGetDiff() are part of the timing system. TICKTYPE is a typedef to define the datatype used for the counter. TickGet() gets the current timing value; the current timing value is automatically incremented by the timer task or the timer interrupt. The timing value increments in such a way that the number of ticks per second is defined by the TICKS_PER_SECOND constant. TickGetDiff() finds the difference between two counter values, in order to find if a certain interval of time has happened. This nomenclature was used for this example because this is the timing system that Microchip has employed for their TCP/IP stack.
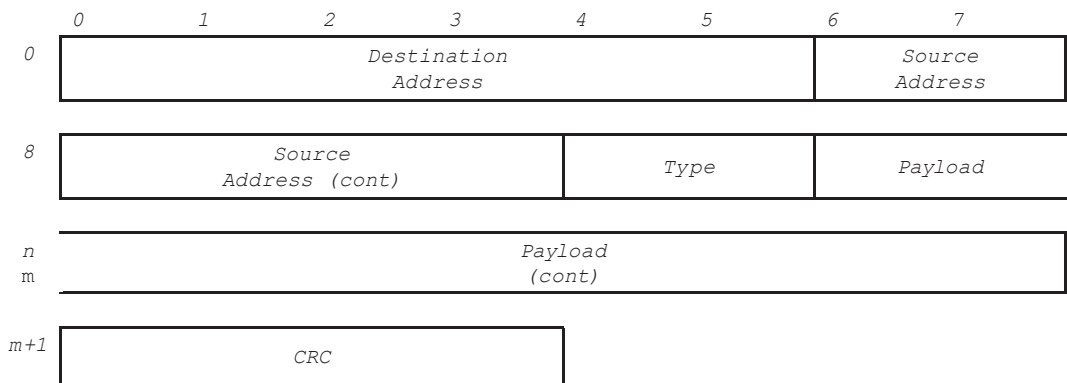
6 THE MICROCHIP TCP/IP STACK OVERVIEW

❑ The TCP/IP stack used in this tutorial is a modified version of Microchip's TCP/IP stack. CCS made modifications in porting the stack to compile under the CCS PCH Compiler, and also added PPP as a possible physical/link layer. Despite these modifications the API remains relatively unchanged, and Microchip provides a documentation of the API in application note AN833.

❑ The heart of the Microchip TCP/IP stack is the function StackTask(). StackTask() is a cooperative multitasking friendly routine that handles all the tasks for the stack including: Ethernet, PPP, IP, ARP, UDP, timing engine, and so forth.

❑ When StackTask() is called in the main loop, StackTask() will process all components of the stack and get the stack ready for the next interation of the user code. For example, if there is a new TCP packet in the Ethernet receive buffer, StackTask() will process the Ethernet, IP and TCP task which will result in TCPIsGetReady() returning TRUE. In the user task, inspect the TCPIsGetReady() after each StackTask() to determine if there is any action that needs to be taken with the received data. The next time StackTask() is called, all remaining data in the receive buffer is discarded and the next set of data in the receive buffer is processed.

7   ETHERNET LAYER

❑ This chapter will start to look at the TCP/IP API. In particular, the Ethernet layer will be reviewed. In general, it is not necessary to know the underlying foundation of a TCP/IP network. To skip the internals of TCP/IP, Chapter 13 outlines how to begin creating an application.

❑ Not every node connected to a network or the Internet is connected using Ethernet. However, it is the physical layer of most home and office local area networks (LAN); and this tutorial will focus on that aspect.

❑ The Ethernet packet is formatted as follows:

	0	1	2	3	4	5	6	7
0	Destination Address						Source Address	
8	Source Address (cont)				Type		Payload	
n m	Payload (cont)							
m+1	CRC							

❑ The Source and Destination address is the 6 byte Media Access Control (MAC) address, and each device is given a unique address. The first three bytes of the MAC address are the hardware vendor of the device, the second three bytes of the MAC address are often the unit's serial number.

❑ The type field is used in two ways. If the value of type is greater than 1500 bytes, the packet is an Ethernet 2 frame and type field represents the protocol ID of the data. The two protocols used in this book will be IP (0x0800) and ARP (0x0806). If the value of type is less than 1500 bytes, then the packet is an IEEE 802.3 frame and type field represents the length of the data.

❑ Data is the data being sent by the Ethernet packet, and can represent many types of packets. The only types of packets that we will care about will be ARP and IP packets. The minimum length of an Ethernet frame is commonly 64 bytes, therefore, if there is less than 64 bytes of data, the data field will be padded with invalid bytes.

❑ The CRC field is used by the link layer to determine if there is an error in the packet, and the link layer will use this to automatically discard packets that are not valid. The CRC will be generated by the TCP/IP stack, and often the CRC is automatically generated by the network interface card (NIC).

❑ The Source and Destination address is the 6 byte Media Access Control (MAC) address, and each device is given a unique address. The first three bytes of the MAC address are the hardware vendor of the device, the second three bytes of the MAC address are often the unit's serial number.

❑ The type field is used in two ways. If the value of type is greater than 1500 bytes, the packet is an Ethernet 2 frame and type field represents the protocol ID of the data. The two protocols used in this book will be IP (0x0800) and ARP (0x0806). If the value of type is less than 1500 bytes, then the packet is an IEEE 802.3 frame and type field represents the length of the data.

```
#include <18F67J60.h>
#use delay(clock=25000000)
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)
#fuses HS, NOWDT, NODEBUG
#define STACK_USE_CCS_PICEEC    TRUE

#define STACK_USE_MCPINC    TRUE                //use the nic card

#include "tcpip\stacktsk.c"                     //include Microchip stack

void MACDisplayHeader(MAC_ADDR *mac, int8 type) {
    int8 i;
    printf("\r\nMAC: ");
    for (i=0;i<6;i++) {
        printf("%X", mac->v[i]);
        if (i!=5)
            putc(':');
    }
    printf("  PROT:0x08%X ",type);
    if (type==MAC_IP)
        printf("[IP]");
    else if (type==MAC_ARP)
        printf("[ARP]");
}

(continued...)
```

```
(continued...)

void main(void) {
    MAC_ADDR mac;
    int8 type;

    printf("\r\n\nCCS TCP/IP TUTORIAL\r\n");

    MACInit();

    while(TRUE) {
        if (MACGetHeader(&mac, &type)) {
            if (type!=MAC_UNKNOWN) {
                MACDisplayHeader(&mac, type);
            }
            ();
        }
    }
}
```

❑ Compile and run on the prototyping board.

❑ Inspect the output on a serial terminal program. The messages should appear as follows:

```
CCS TCP/IP TUTORIAL
MAC: 00:0B:6A:B4:31:8C  PROT:0x0800 [IP]
MAC: 00:D0:59:7F:23:A8  PROT:0x0800 [IP]
MAC: 00:09:5B:E1:30:E2  PROT:0x0800 [IP]
MAC: 00:C0:B6:02:92:BD  PROT:0x0800 [IP]
MAC: 00:03:6D:1D:53:9A  PROT:0x0806 [ARP]
MAC: 00:C0:B6:02:E2:CA  PROT:0x0800 [IP]
```

❑ The MAC displayed is the source MAC address of the unit sending the Ethernet packet.
   PROT is the protocol field of the Ethernet header, and it is filtered by Microchip's TCP/
   IP stack to only allow IP and ARP packets. The NIC automatically filters out Ethernet
   packets that are not destined to the unit, and so it can be assumed that the Ethernet
   packet either had the Destination MAC address or the Ethernet packet was a broadcast
   packet. Broadcast packets are made with a MAC address of FF:FF:FF:FF:FF:FF

NOTES

- MACInit() initializes the ethernet module built into the PIC18F67J60, which includes setting the MAC address and initializing buffers.

- MACGetHeader(*MAC, *type) checks the NIC's receive buffers. If there is data in the receive buffer, MACGetHeader will return TRUE and update the pointer MAC and type. Type will also indicate if it's an IP, ARP or unknown packet. Unknown packets should be thrown away.

- When using the full TCP/IP stack provided by Microchip, MACGetHeader() is called automatically by StackTask() to process any incoming IP and ARP packets. Microchip's TCP/IP stack clears the NIC receive buffer after each StackTask(). Therefore, after each StackTask(), any incoming messages within one task must be processed or the data will be lost.

- After MACGetHeader() returns TRUE, use the routines MACGet() and MACGetArray() to read the data field. See AN833 for documentation. In the next chapter MACGet() and MACGetArray() will be used.

- MACDiscardRX() frees the currently used receive buffer, making it ready to receive more data. Normally this routine will automatically be called by Stack-Task() when needed.

- The data in the Ethernet header, and all other TCP/UDP/IP headers, are stored big endian (most significant byte first). The Microchip PICmicro and CCS C Compiler store data in a little endian format (least significant byte first). Therefore, any data in headers, such as the type field in the Ethernet header, needs to be converted from big endian format to little endian format.

❑ Copy the first nine lines of ex7a.c into a new file called "ccstcpip.h". This header file will be used in the next few examples.

❑ The following example shows how to use the MAC code to send Ethernet packets.

❑ Enter the following code into ccstcpip.h and save.

```
#define BUTTON1_PRESSED()    (!input(PIN_A4))
#define BUTTON2_PRESSED()    (!input(PIN_B1))

#define USER_LED1    PIN_A5
#define USER_LED2    PIN_B4
#define USER_LED2    PIN_B5
#define LED_ON       output_low
#define LED_OFF      output_high

void MACAddrInit(void) {
(continued...)
```

```
(continued...)
   MY_MAC_BYTE1=1;
   MY_MAC_BYTE2=2;
   MY_MAC_BYTE3=3;
   MY_MAC_BYTE4=4;
   MY_MAC_BYTE5=5;
   MY_MAC_BYTE6=6;
}

char ExampleIPDatagram[] = {
   0x45, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x00,
   0x64, 0x11, 0x2A, 0x9D, 0x0A, 0x0B, 0x0C, 0x0D,
   0x0A, 0x0B, 0x0C, 0x0E
};

char ExampleUDPPacket[] = {
   0x04, 0x00, 0x04, 0x01, 0x00, 0x04, 0x00, 0x00,
   0x01, 0x02, 0x03, 0x04
};
```

❑ Now create ex7b.c with the following code.

```
#include "ccstcpip.h"

void main(void) {
   MAC_ADDR mac_dest;

   set_tris_b(0);

   MACAddrInit();

   MACInit();

   mac_dest.v[0]=0xFF;
   mac_dest.v[1]=0xFF;
   mac_dest.v[2]=0xFF;
   mac_dest.v[3]=0xFF;
   mac_dest.v[4]=0xFF;
   mac_dest.v[5]=0xFF;

(continued...)
```

```
(continued...)

   while(TRUE) {
      if (MACIsTxReady()) {
         MACPutHeader(
            &mac_dest, ETHER_IP,
            sizeof(ExampleIPDatagram) + sizeof(ExampleUDPPacket)
         );
         MACPutArray(ExampleIPDatagram, sizeof(ExampleIPDatagram));
         MACPutArray(ExampleUDPPacket, sizeof(ExampleUDPPacket));
         MACFlush();

         output_toggle(USER_LED2);
         delay_ms(1000);
      }
   }
}
```

❑ Compile and run on the prototyping board.

❑ An Ethernet packet with 32 bytes of data is sent every second in this code. To prevent malformed packets from interrupting the network, the data being sent is actually a valid IP datagram holding a valid UDP packet. IP and UDP will be discussed in upcoming chapters. Examining network traffic in Wireshark returns the following results:
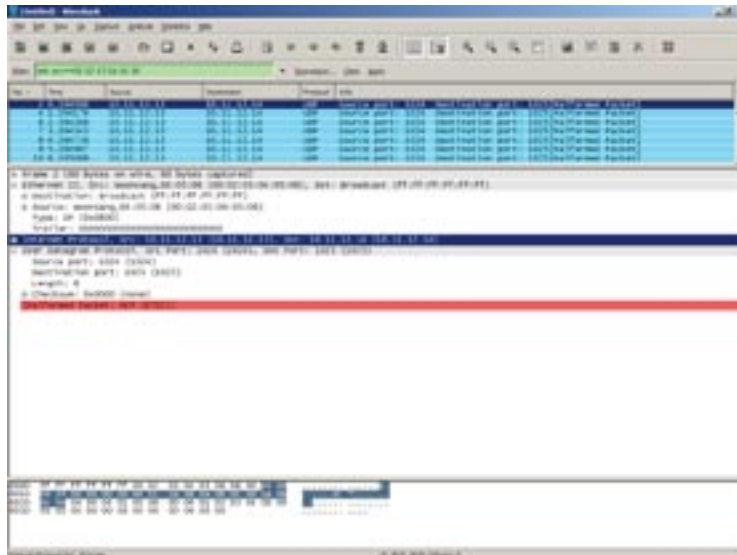
While examining the traffic on a network using Wireshark, there may be a lot more traffic that is not from this example. Because of this, filter out all other traffic that is not from MAC address 01:02:03:04:05:06. This is the MAC address CCS has assigned the NIC, so only these examples will be using it.
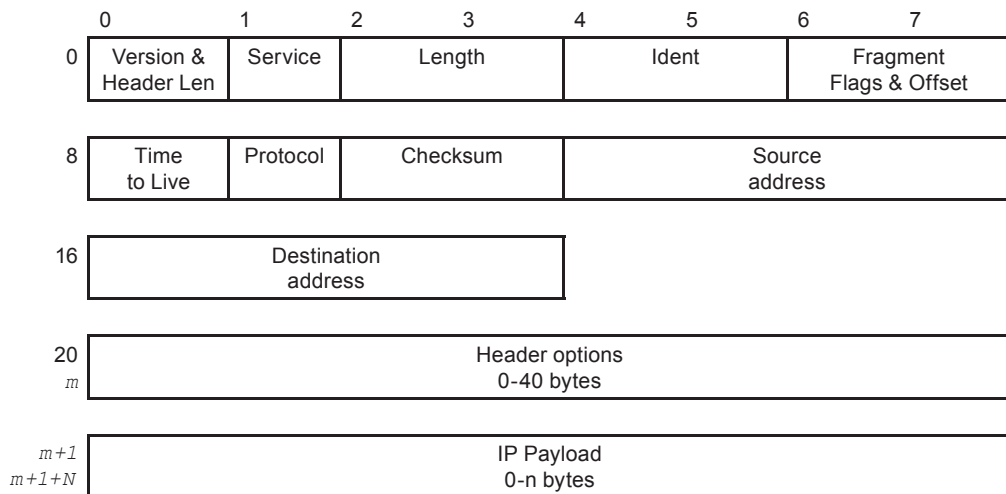
NOTES

- Since the destination MAC address sent was FF:FF:FF:FF:FF:FF, this packet is broadcast to the entire LAN.

- MY_MAC_BYTE1 to MY_MAC_BYTE6 are macros to the stacks method of assigning a unit's MAC address. They have to be set before MACInit() is called. Each unit should have a unique MAC address.

- MACPutHeader(*MAC, type, size) puts the 14 byte Ethernet header to the NIC's transmit buffer, where MAC is the destination address. The source address is always the unit's address. Size is the number of bytes that make up the data field of this Ethernet packet.

- MACPutArray(*array, size) puts the specified data into the NIC's transmit buffer. Call MACPutArray() once to stuff the IP datagram, and then call MACPutArray() again to stuff the UDP datagram. This is similar to what the TCP/IP stack will do when it has to stack all the layers of the protocol. Also, a simpler MACPut() is also provided to stuff just one byte into the transmit buffer.

- MACFlush() will mark the transmit buffer as ready for transmit. Once the data is marked ready for transmit, the NIC will continuously attempt to send the data using Ethernet's collision sense detection method.

- MACIsTxReady() will return TRUE if the transmit buffer is ready and free. It will return FALSE if it is not, probably because it is still handling the last MACFlush() command to send out the transmit buffer.

- Although the two examples in this chapter show how to send and receive Ethernet packets, the examples are purely academic. The TCP/IP stack will handle most of these low level functions.

8 ▸ IP LAYER

❑ The Internet Protocol, or IP, is the network layer that makes up most of the Internet. Its primary design feature was to allow for a packet-switched internetwork, where packets from several networks could be dynamically routed to each other to find the fastest route between two nodes.

❑ Note, this tutorial will be discussing the IPv4 protocol. IPv6 was introduced several years ago to increase the addressing space of the Internet to allow for more devices, and in several years it is planned for all Internet nodes to be running IPv6. However, IPv4 will still be supported for a long time.

❑ The IP packet format looks like this:

	0	1	2	3	4	5	6	7
0	Version & Header Len	Service	Length		Ident		Fragment Flags & Offset	
8	Time to Live	Protocol	Checksum		Source address			
16	Destination address							
20 / m	Header options 0-40 bytes							
$m+1$ / $m+1+N$	IP Payload 0-n bytes							

❑ Version & Header Len – The first byte in the IP header is actually two 4-bit fields. The most significant four bits is the version of IP being used, which will be IPv4. The least significant four bits are the length of the header in 32-bit words, not including the header options. The size of the header shown above is five 32-bit words.

❑ Service – Often used by IP routers as a priority setting. This field is ignored by the microchip TCP/IP stack.

❑ Length – The total length of the IP packet, including the header, header options and data.

❑ Ident and Fragment Flags & Offset – As packets route between several networks, different physical/link layers may have different Maximum Transmission Units (MTU). Therefore, routers may need to split IP packets into smaller IP packets to accommodate networks with smaller MTUs. These fields handle this features. The Microchip TCP/IP stack does not handle fragmentation.

❑ Time to Live – The time for this packet to live in seconds. As a packet is routed through networks, routers will decrease this value until it reaches zero. Once it reaches zero the router will discard the packet, and it is considered lost.

❑ Protocol – The protocol that is being used in the IP data field. Common protocols are TCP (0x06), UDP (0x11) and ICMP (0x01).

❑ Checksum – A checksum of the IP header ensures the packet is not corrupted. Note that this checksum does not include the payload.

❑ Options – Several options can be added to the IP, such as extra information about routing and security. Options will not be used in this tutorial or in the programs.

❑ Source & Destination Address – An address to identify an individual computer on the Internet. Each IP address must be unique. IPv4, the version of IP that we will be using, uses four bytes of data to represent an IP address. IPv6 has increased the address space to 16 bytes.

❑ There are a few IP addresses that have special meaning:
  ● Broadcast – An IP address of 255.255.255.255 is used to broadcast to all nodes on the LAN (local area network). Determining what is the LAN and what is the wide area network (WAN) will be described shortly.
  ● Loopback – Any address starting with 127 is a loopback address, and most TCP/IP stacks will loop all transmitted packets right back into the receiver. Usually the loopback address in TCP/IP stacks is 127.0.0.1, and the rest of the 127.*.*.* address space is unused.
  ● Private Networks - The following address ranges have been reserved for private use and are not explicitly connected to the Internet:
    ● 10.0.0.0 to 10.255.255.255
    ● 172.16.0.0. to 172.31.255.255
    ● 192.168.0.0 to 192.168.255.255
    Many LAN networks will use these addresses, but then use a gateway (router or firewall) to connect the LAN to the Internet.

❑ IP was designed to create a routed, packet-switched internetwork, composed of several different networks. The IP address is the primary component used to determine how to route between several different networks. For example, a computer in the LAN with an address of 192.168.100.15, talks to Google.com with an IP address of 64.233.187.104. Since 192.168.*.* range was reserved for the LAN and not connected to the Internet a gateway is used to bridge the LAN with the WAN (internet) .

❏ A network specific configuration called the "subnet mask" is used to determine the difference between the LAN and the WAN. The source and destination address are logically AND with the subnet mask, and if the two values are the same they are considered on the same network. If the two values are different then the two nodes are considered on a different network, so the communication must proceed through a gateway (router or firewall).

❏ In the case using the above for Google.com, if a subnet mask of 255.255.0.0 is used:
   (255.255.0.0) & (64.233.187.104) = 64.233.0.0
   (255.255.0.0) & (192.168.100.15) = 192.168.0.0
   (64.233.0.0) != (192.168.0.0)
therefore these two nodes are on a different network, use a gateway.

❏ Another network specific configuration is the gateway address. A gateway connects two networks, and is often used to connect a LAN to a WAN. For simplicity a gateway can also be considered a router, although they are not exactly the same. Many hardware firewalls sold today also combine a router and firewall. In the above example, since Google.com is not on the LAN, any physical/link layer packets cannot be directly sent to Google.com. Instead the IP packet is sent to the gateway. The IP header will still contain the IP address of Google.com, but in the physical/link layer the MAC address will be the gateway's address. The gateway/router/firewall will then know how to send the IP packet to the next router in the network.

❏ IP sits on the network layer, above the physical/link layer. In order for two IP nodes to communicate with each other, they need a method to learn the physical/link address of each other. The protocol used in Ethernet to learn the physical/link address is ARP, and is covered in Chapter 10.

❏ The next example will act as an IP packet sniffer and display header information of incoming IP packets.

❏ Enter the following code into ex8a.c.

```
#include "ccstcpip.h"

void IPDisplayHeader(void) {
    IP_ADDR dest_ip;
    NODE_INFO node;
    int8 prot;
    int16 len;

(continued...)
```

```
(continued...)

    if (IPGetHeader(&dest_ip, &node, &prot, &len)) {
        printf("\r\n  DEST: %U.%U.%U.%U  SRC: %U.%U.%U.%U  LEN: %LU  PROT: %X ",
            dest_ip.v[0], dest_ip.v[1], dest_ip.v[2], dest_ip.v[3],
            node.IPAddr.v[0], node.IPAddr.v[1], node.IPAddr.v[2], node.IPAddr.v[3],
        len, prot);
        if (prot==IP_PROT_ICMP) {printf("[ICMP]");}
        else if (prot==IP_PROT_TCP) {printf("[TCP]");}
        else if (prot==IP_PROT_UDP) {printf("[UDP]");}
    }
    else {
        printf("\r\n  [MALFORMED IP]");
        }
}

void MACDisplayHeader(MAC_ADDR *mac, int8 type) {
    int8 i;
    printf("\r\nMAC: ");
    for (i=0;i<6;i++) {
        printf("%X", mac->v[i]);
        if (i!=5)
            putc(':');
    }
    printf("  PROT:0x08%X ",type);
    if (type==MAC_IP)
{
        printf("[IP]");
        IPDisplayHeader();
    }
    else if (type==MAC_ARP)
        printf("[ARP]");
}

void main(void) {
    MAC_ADDR mac;
    int8 type;

    printf("\r\n\nCCS TCP/IP TUTORIAL, EXAMPLE 0A\r\n");

    MACAddrInit();

    MACInit();

    while(TRUE) {
        if (MACGetHeader(&mac, &type)) {
            if (type!=MAC_UNKNOWN) {
                MACDisplayHeader(&mac, type);
            }
        }
    }
}
```

❑ Compile and run on the prototyping board

❑ Examine the serial output, and notice the messages will appear as follows:

```
MAC: 00:C0:EE:D6:0F:99      PROT:0x0806 [ARP]
MAC: 00:03:6D:1D:53:9A      PROT:0x0806 [ARP]
MAC: 00:A0:CC:63:E5:AA      PROT:0x0800 [IP]
    DEST: 192.168.100.255      SRC: 192.168.100.166  LEN: 198  PROT: 11 [UDP]
MAC: 00:0A:E6:60:68:11      PROT:0x0800 [IP]
    DEST: 192.168.100.255      SRC: 192.168.100.106  LEN: 58   PROT: 11 [UDP]
MAC: 00:03:6D:1D:53:9A      PROT:0x0806 [ARP]
MAC: 02:A0:CC:65:8F:C9      PROT:0x0800 [IP]
    DEST: 192.168.100.255      SRC: 192.168.100.209  LEN: 215  PROT: 11 [UDP]
MAC: 00:03:6D:1D:53:9A      PROT:0x0806 [ARP]
```

❑ MAC is the MAC address of the sender, PROT is the protocol field. If an IP protocol was detected, it will display the destination and source IP address of this packet, as well as the length of the IP datagram and the IP protocol being used.

NOTES

- Notice that this is the same code as ex7a.c, but IPDisplayHeader() was added to display IP header information if the received Ethernet packet contains an IP packet.

- IPGetHeader(*DEST_IP, *NODE, *PROTOCOL, &LEN) will return TRUE if that Ethernet packet contains an IP header that is compatible with the IP stack, and if the IP packet destination address matches the IP address (or if its a broadcast IP address). Only call IPGetHeader() if MACGetHeader() was successfully called and the Ethernet protocol field specifies IP.

- The NODE field contains the MAC address and IP address of the node on the network that transmitted the network packet. Keep both of these values in case a reply needs to be sent.

- NODE_INFO and IP_ADDR are structures defined in the TCP/IP stack.

- To read the contents of the data in the IP packet after a succesful IPGetHeader(), use the function IPGetArray(). See AN833 for more documentation.

❑ To inspect the process of sending IP packets, add the following code into ccstcpip.h:

   ***Note: alter the code in IPAddrInit() to use the Subnet Mask and
   Network Gateway address of your network. Also, set the IP address
   of the unit to a free IP address in your network. If these values are
   unknown, consult your network administrator or use the ipconfig tool
   included in Microsoft Windows (ifconfig in Linux).**

```
void IPAddrInit(void) {
    //IP address of this unit
    MY _ IP _ BYTE1=192;
    MY _ IP _ BYTE2=168;
    MY _ IP _ BYTE3=100;
    MY _ IP _ BYTE4=7;
    //network gateway
    MY _ GATE _ BYTE1=192;
    MY _ GATE _ BYTE2=168;
    MY _ GATE _ BYTE3=100;
    MY _ GATE _ BYTE4=1;
    //subnet mask
    MY _ MASK _ BYTE1=255;
    MY _ MASK _ BYTE2=255;
    MY _ MASK _ BYTE3=255;
    MY _ MASK _ BYTE4=0;
}
```

❑ Enter the following code into ex8b.c.

```
void main(void) {
    NODE _ INFO node;

    set _ tris _ b(0);

    IPAddrInit();
    MACAddrInit();

    MACInit();

    memset(&node.MACAddr.v[0], 0xFF, sizeof(MAC _ ADDR));

    node.IPAddr.v[0]=192;
    node.IPAddr.v[1]=168;
    node.IPAddr.v[2]=100;
    node.IPAddr.v[3]=8;
    while(TRUE) {
        if (IPIsTxReady()) {
            IPPutHeader(&node, IP _ PROT _ UDP, sizeof(ExampleUDPPacket));
            IPPutArray(ExampleUDPPacket, sizeof(ExampleUDPPacket));
            MACFlush();
            output _ toggle(USER _ LED1);
            delay _ ms(1000);
        }
    }
}
```
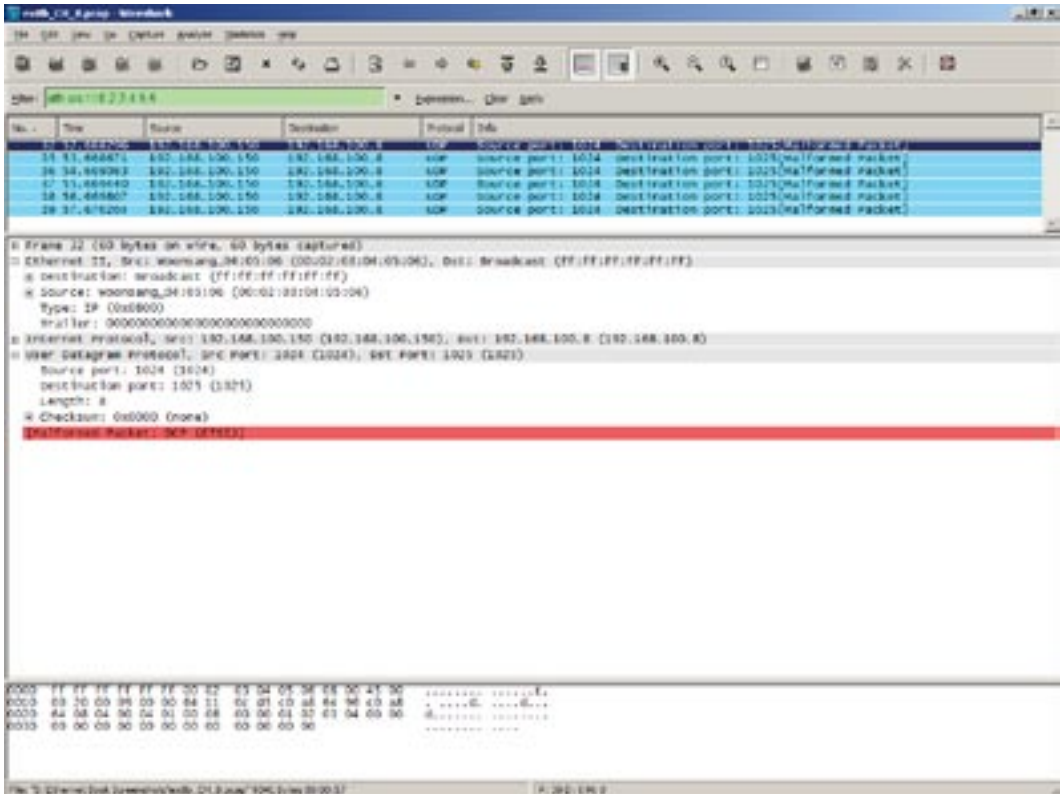
❑ Compile and run on prototyping board.

❑ This example is similar to the second example in Chapter 7: every second an IP datagram is sent that contains a UDP datagram. In fact, the same packet is sent in this example that was sent in Chapter 7. The only difference in this new example is that the stack's IP routines were used to generate the IP header. While this example is running, use Wireshark to view the packets:
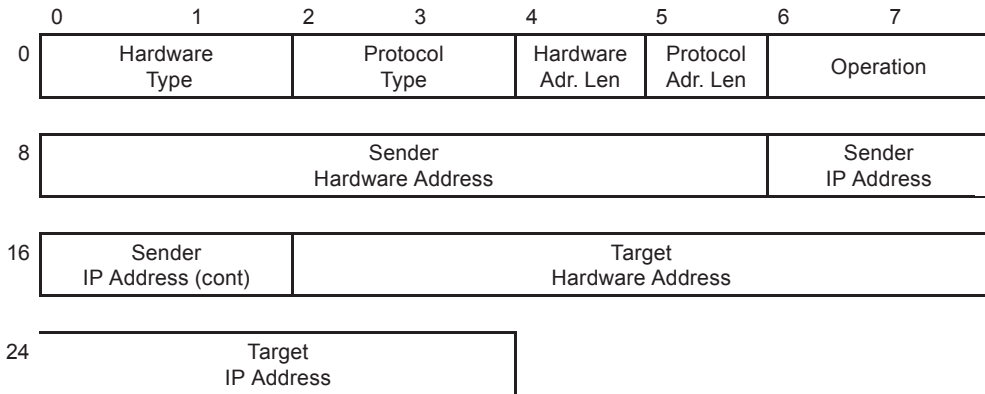
- NODE_INFO is a structure that contains the MAC Address and the IP Address of the destination node. Both addresses are needed.

- This example sets the unit's IP address to 192.168.100.7 and the destination IP address to 192.168.100.8. Make sure that the 18 addresses used are not taken by any other device on your network to prevent confusion within a network.

- This example sets the destination MAC address to FF:FF:FF:FF:FF:FF, which is the broadcast address, so each node on the Ethernet network will receive this packet. It is not efficient to use the broadcast address; later ARP will be used to find the MAC address of the specified IP address.

- IPIsTxReady() verifies that the IP stack is ready to transmit a new IP datagram and also verifies that the Ethernet transmit buffer is open and free.

- IPPutHeader(*NODE_INFO, protocol, size) creates a valid IP datagram header for the specified destination node and puts that header into the transmit buffer.

- IPPutArray(*array, size) puts the specified array into the transmit buffer. Do not call this until after IPPutHeader() has put the IP header into the transmit buffer.

- MACFlush() marks the Ethernet transmit buffer as ready for transmission, just as in the previous example.

❑ While the two previous examples show how to send and receive IP packets, these examples are purely academic. If using TCP/IP or UDP/IP, the Microchip stack will handle all this.

9 ADDRESS RESOLUTION PROTOCOL (ARP)

❑ The IP protocol is in the network layer, so it requires a physical and link layer to act as the medium. In order for two devices on the link layer to communicate the two devices must know their link layer address, which is different from the IP address. In Ethernet the link layer address is the MAC address. Therefore, if two devices using IP want to communicate with each other they must have a way of determining the link layer address between the two nodes. On Ethernet, the method of determining the MAC address of a specified IP address is Address Resolution Protocol (or ARP).

❑ The ARP packet format is as follows:

0	1	2	3	4	5	6	7
0 Hardware Type		Protocol Type		Hardware Adr. Len	Protocol Adr. Len	Operation	
8 Sender Hardware Address						Sender IP Address	
16 Sender IP Address (cont)		Target Hardware Address					
24 Target IP Address							

❑ Hardware Type – The Link Layer medium being used. Ethernet (0x0001) will be used.

❑ Protocol Type – The Network Layer protocol being used. IP (0x0800) will be used.

❑ Hardware Address Length – The size, in bytes, of the Link Layer address. A MAC address is 6 bytes long.

❑ Protocol Address Length - The size, in bytes, of the Network Layer addresses. An IP address is 4 bytes long.

❑ Operation – The opcode. We will handle request (0x0001) and reply (0x0002).

❑ Sender Address – These specify the link layer (Ethernet) and network layer (IP) addresses of the unit making the request/sending reply.

❑ Target Address – These specify the link layer (Ethernet) and network layer (IP) of the unit receiving the request/receiving the reply. If a request is being made for an Ethernet address, the Target Hardware Address will be zeroed out.

❑ The first example will look at answering ARP requests using the ARP functions included in the Microchip TCP/IP stack.

☐ Enter the following code into ex9a.c

```c
#define STACK_USE_ARP 1
#include "ccstcpip.h"

enum {ARP_ST_IDLE=0, ARP_ST_REPLY=1} my_arp_state=0;
NODE_INFO arp_req_src;

void my_arp_task(void) {
    switch (my_arp_state) {
        case ARP_ST_REPLY:
            if ( ARPIsTxReady() ) {
                ARPPut(&arp_req_src, ARP_REPLY);
                my_arp_state=ARP_ST_IDLE;
            }
            break;

        default:
            break;
    }
}

void main(void) {
    NODE_INFO src;
    int8 opCode, type;

    set_tris_b(0);
    printf("\r\n\nCCS TCP/IP TUTORIAL, EXAMPLE 9A (ARP RECEIVE)\r\n");

    MACAddrInit();
    IPAddrInit();

    MACInit();

    while(TRUE) {
        if (MACGetHeader(&src.MACAddr, &type)) {
            if (type==MAC_ARP) {
                if (ARPGet(&arp_req_src, &opCode)) {
                    output_toggle(USER_LED1);
                    my_arp_state=ARP_ST_REPLY;
                }
            }
        }
        my_arp_task();
    }
}
```

☐ Compile and run on the prototyping board.

❑ Once running on the prototyping board, this firmware will answer any ARP requests asking for a MAC address. Unfortunately, there is no simple way of making an ARP request on a Windows PC, but there are some tools available. The ARP tool can be used to display a list of all known Ethernet addresses and their accompanying IP address. Running "arp -a" will display this table. Run "arp -a" in a command line prompt and verify that the IP address of the unit, which is set in the IPAddrInit() function, is not in the current ARP table. If the IP address of the unit is in the ARP table, use "arp -d" to clear the ARP table.

❑ Make an ARP request by issuing a ping to the unit. Ping is a tool that is used to debug IP connections to verify that an IP address is reachable. Ping will be covered in the next chapter. For now use a ping to initiate an ARP request. From the command line, run "ping 192.168.100.7" where 192.168.100.7 is the IP address specified in the IPAddrInit() function. While the unit will not respond to the ping, it will respond to the ARP request. Running "arp -a" should now show 192.168.100.7 in the ARP table at 01:02:03:04:05:06. If running Wireshark to packet sniff this transaction, the following will appear:
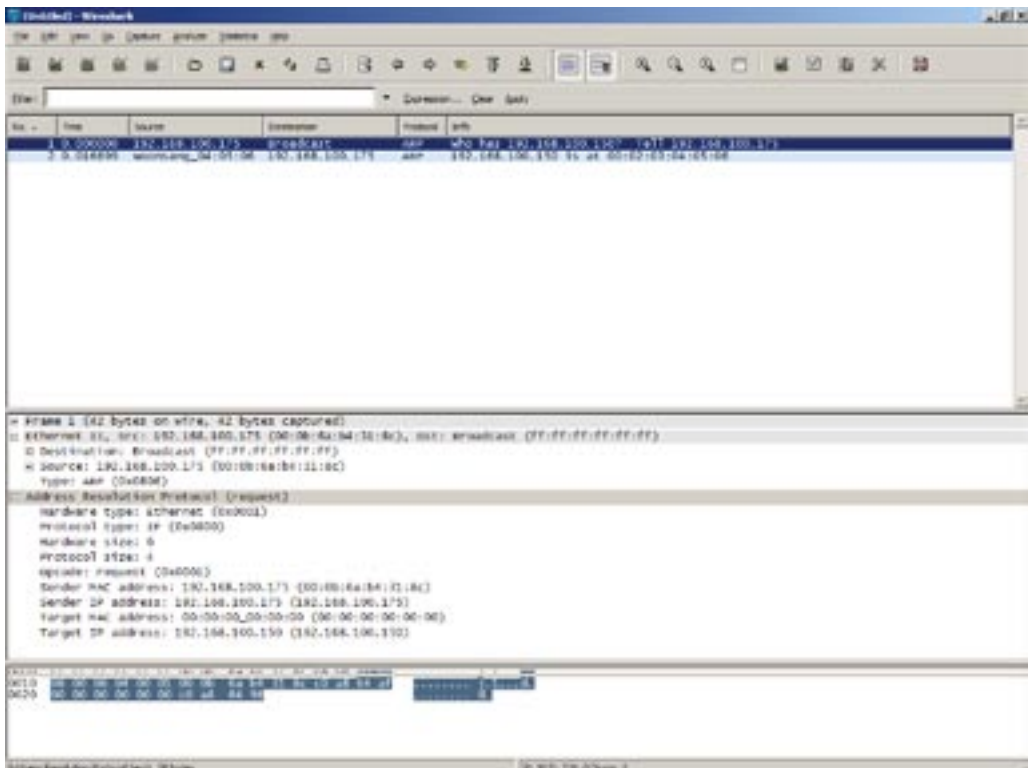
- Define STACK_USE_ARP as TRUE early in the code to include the ARP functions. Not all link layers need ARP (for example point-to-point protocols such as SLIP and PPP do not need address resolution since there is no link layer addressing), so it does not always need to be included. ARP is required for Ethernet.

- ARPGet(*NODE_INFO, *opCode) returns TRUE if it received a request for it's IP address, and the opCode was a request. Over the LAN there will be many ARP requests being made, but the unit should only respond to requests asking for it's address.

- ARPIsTxReady() returns TRUE if the Ethernet transmit buffer is empty and ready, and if the ARP handler is available for sending requests/responses. Do not attempt to send ARP requests/responses unless this returns TRUE.

- To prevent sitting in a loop waiting for ARPIsTxReady() to return TRUE, MY_ARP_TASK() use a cooperative method to free up CPU time for other tasks as detailed in Chapter 5 of this tutorial.

- ARPPut(*NODE_INFO, opCode) sends an ARP response/request to the specified remote node.

- It is possible to replace ARPGet() and ARPPut() with MACGet() and MACPut(), but then you would have to manually parse an ARP packet.

❑ ARP can be used to sniff the entire network to determine the IP<->MAC address of each node in the network. The following example ex9b, will do just that, and has been provided on the Development Tools CD.
Open ex9b.c on the CD, compile and run on the prototyping board. When viewing the output on a serial terminal, notice which IP addresses respond to ARP and what that IP address is:

```
192.168.100.1 <-> 00:0D:88:B0:12:6F
192.168.100.2 <-> NO RESPONSE
192.168.100.3 <-> 61:3B:F7:7A:11:55
192.168.100.4 <-> NO RESPONSE
192.168.100.5 <-> 00:09:5B:E1:30:E2
192.168.100.6 <-> 00:0A:E6:61:F4:1D
```

NOTES

● This example assumes that the network is 192.168.100.*. If this is not the case, edit IPAddrInit() in ccstcpip.h.
● This example calls the same ARP functions as the previous example, but instead of sending a response we send a request.
● Most TCP/IP stacks hold a cache of ARP results. For example, using the ARP tool in Microsoft Windows shows the current cache of ARP results. The Microchip TCP/IP stack only caches one result.

❑ These two examples were purely academic. The Microchip TCP/IP stack will handle all ARP requests and responses.

❑ The Internet Control Message Protocol, or ICMP, is used primarily for diagnostics. The most useful tool in ICMP is "ping", which is used to determine if a specific IP address is reachable. ICMP messages are carried in an IP payload with the IP datagram protocol field set to 1. The ICMP header and packet follows this format:

0	1	2	3	4	5	6	7
0	Type	Code	Checksum		Payload		

8	Payload (cont)			Payload (cont) [optional]			
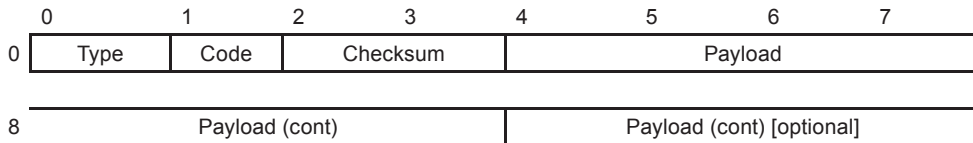
❑ Type – The ICMP request type. Since the primary goal will be to respond to ping requests, the two codes are: 0x00-Echo Reply and 0x08-Echo Request. There are many more requests.

❑ Code – If a destination is unreachable, this field denotes at which level (host, protocol, port, etc).

❑ Checksum – A checksum of the ICMP header and data. The IP header checksum algorithm is used.

❑ Payload – A minimum of eight bytes. When making an echo request/reply, the first two bytes are the identifier and the next two bytes are the sequence number.

❑ When an ICMP packet with an echo request code is received the receiving node just has to echo it back with an echo response code. The following example will do just that and is included on the Development Tools CD.

❑ Open ex10a.c on the CD, compile and run on the prototyping board. Once the firmware is running on the unit, use the command line tool "ping" to ping the unit, and it will respond successfully:

```
C:\ping 192.168.100.7

Pinging 192.168.100.7 with 32 bytes of data:

Reply from 192.168.100.7: bytes=32 time=3ms TTL=100
Reply from 192.168.100.7: bytes=32 time=3ms TTL=100
Reply from 192.168.100.7: bytes=32 time=3ms TTL=100
Reply from 192.168.100.7: bytes=32 time=3ms TTL=100

Ping statistics for 192.168.100.7:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 3ms, Maximum =  3ms, Average =  3ms
```

❑ The following is a screen shot of Wireshark viewing a ping transaction:
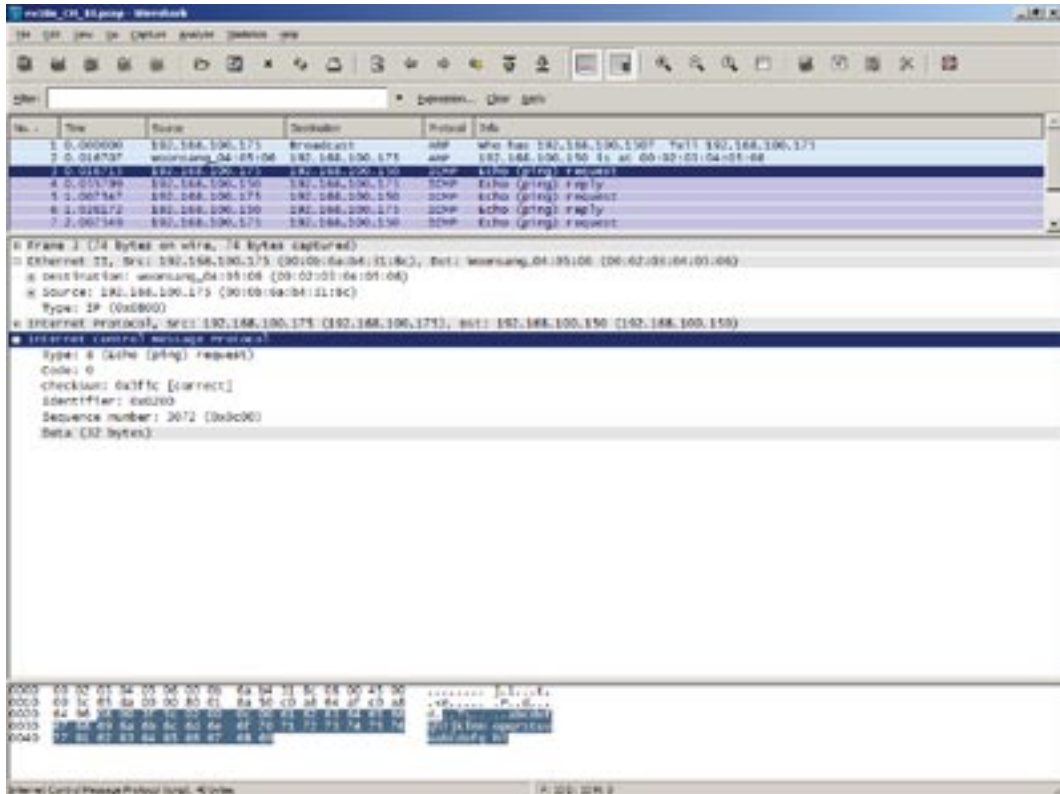
This example is purely academic, the TCP/IP stack provided by Microchip will automatically handle ICMP requests. To demonstrate how the TCP/IP stack provided by Microchip handles all the low level functions, enter the following code as ex10b.c, compile and run on the prototyping board:

```
#define STACK _ USE _ ICMP  1
#define STACK _ USE _ ARP   1
#include "ccstcpip.h"

void main(void) {
    MACAddrInit();
    IPAddrInit();

    StackInit();

    while(TRUE) {
        StackTask();
    }
}
```

NOTES

- StackInit() initializes the Microchip TCP/IP stack. This will initialize the Ethernet controller, the IP, ARP, UDP and TCP state machines, etc.
- It is still necessary to initialize MAC address and IP address, which is done by the custom functions MACAddrInit() and IPAddrInit(). The next chapter will use DHCP to automatically find an IP address and network information.
- StackTask() checks the Ethernet receive buffer for incoming packets, and if there are packets in the buffer, it will pass the packet to the higher layers, such as ARP, IP, TCP, UDP, etc. For example, if an ARP request is received StackTask() it checks to see if it is a request for the user, and if it is, it will parse and send a response.
- StackTask() also keeps an eye on the Ethernet transmit buffer, as certain tasks must wait for the transmit buffer to be free. For example, the ARP task may take a few StackTask() calls until the transmit buffer is free, at which time the response is sent.
- From this point onward StackTask() will be used to handle all the lower level functions.

11 DYNAMIC HOST CONFIGURATION PROTOCOL (DHCP)

❑ Dynamic Host Configuration Protocol, or DHCP, allows a node on an IP network to automatically allocate an IP address to the unit and learn the network parameters such as gateway and netmask. DHCP is found on many networks, and is great for allowing notebooks (or other hosts) to simply plug into a network without having a user enter network configuration parameters.

❑ This chapter will use DHCP to configure the Microchip TCP/IP stack automatically. (If DHCP is not available on your network, skip to the next chapter. This example will write to the LCD on the prototyping board, so it may still be interesting to users without DHCP.)

❑ Implementing DHCP in an application is easy using the Microchip TCP/IP stack, as DHCP support is provided. An example is provided on the Development Tools CD, called ex11.c. Open ex11.c, compile it and run on the prototyping board.

❑ The second line of the LCD will show one of three messages:

No Ethernet	No Ethernet cable is connected to the Ethernet connector.
DCHP Not Bound	DHCP could not reach a DHCP server, or DHCP server has not finished configuring unit.
x.x.x.x	DHCP server has responded by assigning htis IP address to your unit.

❑ Once an IP address is configured, verify by using the ping tool.

NOTES

- StackTask() handles all the Ethernet, IP, ARP, DHCP and ICMP messages necessary for this example. Since StackTask() is doing all the work, the majority of this code is refreshing the LCD every second with the current DHCP status.
- Defining STACK_USE_DHCP to TRUE will include all the DHCP code into the TCP/IP stack. Once it is included StackInit() and StackTask() will automatically handle DHCP requests.
- MACIsLinked() returns TRUE if an Ethernet cable is connected to the Ethernet controller.
- DHCPIsBound() returns TRUE if the unit has been successfully configured by the DHCP server. If using DHCP, do not attempt to use the network until DHCPIsBound() returns TRUE.
- Once DHCP is bound, the network mask and the network gateway will also be configured.
- To dynamically disable DHCP in the code, call DHCPDisable() before the first StackTask().

❑ The rest of this tutorial assumes there is no DHCP, and will statically configure the IP network settings as done in previous chapters.

❑ UDP is a simple, lightweight transaction layer. Combined with IP, commonly called UDP/IP, the user may quickly design simple communications between two devices on an internetwork. As a transaction layer, UDP is a lightweight alternative to TCP; UDP is easier to implement and requires fewer resources on the microcontroller.

❑ UDP does have some drawbacks. First, there is no guarantee that packets will be received in the order they were sent. Second, there is no method to determine if a packet was successfully received. TCP does not have these drawbacks, and TCP will automatically resend a packet if it was not successfully received. For some applications, such as audio and video streaming, these drawbacks are not a concern. If these drawbacks are a problem in an application, implement a sequencing and ACK messaging format.

❑ The format of a UDP datagram is as follows:

0	1	2	3	4	5	6	7
Source Port		Destination Port		Message Length		Checksum	

8							
Payload							
n							
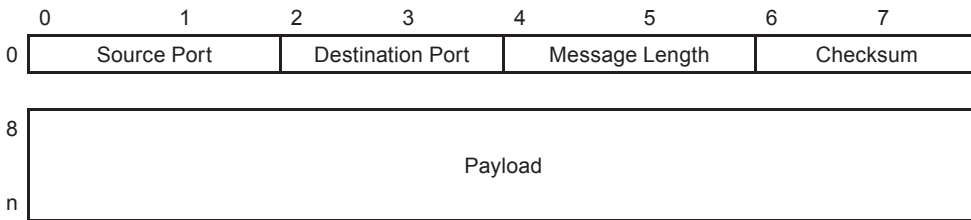
❑ Source/Destination Port – Application multiplexing is provided through the source and destination port. An application can listen to a specific port, and any packet received that matches that specific destination port is considered destined for that application. The application could then respond by sending a response to the port specified in the source port.

❑ Message Length – Length of UDP header and data.

❑ Checksum – The checksum of the UDP header and data.

❑ Payload – Data of the UDP datagram. It can be 0 bytes if the request contains no data. There is no limit to the size of the data field as IP will fragment messages as needed by network constraints.

❑ To demonstrate UDP, CCS has provided an example that runs on the development kt and talks to a simple PC application. First, compile and run EX12.C included on the Development Tools CD. Once running, note the following on the LCD screen:

```
CCS UDP TUTORIAL
LISTENING
```

❑ The prototyping board is now listening for incoming UDP packets on port 1024. Execute UDP.EXE that is also on the CD. UDP.EXE is a simple application that will send/receive UDP packets to a specified IP address:

❑ Once both UDP.EXE and EX12.C is running, press the "Open port" button in UDP.EXE. (Note: The IP address of the development kit should be entered into the text box labeled "Dest IP"). Once the port is open, insert characters into the "Outgoing Data" text box and press Send. If successful, the text will show up on the LCD screen. The LCD will also show the IP address of the last unit that sent a UDP packet.

❑ Once the prototyping board has received an IP address, press the button labeled A4 on the development board. Pressing this button will send a plain-text "BUTTON" over UPD. This text will show up in the "Incoming Data" textbox in UDP.EXE..
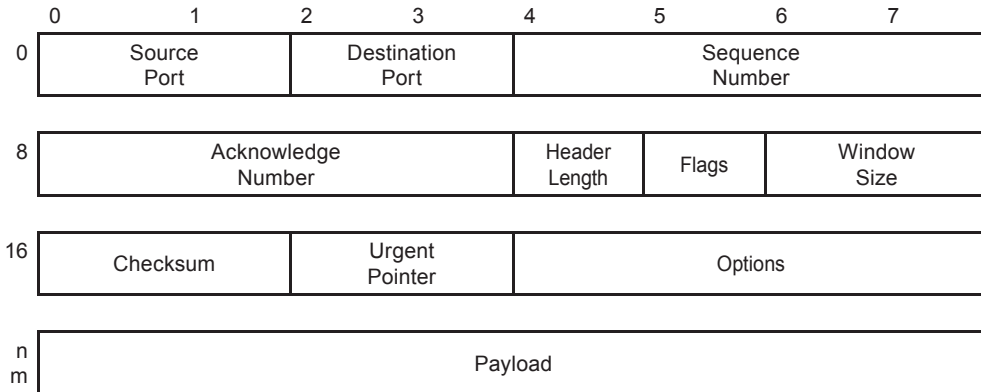
- Before any UDP communication can take place, UDPOpen() must be called. Calling UDPOpen() creates an active socket between the microcontroller and the remote node. This socket identifier is used to distinguish between several active communications. With cooperative multitasking, several of these communication sockets can be active at once.

- UDPOpen(localPort, *remoteNode, remotePort) will create a communication socket between the microcontroller (using localPort as its socket) and the remote node (where remoteNode is a NODE_INFO struct holding the IP and MAC address, and remotePort is the UDP port of the remote node)

- UDPOpen(localPort, NULL, INVALID_UDP_PORT) will create a communication socket that listens for incoming packets from any remote node.

- If UDPOpen() returns INVALID_UDP_PORT that means there are no more communication sockets. Define the maximum number of UDP and TCP sockets by the constants MAX_UDP_SOCKETS and MAX_SOCKETS respectively. One socket takes more than 35 bytes of RAM, so only enable as many sockets as needed to save RAM.

- UDPIsPutReady(socket) checks to make sure the Ethernet transmit buffer is free, and that the socket is still connected. Make sure this returns TRUE before an attempt to transmit any data. This also sets this socket as the active socket for any future UDPPut() calls.

- UDPPut(c) puts the specified byte into the UDP datagram's data field. It will return TRUE if successfully.

- UDPFlush() marks the UDP datagram as ready to transmit. Once the Ethernet transmit buffer is free and ready, the Microchip TCP/IP stack will then create the Ethernet and IP headers automatically, and then send the UDP datagram.

- UDPIsGetReady(socket) checks the Ethernet receive buffer, and returns TRUE if the data in the Ethernet receive buffer is destined for this socket. It also marks this socket as the active socket for any future UDPGet() calls. IMPORTANT: if data is in the receive buffer it must be received and processed, because the next time StackTask() is called, the data in the receive buffer will be thrown away.

- UDPGet(&c) returns the next byte of data in the UDP datagram's data field. If it successfully received a byte, it saves to pointer and returns TRUE. If it returns FALSE there is no more data left in the datagram.

- UDPDiscard() throws away the rest of the data in the receive buffer.

- UDPClose(socket) frees the specified socket. Once freed, use UDPOpen() to create a new connection.

- The source code to UDP.EXE is also available on the Development Tools CD.

13 TRANSMISSION CONTROL PROTOCOL (TCP)

❑ The previous chapter reviewed the UDP transport layer, and was noted that UDP's simple design does not include guaranteed packet delivery. TCP is a more complex transport layer that adds: a persistent bi-directional connection, datagram duplication handling, datagram out-of-order handling, datagram loss handling, and flow control.

❑ The TCP datagram appears as follows:

0	1	2	3	4	5	6	7
0	Source Port		Destination Port		Sequence Number		
8	Acknowledge Number				Header Length	Flags	Window Size
16	Checksum		Urgent Pointer		Options		
n m	Payload						

❑ Source Port and Destination Port – To allow multiplexing TCP over several different applications, source and destination port combined with remote node's IP address can give many concurrent, persistent and individual connections. Several ports are well known, for example port 80 is used for HTTP. Web servers will listen to port 80 for incoming TCP connections.

❑ Sequence Number and Acknowledge number – These provide a method of out-of-order datagram and datagram loss handling. For example, if a node sends a TCP datagram with 20 data bytes with a sequence number of 100, the receiving node will send an acknowledge of 120 to signify to the transmitting node that it received the 20 bytes of data. The receiving node now knows that the next packet should have a sequence number of 120, otherwise it will be out-of-order. If a node received two packets with the same sequence number and same data size, it would know that one is a duplicate. If a node received a packet with a sequence number that was out-of-order, it must save the packet until the missing packet arrives. The node will then place the packets back into order. Also see Window Size.

❑ Header Length – Total size of the TCP datagram header, including options, divided by 4.

❑ Flags – Option flags. Here are a few of the more important ones:
- Bit 4 – ACK – Acknowledge the last received packet
- Bit 2 – RESET – Force a closure of the connection
- Bit 1 – SYN – Synchronization (create connection)
- Bit 0 – FIN – Ask for a closure of the connection

❑ Window Size - Specifies how much buffer space the receiving node has for reconstructing packets if they arrive out-of-order. The Microchip TCP/IP stack will set the window size to one segment as Microchip microcontrollers do not have enough RAM to reconstruct out-of-order packets.

❑ Checksum – Checksum of the header and data.

❑ Urgent Pointer – Can be used to signify urgent data. This is not used by the Microchip TCP/IP stack.

❑ TCP is a connection orientated transport layer. Once a connection is made, data can be sent bi-directionally until one of the units initiates a disconnect and both units close the connection. The methods to listen, open and close connections can best be described with the TCP/IP state diagram:
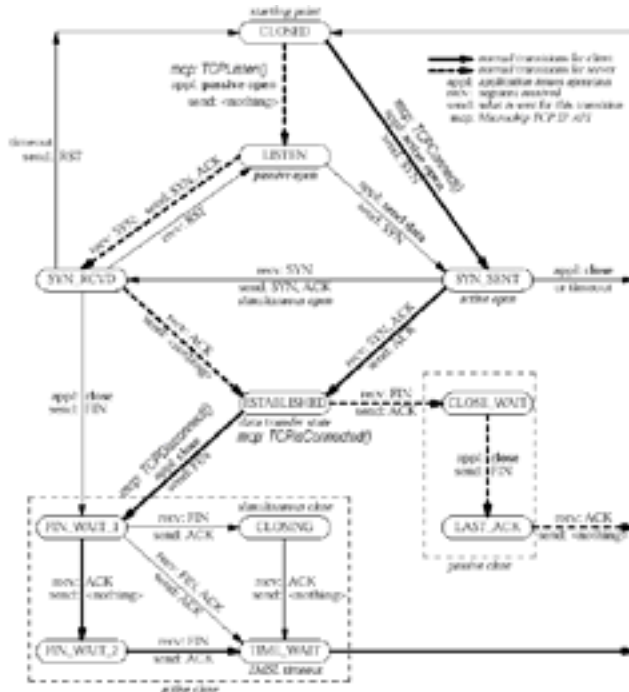
❑ Imposed on the previous state diagram are the routines provided by the Microchip TCP/IP stack to open connections and close connections. Once a connection is established, data can be sent and received bi-directionally.

❏ This chapter will use the Microchip TCP/IP routines to create an example TCP/IP client that will talk to a TCP/IP server on a PC. Open EX13.C on the Development Tools CD. Change the IP address in ServerAddrInit() to the IP address of the computer. Then compile and run EX13.C on the prototyping board. Once it is running, the following will appear on the LCD screen:

```
CCS TCP TUTORIAL
CONNECTING
```

❏ This example is now attempting to connect to the IP address set in ServerAddrInit(). On the CD, run TCPServer.EXE. Note the following on the PC:

❏ Press the "Listen" button. The PC will now listen to the specified socket, which is what the prototyping board is trying to connect to. Now that the PC is listening, a connection should eventually be made. Notice a successful connection in the "Incoming Data" text box in TCPServer.EXE. Also, the LCD it should now say "Connected!" on the bottom line. Now that it is connected, enter text into the "Outgoing Data" window of TCPServer.EXE and press send – that text should now be shown on the second line of the LCD.

❏ When connected, pressing the left button on the prototyping board should send a "BUTTON C=1" string to the PC, which will be displayed in the "Incoming Data" text box of TCPServer.exe. The number will increment with each button press.

- TCPConnect(*NODE_INFO, port) will attempt a connection to the remote node (specify both an IP address and a MAC address) using the specified port. After issuing a TCPConnect(), the Microchip TCP/IP stack will start issuing SYN packets to the specified remote node and handle responses to the SYN request. TCPConnect() will return a socket number used to signify this connection. The firmware must remember this socket number if it wishes to use this connection.

- TCPIsConnected(socket) returns TRUE when the TCP/IP state has reached Established state. Due to the cooperative method that the Microchip TCP/IP stack is implemented, do not sit in an infinite loop waiting for TCPIsConnected() to return true. If it returns FALSE, wait until the next task time to try again. (A task time in the Microchip TCP/IP stack is the next time after calling StackTask()).

- TCPIsPutReady(socket) returns TRUE if the TCP/IP socket is connected, the Ethernet transmit buffer is free and the TCP/IP state is ready to handle transmitting a packet. It also prepares the TCP/IP stack for the next TCP datagram by initializing the TCP header and writing the Ethernet and IP header to the Ethernet transmit buffer.

- TCPPut(socket, c) puts a character into the TCP datagram data field, and returns TRUE if successful.

- TCPPutArray(socket, int8 *ptr, int16 size) will put the specified characters into the TCP datagram field and return TRUE if successful. (Note: This function was not written by Microchip and so it is not in their API documentation)

- TCPFlush(socket) marks the TCP datagram (and accompanying Ethernet and IP header) as ready for transmission. It may take several task times for a TCP datagram to be sent, especially if there is a retry. After calling TCPFlush() do not use TCPPut() again until TCPIsPutReady() returns TRUE.

- TCPIsGetReady(socket) returns TRUE if there is a TCP datagram destined for this socket that must handled. The Microchip TCP/IP stack verifies all header checksums, and that the TCP packets are received in order, and that there are no duplicate TCP packets. If the TCP packet is not handled now, the datagram will be lost by the next task time.

- TCPGet(socket, *c) and TCPGetArray(socket, *array, size) reads data out of the TCP datagram data field. TCPGet() returns TRUE if successful, and TCPGetArray() returns the number of bytes read.

- TCPDiscard(socket) discards any remaining data left in the TCP datagram and frees up the Ethernet receive buffer for more data. This is automatically done at the end of the task time.

- The source code for TCPServer.Exe is on the Development Tools CD.

FURTHER STUDY

A Try EX13B.C with TCPClient.Exe.
Note: *In this example the PIC18F67J60 is the server and the PC is the client..*

14 HYPERTEXT TRANSFER PROTOCOL (HTTP)

❑ The last chapter reviewed a custom-made TCP client that talked to a simple TCP server running on the PC. This chapter will focus on using the Microchip TCP/IP stack and the develment kit to make a TCP server, in particular a webserver.

❑ HTTP is the protocol used to serve and request web pages from webservers. The protocol is fairly simple: requests are made in ASCII strings, where each parameter of the request is a separate line of ASCII. An empty line is used to denote end of parameters. After a client makes its requests, the web server answers with its response. First the webserver responds with its own parameters, where each parameter is a seperate line of ASCII. An empty line is used to denote end of parameters, and then any data following is considered the web page or the content that was requested by the web client.

❑ Most HTTP servers listen to TCP port 80, but this is not a requirement. Typically a web browser (such as Internet Explorer) opens a communication socket to TCP port 80 and will not close the socket until after a user closes the web browser or goes to a different site. By keeping the TCP port open as long as possible, it can speed up the time it takes to download a page by not having to open/close a socket for each request, especially since a webpage may have linked images that need to be downloaded to properly display the page.

❑ For example, here is a simple request:

```
GET /DIRECTORY/FILE HTTP/1.1
```

❑ /directory/file is the file requested. There are many other options that may be found in a HTTP requests, but the only necessary one is the GET command which tells the webserver which file to download. The file requested does not have to be a webpage, HTTP can be used to transfer any kind of file – an example would be when downloading a ZIP file or a PDF from a website. /directory/file is relative to the host domain. For example, when viewing http://www.google.com/dir1/dir2/file.html, the GET request would be /dir1/dir2/file.html. Asking for www.google.com would result in a GET/ (this is called the root index).

❑ Here is a simple response to the above request:

```
HTTP/1.1 200 OK
Content-Type: text/html

insert webpage here
```

❑ There can be many other responses, such as content-length, date/time, etc, but these are the only two that are needed. The first line is the error code result as a response to the GET command. 200 is an error code for success, 404 is the error code for page missing, etc. The second line is the response that tells the browser what format the resulting data is in. Do not forget that after all response options, a blank line is needed before sending data.

❑ To see more HTTP requests examples, use Wireshark to packet sniff while surfing with a web browser.

❑ To demonstrate the server mode of the TCP/IP stack, a simple webserver example has been created and can be found on the Development Tools CD as EX14.C. Compile EX14.c and run on the prototyping board. Once the example is running, the following will appear on the LCD:

```
192.168.100.7
Listening
```

❑ The top line of the LCD is the IP address of the development kit. In a web browser, open that IP address. There should be a page that displays the analog-to-digital values for AN0:
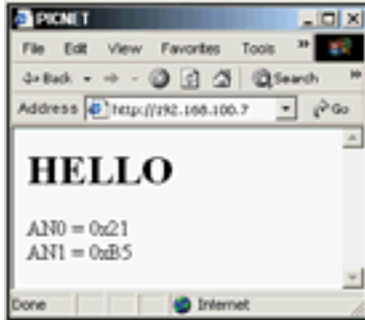
❑ Turn the potentiometer on the prototyping board and refresh the page to change what is displayed on the webpage. While a web browser is connected to the prototyping board, the IP address of the web browser will be displayed on the second line of the LCD. If no browser is connected, it should say 'Listening'.

NOTES

- TCPListen(port) will open the specified port and listen for a connection. TCPListen() will return a socket number if successful, or INVALID_SOCKET if there are no more sockets left. To have multiple sockets listening to the same port, call TCPListen() more than once with the same port.

- HTTPTask() only creates one socket, and therefore, only one web browser may connect at once. Add more sockets for more simultaneous connections by changing HTTPSocket and state in HTTPTask() to be an array; where each element of such array represents the socket and state for each connection. HTTPTask() will have to iterate though the state machine for each socket within one task time.

- This simple webserver does not parse any of the requests made by the websever. In particular, it does not parse the GET command to find the requested page. The webserver assumes it is only asking for one page.

❑ Chapters 12 and 13 used custom PC applications to send and receive data to the microcontroller. While this method works, each PC needs that specific application installed on the computer – and some computers cannot run those applications because of incompatible processor architecture or incompatible operating systems. Every modern operating system has a web browser, and implementing a web interface on the embedded device may give access to the device from any computer. Such web interfaces can be implemented using Comman Gateway Interface (CGI).

❑ CGI is the method used to transfer data from the client (or web browser) to the web server at which point the web server can execute the data and generate a dynamic page based upon that content. An example is google.com – When entering a search term on google.com, CGI is used to pass that value to the web server, at which point google.com executes the search and reports the result.

❑ In order to use CGI, a web server must support it. The previous chapter created a very simple webserver, but it did not include CGI support. This chapter will look at using a more advanced HTTP server that is included in the TCP/IP stack that supports CGI and multiple pages.

NOTES

● The HTTP server included by CCS was written by CCS, and is different than the HTTP server that was originally included by Microchip.

❑ There are two methods by which the web browser can send data to the HTTP server, GET and POST. The previous chapter reviewed GET, although no CGI data was sent. GET sends CGI data by appending key/value pairs onto the request line. The following is an example request:

```
GET /directory/file.html?KEY1=VALUE1&KEY2=VALUE2&KEY3=VALUE3 HTTP/1.1
```

❑ This looks similar to the GET request that was demonstrated in the previous chapter, but the key/value pairs are appended after the filename. The '?' character is used to denote parameters that the HTTP server is supposed to parse for CGI when using GET. Each key/value pair is then seperated by the '&' character.

❑ When using the POST method, CGI data is appended to the end of the HTTP request. Here is an example:

```
POST /directory/file.html HTTP/1.1
Content-Length: xx

KEY1=VALUE1&KEY2=VALUE2&KEY3=VALUE3
```

- ❑ For both GET and POST examples there may be many more parameters being passed, and these examples only show the required parameters. When a POST request is made, an important parameter is Content-Length, which tells the server how many characters of CGI data are being sent.

- ❑ GET requests are easier to develop and use because parameters can be passed using the location bar of the web browser, but there is a limitation of 255 characters for the GET request. (This is a standard limitation of HTTP, not a limitation based upon the Microchip TCP/IP stack or CCS HTTP server). POST gives you more security as the data is not shown in the location bar, and there is no maximum number of characters that POST can handle.

- ❑ Using the more advanced HTTP server can be done by defining STACK_USE_HTTP to TRUE before including the TCP/IP stack header file stacktsk.h. The application must also include three callback functions that the HTTP server must use for finding pages and processing CGI data. Open ex15.c in the examples directory. Compile and run ex15.c on the prototyping board.

- ❑ Once the example is running in the prototyping board, use the web browser to open the IP address of the prototyping board. (The IP address of the prototyping board is set in IPAddrInit() and is displayed on the top line of the LCD). A web page with a form allows for change of the LEDs or the message displayed on the LCD:

- ❑ The website uses an HTML input form to change the data displayed on the LCD and to change the status of the two LEDs. Click on the "Analog Readings" link to go to another page on the prototyping board server that displays the current ADC reading of AN0.

- StackTask() will call HTTP_Task() to automatically answer HTTP requests made to the unit. In order to transfer dynamic content HTTP_Task() will use three call back functions that the application must provide: http_get_page(), http_format_char(), http_exec_cgi().

- http_get_page(char *filename) is a callback function the application must provide, and will be called by HTTP_Task() to find the requested GET/POST page. Instead of implementing a file system on a device, this HTTP server expects the pages to be stored into program memory. The http_get_page() then must find the specified filename in ROM, and return the location in program memory. If this page is not in ROM http_get_page() must return 0 to indicate page not found.

- http_format_char(int32 file, char id, char *str, int8 max_ret) is a callback function the application must provide, and returns special formatting information for the web pages stored in the program memory. For special formatting, use a % character, when HTTP_Task() is serving a web page and sees a % it calls this function with the id variable set to the formatting character. *str is where the callback function must save the formatting result, and it should not store more than max_ret characters to this pointer (buffer-overrun protection). File is the address in ROM of the webpage being served, if special formatting characters need special meaning depending on what page is being served. This function must return the number of characters saved to *str.

- http_exec_cgi(int32 file, char *key, char *val) is a callback function the application must provide, and is called with each key=value pair read in the GET/POST request. File is the location in ROM of the webpage being served, in the case special processing is desired, depending on what page is being served.

- http_exec_cgi() is called for each key/value pair before any web data is sent to the web browser.

- HTML forms are used to place user input fields in webpages. For more help about HTML forms, see documentation about <FORM> and <INPUT> tags in any HTML documentation.

- The example uses GET transactions. Try this example using POST by changing the <FORM method=GET> to <FORM method=POST> in HTML_INDEX_PAGE[].

16 FILES, FILE SYSTEMS, AND OTHER ADVANCED HTTP SERVER OPTIONS

❑ Chapter 15 created a basic web server with support for CGI. The HTML web pages for this server were written directly into the C source code as constant strings in program memory. This means the web server source code has to be recompiled every time the web pages are to be changed. Also, the HTML must be formatted such that the compiler interprets it correctly. Any images included in the web page must be linked from a remote location on the Internet; images cannot be stored on the web server.

❑ The http2 server was created to allow users to create web pages on the PC and use them with the web server. Web pages can be created in any text editor or visual editor, and do not have to be embedded into the source code of the web server. Images can be stored on the http2 web server. The http2 server also includes many options that provide advanced features to the web server such as file systems, file transfers, and authentication.

NOTES

- The http2 server uses a similar set of callbacks as the original HTTP server to provide dynamic content to the web pages.
- The http_get_page() callback has been removed and no longer has to be provided by the application. Files are automatically opened by the http2 server when requested.
- http_format_char() and http_exec_cgi() now pass the name of the file being served by the http2 server instead of the ROM address as was the case in the previous HTTP server.
- The functionality of http_format_char() and http_exec_cgi() are otherwise the same as before.
- Other callbacks may be required when using other features of the http2 server, such as FTP or authentication.

❑ To provide this support, the http2 server can use one of two possible file systems: MPFS or FAT. To use either of these file systems, define either STACK_USE_MPFS or STACK_USE_FAT to TRUE before including the TCP/IP stack header file, stacktsk.h. Only one file system may be used at a time.

NOTES

- MPFS (Microchip PIC File System) is a simplified file system developed by Microchip for use with their TCP/IP stack. It was modified for use with the CCS C compiler and CCS http2 server.
- FAT (File Access Table) is a file system developed by Microsoft for use with the Windows operating system.

❑ MPFS allows the http2 server to store files either in program memory or on an EEPROM chip. Define either MPFS_USE_EEPROM or MPFS_USE_PGRM before including stacktsk.h. MPFS uses a program named mpfs.exe to create the file system image that the web server accesses to load the web pages. An MPFS image does allow an individual file to be modified or deleted; a whole new MPFS image has to be compiled and re-loaded if a file needs to be changed. To compile an MPFS image, run mpfs.exe from a command prompt as follows:

```
mpfs /ll /r(0) /k <InputDir> <OutputFile>
```

❑ The /ll option tells MPFS to use 32-bit addressing. The /r(0) option tells MPFS not to allocate a storage block before the MPFS (this is taken care of in the firmware). The /k option tells MPFS to keep carriage return and line feed characters. These are not generally needed in the HTML, but occasionally cause problems when removed. The <InputDir> is the directory that contains all of the files that should be added to the image. MPFS will add all of the files in this directory, so it should contain only files that are to be added. The <OutputFile> is the name of the output file that the MPFS utility will create.

❑ When using MPFS with program memory, the image must be included when the source code is compiled. This is done using the #import directive in mpfs.h when MPFS_USE_PGRM is defined. Rename the MPFS image "mpfsimg.bin" and put it in the project's directory. It will now be included when you compile the web server.

❑ When using MPFS with EEPROM, the MPFS image is downloaded to the device after it is programmed and connected to the network. The MPFS image is created the same way as it was when using MPFS with program memory. An MPFS image is then downloaded to the device using one of the two supported file transfer protocols: FTP or TFTP.

NOTES

● The EEPROM chip on the CCS development kits are too small to hold an entire MPFS image. For this reason, program memory must be used when utilizing MPFS with a CCS development kit.

● MPFS supports short (8.3) filenames only.

❑ The FAT file system was implemented for use with a SD Card on the CCS development kits. It allows for web pages and images to be created and added to the card using a PC and then used with the web server. When using the FAT file system, files can be added, removed, or modified individually—either by inserting the card into a card reader on a PC or by using a file transfer protocol: FTP or TFTP.

NOTES

- FAT supports the same folder/directory structure that a PC uses. Folders and directories, however, are not supported by the http2 server. Only files in the root directory may be accessed.

- The FAT driver supports filenames of up to 30 characters. They do not have to be in 8.3 format.

❑ The http2 server includes the MIME/encoding type header in every HTTP response it sends. This encoding type is determined solely based on the extension of the file requested. File types that do not contain dynamic content (such as images) are also given an extremely old datestamp to tell the web browser to cache this file. This decreases loading times of pages that do not contain dynamic content. The types of files that are supported:

- HTML files (.htm extension)
- Text files (.txt)
- XML files (.xml)
- GIF images (.gif)
- JPEG images (.jpg)
- PNG images (.png)
- Icons (.ico)
- Files with any other extension will be interpreted as plain text.

Support for additional file types can be added by modifying the tcp_http_put_file() function in http2.c

- There are several files which can be included to provide improved functionality to the http2 server. The file "index.htm" will be called when no file is explicitly specified in the URL. The http2 server allows for custom error pages to be loaded when the server cannot find a file (error404.htm), or when authentication fails (error401.htm).

- When using MPFS with EEPROM or FAT file systems, files or file images may be uploaded to the device from a PC using a file transfer program. Two different file transfer protocols are supported by the http2 server: FTP (File Transfer Protocol) and TFTP (Trivial FTP). These two protocols allow for the files on the web server to be added, deleted, or modified without having to re-compile the program. To use either of these protocols, define STACK_USE_FTP or STACK_USE_TFTP to TRUE before including "stacktsk.h".

- FTP is a connection-based file transfer protocol built over TCP. This gives FTP the increased reliability that comes with TCP. FTP uses a larger data packet size than TFTP which makes it ideal for relatively large file transfers. The FTP server also uses relatively more ROM and less RAM than the TFTP server.

NOTES

- The FTP server does not support all of the FTP commands. The FTP commands that are supported include: "USER" (login user name), "PASS" (login password), "QUIT" (quit the session), "STOR" (add or modify a file on the server), "PORT" (change the TCP data port), "ABOR" (abort a data transfer), and "DELE" (delete a file from the server).

- The FTP server was designed to work with most command-line FTP programs, such as the one included with Windows. Many FTP programs (such as those built into web browsers) automatically issue an initial "PWD" or "LIST" command. These commands are not supported by the FTP server and, therefore, neither are browser-based FTP programs.

- The FTP server requires a callback be provided by the application. FTPVerify(char *login, char *password) is called to authenticate an FTP login attempt. login is the user name provided by the user attempting to access to the FTP server. password is the password provided. The callback should return TRUE of the user name and password are OK; false if verification fails.

❑ TFTP is a simplified version of the FTP protocol that is built over UDP. There is no verification method provided when using TFTP. Because TFTP uses UDP and not TCP, some of the reliability of FTP is lost. There are some error checking and reliability features built into the TFTP protocol to make up for this, however. TFTP uses less ROM, but more RAM than FTP. This makes it ideal for smaller transfers or frequently-occurring ones.

NOTES

- The TFTP server can be optimized for either reduced RAM-usage or faster transfers speeds. RAM can be freed by reducing the receive buffer used in the server. Reduce TFTP_BUFFER_SIZE in tftp.h to reduce free up some RAM. This will also slow the transfer speed. TFTP_BUFFER_SIZE can be increased to add marginal gains to the speed of the transfer. TFTP_BUFFER_SIZE should not be made larger than 512 (bytes).

- The TFTP server expects that the block size of the file transfers will be 512 bytes. This is the default specified in the TFTP RFC document. Some TFTP client software may attempt to use a larger block size; change the settings to use a 512 bytes transfer block.

- The TFTP server also assumes all transfers are made in binary mode—also known as image or octet mode. This is especially important when using MPFS as the file system on the web server.

- The TFTP server works well with the Windows command-line TFTP program. To use this, be sure to specify the -i option for image mode.

❑ The http2 server also includes the ability to add HTTP authentication to the web server. This will password-protect some or all of the files stored on the server to prevent unauthorized access to them. To enable authentication, define HTTP_USE_AUTHENTICATION as TRUE before including "stacktsk.h" in your application.

❑ To determine what files to authenticate, the http2 server requires a file named "htaccess.txt" to be present (in the root directory if using FAT) on the server's file system. The format of htaccess.txt is as follows: user|password|file1|file2|file3.

- The http2 server uses two sources for authentication. The user name and password can be specified in the htaccess.txt file. This allows the user name and password to be changed without having to re-compile and load new firmware to the web server. For the second method of authentication, the HTTP stack will callback function, http_check_authentication(). If the user input matches the user name/password from the htaccess.txt, or http_check_authentication() returns TRUE then authentication is granted.

NOTES

- If using the FAT file system, directories can be specified in htaccess.txt.

- To password protect the entire file system then use / as the filename.

- To have the HTTP stack ignore the username/password from htaccess.txt then leave those fields blank.

- The http_check_authentication(char *fileName, char *user, char *pwd) function should return TRUE if the user name and password are correct for the given file name. Return FALSE if authentication is incorrect for the given file.

- The user name and password in htaccess.txt uses the same case-sensitive password for all files.

SIMPLE MAIL TRANSFER PROTOCOL (SMTP)

❑ The Simple Mail Transfer Protocol, or SMTP, is the current de-facto standard mail transfer protocol used on the Internet today. SMTP uses a simple, text-based protocol from which a client or server can relay an e-mail message. When a client uses SMTP to send an e-mail, if the e-mail is not destined to someone on that server it will be relayed to the next SMTP server in a manner similar to how a real-life post-office relays paper mail.

❑ Since SMTP uses a text based protocol, the standard telnet tool can be used as an SMTP client. SMTP commonly uses TCP port 25. Here is an example transaction, which can be reproduced by telneting into a SMTP server. The shaded lines are sent by the client (you), the non-bold lines are responses sent by the SMTP server:

220 mail.host.com SMTP	The 220 is the result code for service ready. Any other number is an error. The data after the result code will be different for each server.
ehlo my.host.com	Identify ourselves as an SMTP client, with this host name. Most SMTP servers ignore the host name here.
250-parameter 1 250-parameter 2 250 parameter 3	SMTP accepts our identification as a client, and responds with it's configuration parameters (such as max size, MIME types, etc). 250 is a successful error code, any other number would be an error.
mail from: me@somewhere.com	Client identifies the sender for this e-mail. Some SMTP servers will verify that this domain exists first.
250 Sender <me@somewhere.com> Ok	SMTP accepts the sender address. 250 is a successful error code.
rcpt to: you@somewhere.com	Client identifies the recipient for this e-mail. Some SMTP servers will verify that this domain exists first.
250 Recipient <you@somwhere.com> Ok	SMTP accepts the recipient address. 250 is a successful error code.
data	Tell the SMTP server that any following data being sent is the body of the e-mail.
354 Ok Send data ending with <CRLF>.<CRLF>	SMTP server is ready to accept data. If you want to stop sending e-mail you must put a period on an empty line.
From: me@somewhere.com **To:** you@somewhere.com **Subject: An E-Mail**	The SMTP client sends standard e-mail headers. Notice that the mail from: and rcpt to: commands do not create the e-mail header. If you use a different From and To address than what was specifeid in the mail from: and rcpt to: commands many spam filters may delete the e-mail as header spoofing is common from spammers.
Body of the email	This is the body of the e-mail.
.250 Message received	Message was successfully read by the server. There is no guarantee that it will reach the recipient, however.
QUIT	Gracefully terminate session with SMTP server. You could also just close the TCP socket.
221 Goodbye	Server accepts our session termination, and closes the connection.

❑ CCS has implemented an SMTP engine as an add-on to the Microchip TCP/IP stack that will properly handle all the SMTP commands. To demonstrate this SMTP engine, look at the example code ex16.c that is included on the Development Tools CD. Before running this example code, look at the function MYSMTPInit() and change the IP address to the IP address of your SMTP server. Compile and run this example on the prototyping board.

❑ When running and idle, the following will appear on the LCD:

```
SMTP Idle
Emails Sent: 0
```

❑ Pressing the button next to the potentiometer will send an e-mail using the SMTP IP address and e-mail addresses provided in the MYSMTPInit() function. On the first line, the LCD will show the current status of the SMTP connection in progress. When completed the second line will increment the counter of e-mail sent, or if there was an error it will display the error.

NOTES

● SMTPConnect(*IP, port, char *from, char *to, char *subject) will initiate the SMTP engine. If needed, it will do an ARP lookup first. It will return TRUE if a socket was created, but that does not mean a successful SMTP connection was made. from, to and subject must be global. port is almost always 25.

● The SMTP engine can only handle one socket at a time. Therefore, it is not possible to call a SMTPConnect() until the previous SMTPConnect() has been disconnected. SMTPIsFree() will return TRUE if you can call SMTPConnect().

● Once SMTPConnect() is called and returned TRUE, poll SMTPIsPutReady() and SMTPLastError(). SMTPIsPutReady() will return TRUE if a successful SMTP connection was made and is ready for the body of the e-mail. SMTPLastError() will return a non-zero value if there was an error creating the SMTP connection. Look at the SMTP_EC enum in smtp.h for documentation on the error codes returned by SMTPLastError()

● SMTPPut(char c) can be use to put a character into the body of the e-mail. SMTPIsPutReady() must return TRUE before you use SMTPPut().

● SMTPDisconnect() will finish off the e-mail and close the connection. To determine if the e-mail was accepted by the SMTP server, wait for SMTPIsFree() to return TRUE and then check SMTPLastError() to verify that it still returns 0.

● Use the SMTP server provided by your ISP. If unknown ask the ISP. The reason for this is that in the war on spam, almost all SMTP servers block access to clients who are not on their network.

● Due to the war on spam, many internet service providers are placing restrictions upon SMTP servers. Such restrictions may be authentication, sender-id, message-id and maximum message-per-minute rate. This engine deals with none of those restrictions. Its very likely in the future that it will be impossible for a microcontroller to have the resources to send e-mail using SMTP.

References

Book: TCP/IP Lean

By: Jeremy Bentham

Summary: An excellent book that details the internals of Ethernet, IP, ARP, UDP, TCP, PPP and more. Its primary focus is how to develop a TCP/IP application on microcontrollers with limited resources.

Application Note: AN833

By: Microchip

Summary: Documentation of Microchip's TCP/IP, including example code and API specifications.

Software: Wireshark

URL: http//wireshark.org/

Summary: Excellent, free, open-source packet sniffing software for the computer. Can analyze network traffic of many types: Ethernet, 802.11, IP, UDP, TCP, PPP and more. Work in Windows, Linux, BSD, Mac OS X and more.

On The Web

Comprehensive list of PIC® MCU Development tools and information	www.mcuspace.com
Microchip Home Page	www.microchip.com
CCS Compiler/Tools Home Page	www.ccsinfo.com
CCS Compiler/Tools Software Update Page/ TCP/IP Stack Download	www.ccsinfo.com click: Support → Downloads
C Compiler User Message Exchange	www.ccsinfo.com/forum
Device Datasheets List	www.ccsinfo.com click: Support → Device Datasheets
C Compiler Technical Support	support@ccsinfo.com

Other Development Tools

EMULATORS

The ICD used in this booklet uses two I/O pins on the chip to communicate with a small debug program in the chip. This is a basic debug tool that takes up some of the chip's resources (I/O pins and memory). An emulator replaces the chip with a special connector that connects to a unit that emulates the chip. The debugging works in a simulator manner except that the chip has all of its normal resources, the debugger runs faster and there are more debug features. For example an emulator typically will allow any number of breakpoints. Some of the emulators can break on an external event like some signal on the target board changing. Some emulators can break on an external event like some that were executed before a breakpoint was reached. Emulators cost between $500 and $3000 depending on the chips they cover and the features.

DEVICE PROGRAMMERS

The ICD can be used to program FLASH chips as was done in these exercises. A stand alone device programmer may be used to program all the chips. These programmers will use the .HEX file output from the compiler to do the programming. Many standard EEPROM programmers do know how to program the Microchip parts. There are a large number of Microchip only device programmers in the $100-$200 price range. Note that some chips can be programmed only once (OTP) and some parts need to be erased under a UV light before they can be re-programmed (Windowed). CCS offers the Mach X which is a stand-alone programmer and can be used as an in-circuit debugger.

PROTOTYPING BOARDS

There are a large number of Prototyping boards available from a number of sources. Some have an ICD interface and others simply have a socket for a chip that is externally programmed. Some boards have some advanced functionality on the board to help design complex software. For example, CCS has a Prototyping board with a full 56K modem on board and a TCP/IP stack chip ready to run internet applications such as an e-mail sending program or a mini web server. Another Prototyping board from CCS has a USB interface chip, making it easy to start developing USB application programs.

SIMULATORS

A simulator is a program that runs on the PC and pretends to be a microcontroller chip. A simulator offers all the normal debug capability such as single stepping and looking at variables, however there is no interaction with real hardware. This works well if you want to test a math function but not if you want to test an interface to another chip. With the availability of low cost tools, such as the ICD in this kit, there is less interest in simulators. Microchip offers a free simulator that can be downloaded from their web site. Some other vendors offer simulators as a part of their development packages.

CCS Programmer Control Software

The CCSLOAD software will work for all the CCS device programmers and replaces the older ICD.EXE and MACHX.EXE software. The CCSLOAD software is stand-alone and does not require any other software on the PC. CCSLOAD supports ICD-Sxx, ICD-Uxx, Mach X, Load-n-Go, and PRIME8.

Powerful Command Line Options in Windows and Linux
- Specify operational settings at the execution level
- Set-up software to perform, tasks like save, set target Vdd
- Preset with operational or control settings for user

Easy to use Production Interface
- Simply point, click and program
- Additions to HEX file organization include associating comments or a graphic image to a file to better ensure proper file selection for programming
- Hands-Free mode auto programs each time a new target is connected to the programmer
- PC audio cues indicate success and fail

Extensive Diagnostics
- Each target pin connection can be individually tested
- Programming and debugging is tested with known good programs
- Various PC driver tests to identify specific driver installation problems

Enhanced Security Options
- Erase chips that failed programming
- Verify protected code cannot be read after programming
- File wide CRC checking

Automatic Serial Numbering Options
- Program memory or Data EEPROM
- Incremented, from a file list or by user prompt
- Binary, ASCII string or UNICODE string

CCS IDE owners can use the CCSLOAD program with:
- MPLAB®ICD 2/ICD 3
- MPLAB®REAL ICE™
- **All CCS programmers and debuggers**

How to Get Started:
Step 1: *Connect Programmer to PC and target board. Software will auto-detect the programmer and device.*
Step 2: *Select Hex File for target board.*
Step 3: *Select Test Target. Status bar will show current progress of the operation.*
Step 4: *Click "Write to Chip" to program the device.*

Use the Diagnostics tab for troubleshooting or the ccsload.chm help file for additional assistance.

RJ45

330
330
LEDB
LEDA

5 T+
4 TCT
6 T−
1 R+
3 RCT
2 R−
7
8 POE

+3.3V
Fbead
.01
.01
49.9 1%
49.9 1%
49.9 1%
49.9 1%
TPOUT+
TPOUT−
TPIN+
TPIN−

MMC−SD2
1
2
3
4
5
6
7
8
9
10

+3.3V 100K
+3.3V 100K
+3.3V 100K
+3.3V 4.7K
+3.3V 4.7K
+3.3V
C2
C5
C3
C4
E5

.1
+3.3V
C3
C5
8 VCC
7 HOLD
6 SCK
5 SI
1 CS
2 SO
3 WP
4 GND
AT25256A
C1
C4

+3.3V
150uf
22 uh
MBRS340T3
IRF7416
0.1
6 CS
7 EXT
1 OUT
8 GND
5 V+
3 SHDN
4 REF
2 3/5
MAX1626
.47
.1
150uf
POWER IN

B5

B4

B3

Pushbutton A4

Ethernet

SD Card

G G
G G
NC NC
NC NC
NC NC
A4 A5
C0 A3
B4 B5
B2 B3
B0 B1
E4 E5
E2 E3
E0 E1
D2 G4
D0 D1
+33 +33
+33 +33

Pot A2

Power 9V DC

Pushbutton RESET

RS232 C6, C7

ICD Connector