H04L2A
Software for Real-Time and Embedded Systems

Project 2

DHCP Relay on a PIC18F97J60-based Devboard

Group 14
Olle Calderon
Orsolya Lukács-Kisbandi

1 Introduction

The aim of this project was to understand and implement a DHCP relay on a development board based on the `PIC18F97F60` microprocessor by Microchip. We were provided the source code for the TCP/IP suite by Microchip which we had to modify in order to fit our needs. The DHCP server and client source codes were used to a great deal in order to merge the functionalities of these two network entities into a relay.

There were troubles with the network settings and difficulties with the router. Because of these problems, we lost a significant amount of time and we were unable to fully test the functionality of the relay. Be that as it may, the source code is provided with this report which will outline the functionality of the relay, although it was never fully evaluated.

2 Documentation

2.1 Principles of DHCP

DHCP[1] is a protocol used to distribute IP addresses to clients over UDP[2]. Since the protocol is dynamic, IP addresses are assigned dynamically in a certain range.

The client sends a broadcast packet (DISCOVER) over the network, notifying it wished to get an IP. The DHCP server intercepts the packet and replies with a packet containing an IP address which the client may use (OFFER). The server reserves this IP locally. When the client receives the offer, it responds with another broadcast packet, confirming it will use the offered IP address (REQUEST). On receiving the request, the server will acknowledge the request by replying with an ACK packet. At this point, the DHCP process is completed.

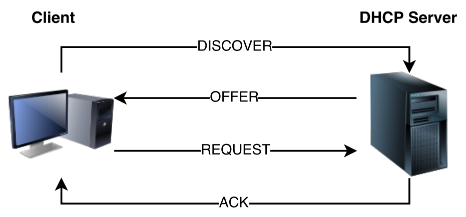

Figure 1: DHCP communication between client and server on the same subnet.

2.2 DHCP Relay

On large networks spanning a wide range of IP addresses, it may be of interest to divide the network into multiple subnets. In this configuration, it is not possible for the client to communicate with the DHCP server directly due to the fact that it cannot get a routable address to it. A solution to this is to use relays on these subnets, which serve as middle men for DHCP request over each individual subnet. The relays forward DHCP packets between the server and the client.

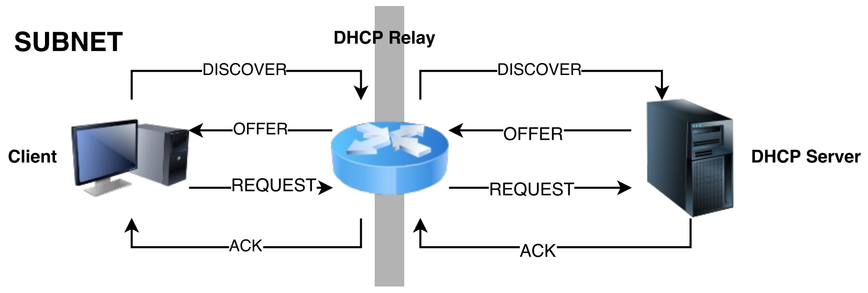

Figure 2: DHCP communcation using a relay.

[1]Dynamic Host Configuration Protocol.
[2]User Datagram Protocol.

The process as described earlier is the same from the client's and server's perspective. The relay, having communication channels to both the client and the server, will listen to broadcast messages on the client's subnet, forward these messages to the server via unicast and stores its own IP in the gateway address field in the packet header. This IP is used by the server to determine which subnet to reserve an IP on. The server replies with the reserved IP to the given gateway address, which the relay can forward to its subnet. This will then be intercepted by the client. The process continues for the REQUEST, OFFER, and ACK messages.

2.3 ASG Diagram of Relay

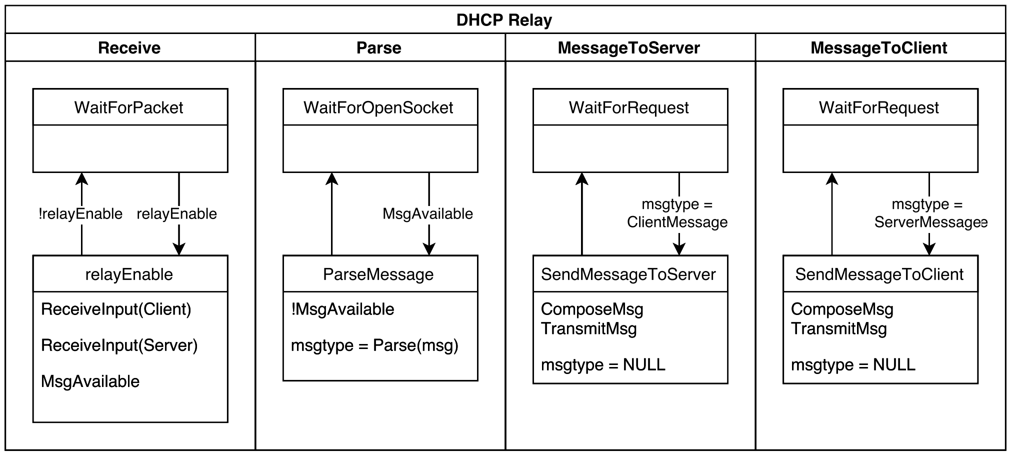

Figure 3: ASG diagram of DHCP relay.

The ASG diagram (figure 3) shows the logic behind the program. The DHCP relay's tasks were separated into 4 separate procedures:

- *Recieve* opens two sockets, a relay-server socket and a client-relay socket and starts listening to these sockets until they are enabled.
- *Parse* waits for open sockets, if socket is open and a message is available it decides what kind of message it is (DISCOVER, OFFER, REQUEST, ACK, or otherwise) and which process should recieve it, then marks the MsgAvailable field as false and returns to waiting for another message.
- When *MessageToServer* is invoked by *Parse*, meaning the message input requires communication between the relay and the server, it forwards the message to the server. The server will then generate the response and sends it back to the relay, which is requested to forward it to the client.
- When *MessageToClient* is invoked by *Parse*, the communcation will take place between the client and the relay. The recieved message will be transmitted to the client, which generates the appropriate response and sends it back to the relay.

The functionalities of *MessageToServer* and *MessageToClient* are seemingly very similar and indeed they are. However, the message composed and transmitted will be different depending on which part to receive the message (client or server) and the message type (DISCOVERY, REQUEST etc.). In reality our code looks slightly different. The `relayEnable` is represented by a successful establishment of client and server sockets. The function *ReceiveInput* recieves and parses the messages invoking four different functions, two of which belong to *MessageToServer* (`DiscoveryToS` and `ReqToS`) and two which belong to *MessageToClient* (`OfferToC` and `AckToC`). These functions are quite similar, which mirrors the similarities between *MessageToServer* and *MessageToClient* in the ASG diagram.

The `main` function in the code will try to open the required UDP sockets and, upon establishing these, go into a receiving state using the `ReceiveInput` function. It will listen to the sockets, receive message from these and parse them. The relay will then, dependent on the type of message received, compose a suitable response message and transmit this to the appropriate receiver. The functions `DiscoveryToS`, `OfferToC`, `ReqToS` and `AckToC` are responsible for composing these responses and transmitting them.

The Microchip TCP/IP stack is inherently a cooperative multitasking system[3].

The code can definitely be refactored to a more compact form, since the message composing and transmitting functions are very similar.

3 User manual

3.1 Router configuration

The router has to be configured to accomadate for our DHCP relay setup. Firstly, the internal DHCP server has to be disabled. Also the NAT service should be disabled. The IP address of the PIC is static (192.168.97.70), which means that the router has to be configured to forward UDP packets to this IP address. The PIC will relay UDP packets to the server, so the IP of the WAN port (e.g. port 1) on the router has to be set to the static IP 192.168.1.2, subnet 255.255.255.0.

It's important to note that this procedure requires you to connect to the router through its MAC address. Some network settings and setups will not allow such connections. We had big troubles with connecting to the router using only its MAC address, the solution to which we are not sure of what it specifically was since many methods were evaluated. So, your mileage may vary. Here are nevertheless a few factors that you may have to consider to be able to connect to the router using only its MAC:

- Turn off the firewall on your computer. Your firewall may block connections using only MAC addresses.
- Turn off any other network adapters (other ethernet interfaces, WiFI etc.). These may interfere in the connection.
- Use a computer with a native ethernet port. Adapters from e.g. thunderbolt/USB to ethernet may cause problems.
- Misc. security settings on your computer. They might block direct connections of this kind.

[3]N. Rajbharti Microchip Technology Inc. "AN833 The Microchip TCP/IP Stack". In: 2002, pp. 1–2.

You can see if your computer notices the router on the network using an `arp -a` request. However, just because your computer can see the router and its MAC doesn't necessarily mean your computer can connect to it. This is dependent on how your network adapter is configured and the settings on your computer. Different operating systems will also have different default security and network settings and you have to adjust these to allow this kind of connection.

3.2 DHCP Server

Set up a DHCP server on a computer and connect it to the WAN port of the router (port 1). It should be set to the static IP 192.168.1.2 and offer IP addresses in a suitable range (e.g. 192.168.1.3-192.168.1.32).

3.3 Client

Connect your client computer to a free port on the router. The client should be granted an IP address in the range provided by the DHCP server, via the relay.

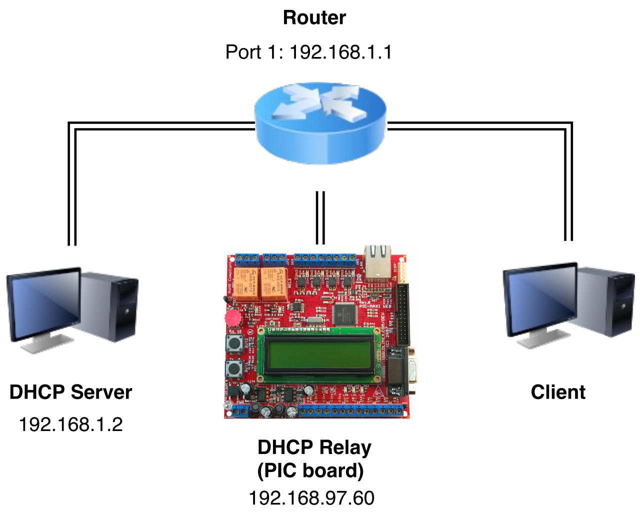

Figure 4: Connection diagram.

4 Instructions

To compile the code, you need to install a few things on your computer. This guide is for UNIX and UNIX-like systems and these commands were succesfully used under Mac OSX[4] during the development of this code.

4.1 Prerequisites

First, you'll need to configure, make and install GPUTILS[5] and SDCC[6]. To do this, you might have to install `bison` and `flex` on your system (if the printouts in your terminal tells you these components are missing).

Now, go to `/usr/local/share/sdcc/lib/pic16` using your terminal and run these linker scripts:

```
ln -s libm18f.a libm18f.lib
ln -s libsdcc.a libsdcc.lib
ln -s libio18f97j60.a libio18f97j60.lib
ln -s libdev18f97j60.a libdev18f97j60.lib
```

After running the linker scripts, your system is ready to compile the program.

4.2 Compiling

If you've followed the previous steps, your system should now be able to compile using the included `Makefile`. Using your terminal, go to the directory containing `RelayDHCP.c` and type the command `make RelayDHCP`. If the compilation was successful, the file `RelayDHCP.hex` (in the same directory as `RelayDHCP.c`) should now be generated (or if it already existed, replaced by the new code). If this file is not generated, you might want to try entering `sdcc -mpic16 -p18f97j60 RelayDHCP.c` followed by `sdcc -mpic16 -p18f97j60 -L/usr/local/lib/pic16 RelayDHCP.c` instead.

4.3 Download program to board

If you're not already there, go to the folder containing `RelayDHCP.hex` using your terminal. When the program is compiled, connect the devboard to your computer[7] using an ethernet cable. Run `tftp` in the terminal using IP address `192.168.97.60`, which is the IP address of the devboard. Enter `binary`, `trace` and `verbose` in the terminal when connected to the board using `tftp`. Type `put RelayDHCP.hex` into the terminal, but don't hit enter just yet. Push the reset button on the board, wait one or two seconds for the board to connect to your network and then hit enter. If the program is successfully transferred to the board, you should see in the terminal that packets have been sent and that ACKs are received properly. At this point, the board will reset and run the program you just transferred.

[4] We used version 10.11.6, but other version should probably work the same.

[5] We used v0.13.7, but other versions might also work (not tested, your mileage may vary).

[6] We used SRC-20091215-5595, but other versions might also work (not tested).

[7] We used a router in between, but you may be able to connect directly. Regardless, you need to configure your network to allow connection between your PC and the devboard.

5 CODE

```
#define __DHCPS_C
#define __18F97J60
#define __SDCC__
#define THIS_INCLUDES_THE_MAIN_FUNCTION
#define THIS_IS_STACK_APPLICATION
#define DHCP_LEASE_DURATION     60ul

#include <pic18f97j60.h>        //ML
#include "Include/TCPIPConfig.h"
#include "Include/TCPIP_Stack/TCPIP.h"
#include "Include/TCPIP_Stack/DHCP.h"
#include "Include/TCPIP_Stack/UDP.h"
#include "Include/GenericTypeDefs.h"

static  UDP_SOCKET      CSocket,SSocket;
APP_CONFIG AppConfig;

void ReceiveInput(UDP_SOCKET listen);
void DiscoveryToS(BOOTP_HEADER *Header);
void OfferToC(BOOTP_HEADER *Header);
void ReqToS(BOOTP_HEADER *Header);
void AckToC(BOOTP_HEADER *Header);
void DisplayString(BYTE pos, char* text);
static void InitAppConfig(void);
static void Receive();

int main(void){
        // Initialize Stack and application related variables in AppConfig.
    InitAppConfig();

    // Initialize core stack layers (MAC, ARP, TCP, UDP) and
    // application modules (HTTP, SNMP, etc.)
    StackInit();

    LCDInit();

        DisplayString(0,"START RELAY");
        while(1)
                Receive();
        return 1;
}

static void Receive(){
        CSocket = UDPOpen(DHCP_SERVER_PORT, NULL, DHCP_CLIENT_PORT);
        SSocket = UDPOpen(DHCP_CLIENT_PORT, NULL, DHCP_SERVER_PORT);
        if (CSocket == INVALID_UDP_SOCKET)
        {
                LCDErase();
                DisplayString(0 , "Client Socket Error!");
                return;
        }
        else if(SSocket == INVALID_UDP_SOCKET)
        {
                LCDErase();
                DisplayString(0 , "Server Socket Error!");
                return;
        }
        ReceiveInput(SSocket);  // Start listening
        ReceiveInput(CSocket);
        UDPDiscard();
        return;
}

void ReceiveInput(UDP_SOCKET listen) {
        BOOTP_HEADER clientHeader;
        BYTE Option, Len, i;
        DWORD dw;

        DisplayString (0,"ReceiveInput");
        if (UDPIsGetReady(listen) < 241u)       // Check to see if a valid DHCP packet
             has arrived
                return;
        UDPGetArray((BYTE*)&clientHeader, sizeof(clientHeader));
        if (clientHeader.HardwareType != 1u || clientHeader.HardwareLen != 6u)
                return;
        UDPGetArray((BYTE*)&dw, sizeof(DWORD));
        if (dw != 0x63538263ul)
                return;

        while (1){
                if (!UDPGet(&Option))   // Get option type
                        break;
                if (Option == DHCP_END_OPTION)
                        break;
                UDPGet(&Len);   // Get option length

                switch (Option) // Process option
                {
                case DHCP_MESSAGE_TYPE:
                        UDPGet(&i);
                        switch (i)
                        {
                        case DHCP_DISCOVER_MESSAGE:
                                DisplayString(0,"DISCOVER");
                                DiscoveryToS(&clientHeader);
                                break;
                        case DHCP_OFFER_MESSAGE:
                                DisplayString(0,"OFFER");
                                OfferToC(&clientHeader);
                                break;
                        case DHCP_REQUEST_MESSAGE:
                                DisplayString(0,"REQUEST");
                                ReqToS(&clientHeader);
                                break;
                        case DHCP_DECLINE_MESSAGE:
                                DisplayString(0,"DECLINE");
                                break;
                        case DHCP_ACK_MESSAGE:
                                DisplayString(0,"ACK");
                                AckToC(&clientHeader);
                                break;
                        default:
                                DisplayString(0,"DEFAULT");
                                break;
                        }
                        break;
                //case DHCP_PARAM_REQUEST_IP_ADDRESS:
                case DHCP_END_OPTION:
                        UDPDiscard();
                        return;
                }
                while (Len--)    // Remove any unprocessed bytes that we don't care
                    about
                        UDPGet(&i);
        }
}

void OfferToC(BOOTP_HEADER *Header)
{
        BYTE i,a;
        DWORD RequestedIP = 0;

        // Set the correct socket to active and ensure that
        // enough space is available to generate the DHCP response
        if (UDPIsPutReady(CSocket) < 300u)
                return;

        Header->BootpFlags = 0x8000;    //Send via broadcast

        UDPPutArray((BYTE*)&(Header->MessageType), sizeof(Header->MessageType));
        UDPPutArray((BYTE*)&(Header->HardwareType), sizeof(Header->HardwareType));
        UDPPutArray((BYTE*)&(Header->HardwareLen), sizeof(Header->HardwareLen));
        UDPPutArray((BYTE*)&(Header->Hops), sizeof(Header->Hops));
        UDPPutArray((BYTE*)&(Header->TransactionID), sizeof(Header->TransactionID));
        UDPPutArray((BYTE*)&(Header->SecondsElapsed), sizeof(Header->SecondsElapsed));
        UDPPutArray((BYTE*)&(Header->BootpFlags), sizeof(Header->BootpFlags));
        UDPPutArray((BYTE*)&(Header->ClientIP), sizeof(Header->ClientIP));
        UDPPutArray((BYTE*)&(Header->YourIP), sizeof(Header->YourIP));
        UDPPutArray((BYTE*)&(Header->NextServerIP), sizeof(Header->NextServerIP));
        UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr));
        UDPPutArray((BYTE*)&(Header->ClientMAC), sizeof(Header->ClientMAC));

        // Remaining 10 bytes of client hardware address, server host name: Null
            string (not used)
        for (i = 0; i < 64 + 128 + (16 - sizeof(MAC_ADDR)); i++)
                UDPPut(0x00);
                                    // Boot filename: Null string (not used)
        UDPPut(0x63);                               // Magic Cookie: 0x63538263
        UDPPut(0x82);                               // Magic Cookie: 0x63538263
        UDPPut(0x53);                               // Magic Cookie: 0x63538263
        UDPPut(0x63);                               // Magic Cookie: 0x63538263

        UDPPut(DHCP_MESSAGE_TYPE);      // Message type = REQUEST
        UDPPut(DHCP_MESSAGE_TYPE_LEN);
        UDPPut(DHCP_OFFER_MESSAGE);

        UDPPut(DHCP_SUBNET_MASK);       // Subnet mask
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&AppConfig.MyMask, sizeof(IP_ADDR));

        UDPPut(DHCP_ROUTER);    // Option: Router/Gateway address
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&AppConfig.MyIPAddr, sizeof(IP_ADDR));

        UDPPut(DHCP_IP_LEASE_TIME);     // Option: Lease duration
        UDPPut(4);
        UDPPut((DHCP_LEASE_DURATION>>24) & 0xFF);
        UDPPut((DHCP_LEASE_DURATION>>16) & 0xFF);
        UDPPut((DHCP_LEASE_DURATION>>8) & 0xFF);
        UDPPut((DHCP_LEASE_DURATION) & 0xFF);

        UDPPut(DHCP_SERVER_IDENTIFIER); // Option: Server identifier
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&(Header->NextServerIP), sizeof(Header->NextServerIP));

        UDPPut(DHCP_END_OPTION);        // No more options, mark ending

        // Add zero padding to ensure compatibility with old BOOTP relays that discard
             small packets (<300 UDP octets)
        while (UDPTxCount < 300u)
                UDPPut(0);

        UDPFlush();     // Transmit the packet
}

void AckToC(BOOTP_HEADER *Header)
{
        BYTE i,a;
        DWORD RequestedIP = 0;

        // Set the correct socket to active and ensure that
        // enough space is available to generate the DHCP response
        if (UDPIsPutReady(CSocket) < 300u)
                return;

        Header->BootpFlags = 0x0000;    //Send via unicast

        UDPPutArray((BYTE*)&(Header->MessageType), sizeof(Header->MessageType));
        UDPPutArray((BYTE*)&(Header->HardwareType), sizeof(Header->HardwareType));
        UDPPutArray((BYTE*)&(Header->HardwareLen), sizeof(Header->HardwareLen));
        UDPPutArray((BYTE*)&(Header->Hops), sizeof(Header->Hops));
        UDPPutArray((BYTE*)&(Header->TransactionID), sizeof(Header->TransactionID));
        UDPPutArray((BYTE*)&(Header->SecondsElapsed), sizeof(Header->SecondsElapsed));
        UDPPutArray((BYTE*)&(Header->BootpFlags), sizeof(Header->BootpFlags));
        UDPPutArray((BYTE*)&(Header->ClientIP), sizeof(Header->ClientIP));
        UDPPutArray((BYTE*)&(Header->YourIP), sizeof(Header->YourIP));
        UDPPutArray((BYTE*)&(Header->NextServerIP), sizeof(Header->NextServerIP));
        UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr));
        UDPPutArray((BYTE*)&(Header->ClientMAC), sizeof(Header->ClientMAC));

        // Remaining 10 bytes of client hardware address, server host name: Null
            string (not used)
        for (i = 0; i < 64 + 128 + (16 - sizeof(MAC_ADDR)); i++)
                UDPPut(0x00);
                                    // Boot filename: Null string (not used)
        UDPPut(0x63);                           // Magic Cookie: 0x63538263
        UDPPut(0x82);                           // Magic Cookie: 0x63538263
        UDPPut(0x53);                           // Magic Cookie: 0x63538263
        UDPPut(0x63);                           // Magic Cookie: 0x63538263
        UDPPut(DHCP_MESSAGE_TYPE);      // Message type = REQUEST
        UDPPut(DHCP_MESSAGE_TYPE_LEN);
        UDPPut(DHCP_ACK_MESSAGE);

        UDPPut(DHCP_SUBNET_MASK);       // Subnet mask
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&AppConfig.MyMask, sizeof(IP_ADDR));

        UDPPut(DHCP_ROUTER);    // Option: Router/Gateway address
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&AppConfig.MyIPAddr, sizeof(IP_ADDR));

        UDPPut(DHCP_IP_LEASE_TIME);     // Option: Lease duration
        UDPPut(4);
        UDPPut((DHCP_LEASE_DURATION>>24) & 0xFF);
        UDPPut((DHCP_LEASE_DURATION>>16) & 0xFF);
        UDPPut((DHCP_LEASE_DURATION>>8) & 0xFF);
        UDPPut((DHCP_LEASE_DURATION) & 0xFF);

        UDPPut(DHCP_SERVER_IDENTIFIER); // Option: Server identifier
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&(Header->NextServerIP), sizeof(Header->NextServerIP));

        UDPPut(DHCP_END_OPTION);        // No more options, mark ending

        // Add zero padding to ensure compatibility with old BOOTP relays that discard
             small packets (<300 UDP octets)
        while (UDPTxCount < 300u)
                UDPPut(0);

        UDPFlush();     // Transmit the packet
}

void DiscoveryToS(BOOTP_HEADER *Header) {
        BYTE i;

        // Set the correct socket to active and ensure that
        // enough space is available to generate the DHCP response
        if (UDPIsPutReady(SSocket) < 300u)
                return;

        UDPPutArray((BYTE*)&(Header->MessageType), sizeof(Header->MessageType));
        UDPPutArray((BYTE*)&(Header->HardwareType), sizeof(Header->HardwareType));
        UDPPutArray((BYTE*)&(Header->HardwareLen), sizeof(Header->HardwareLen));
        UDPPutArray((BYTE*)&(Header->Hops), sizeof(Header->Hops));
        UDPPutArray((BYTE*)&(Header->TransactionID), sizeof(Header->TransactionID));
        UDPPutArray((BYTE*)&(Header->SecondsElapsed), sizeof(Header->SecondsElapsed));
        UDPPutArray((BYTE*)&(Header->BootpFlags), sizeof(Header->BootpFlags));
        UDPPutArray((BYTE*)&(Header->ClientIP), sizeof(Header->ClientIP));
        UDPPutArray((BYTE*)&(Header->YourIP), sizeof(Header->YourIP));
        UDPPutArray((BYTE*)&(Header->NextServerIP), sizeof(Header->NextServerIP));
        UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr));
        UDPPutArray((BYTE*)&(Header->ClientMAC), sizeof(Header->ClientMAC));

        // Remaining 10 bytes of client hardware address, server host name: Null
            string (not used)
        for (i = 0; i < 64 + 128 + (16 - sizeof(MAC_ADDR)); i++)
```

```
                UDPPut(0x00);
                                    // Boot filename: Null string (not used)
        UDPPut(0x63);                           // Magic Cookie: 0x63538263
        UDPPut(0x82);                           // Magic Cookie: 0x63538263
        UDPPut(0x53);                           // Magic Cookie: 0x63538263
        UDPPut(0x63);                           // Magic Cookie: 0x63538263

        UDPPut(DHCP_MESSAGE_TYPE);      // Message type = DISCOVERY
        UDPPut(1);
        UDPPut(DHCP_DISCOVER_MESSAGE);

        UDPPut(DHCP_SERVER_IDENTIFIER); // Option: Server identifier
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&AppConfig.MyIPAddr, sizeof(IP_ADDR));

        UDPPut(DHCP_ROUTER);    // Option: Router/Gateway address
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&AppConfig.MyIPAddr, sizeof(IP_ADDR));

        UDPPut(DHCP_END_OPTION);        // No more options, mark ending

        // Add zero padding to ensure compatibility with old BOOTP relays that discard
             small packets (<300 UDP octets)
        while (UDPTxCount < 300u)
                UDPPut(0);

        UDPFlush();     // Transmit the packet
}

void ReqToS(BOOTP_HEADER *Header) {
        BYTE i,a;
        DWORD RequestedIP = 0;

        // Set the correct socket to active and ensure that
        // enough space is available to generate the DHCP response
        if (UDPIsPutReady(SSocket) < 300u)
                return;

        // Search through all remaining options and look for the Requested IP address
            field
        // Obtain options
        while(UDPIsGetReady(SSocket))
        {
                BYTE Option, Len;

                if(!UDPGet(&Option))    // Get option type
                        break;
                if(Option == DHCP_END_OPTION)
                        break;

                UDPGet(&Len);   // Get option length

                // Process option
                if((Option == DHCP_PARAM_REQUEST_IP_ADDRESS) && (Len == 4u))
                {
                        // Get the requested IP address and see if it is the one we
                            have on offer. If not, we should send back a NAK
                        // but since there could be some other DHCP server offering
                            this address, we'll just silently ignore this request.
                        UDPGetArray((BYTE*)&RequestedIP, 4);
                        break;
                }

                while(Len--)     // Remove the unprocessed bytes that we don't care
                    about
                        UDPGet(&i);
        }

        UDPPutArray((BYTE*)&(Header->MessageType), sizeof(Header->MessageType));
        UDPPutArray((BYTE*)&(Header->HardwareType), sizeof(Header->HardwareType));
        UDPPutArray((BYTE*)&(Header->HardwareLen), sizeof(Header->HardwareLen));
        UDPPutArray((BYTE*)&(Header->Hops), sizeof(Header->Hops));
        UDPPutArray((BYTE*)&(Header->TransactionID), sizeof(Header->TransactionID));
        UDPPutArray((BYTE*)&(Header->SecondsElapsed), sizeof(Header->SecondsElapsed));
        UDPPutArray((BYTE*)&(Header->BootpFlags), sizeof(Header->BootpFlags));
        UDPPutArray((BYTE*)&(Header->ClientIP), sizeof(Header->ClientIP));
        UDPPutArray((BYTE*)&(Header->YourIP), sizeof(Header->YourIP));
        UDPPutArray((BYTE*)&(Header->NextServerIP), sizeof(Header->NextServerIP));
        UDPPutArray((BYTE*)&(AppConfig.MyIPAddr), sizeof(AppConfig.MyIPAddr));
        UDPPutArray((BYTE*)&(Header->ClientMAC), sizeof(Header->ClientMAC));

        // Remaining 10 bytes of client hardware address, server host name: Null
            string (not used)
        for (i = 0; i < 64 + 128 + (16 - sizeof(MAC_ADDR)); i++)
                UDPPut(0x00);                       // Boot filename: Null string (not
                    used)
        UDPPut(0x63);                               // Magic Cookie: 0x63538263
        UDPPut(0x82);                               // Magic Cookie: 0x63538263
        UDPPut(0x53);                               // Magic Cookie: 0x63538263
        UDPPut(0x63);                               // Magic Cookie: 0x63538263

        // Message type = REQUEST
        UDPPut(DHCP_MESSAGE_TYPE);
        UDPPut(1);
        UDPPut(DHCP_REQUEST_MESSAGE);

        // Option: Server identifier
        UDPPut(DHCP_SERVER_IDENTIFIER);
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&AppConfig.MyIPAddr, sizeof(IP_ADDR));

        // Option: Router/Gateway address
        UDPPut(DHCP_ROUTER);
        UDPPut(sizeof(IP_ADDR));
        UDPPutArray((BYTE*)&AppConfig.MyIPAddr, sizeof(IP_ADDR));

        //Add the client's requested IP, if there's one available
        if(RequestedIP)
        {
                UDPPut(DHCP_PARAM_REQUEST_IP_ADDRESS);
                UDPPut(DHCP_PARAM_REQUEST_IP_ADDRESS_LEN);
                UDPPutArray((BYTE*)&RequestedIP, sizeof(IP_ADDR));
        }

        UDPPut(DHCP_END_OPTION);        // No more options, mark ending
```

```
        // Add zero padding to ensure compatibility with old BOOTP relays that discard
             small packets (<300 UDP octets)
        while (UDPTxCount < 300u)
                UDPPut(0);

        UDPFlush();     // Transmit the packet
}

void DisplayString(BYTE pos, char* text){
  BYTE        l = strlen(text); /*number of actual chars in the string*/
  BYTE      max = 32 - pos;     /*available space on the lcd*/
  char       *d = (char*)&LCDText[pos];
  const char *s = text;
  size_t      n = (l < max) ? l : max;
  if (n != 0)
    while (n-- != 0)*d++ = *s++;/* Copy as many bytes as will fit */
  LCDUpdate();
}

static void InitAppConfig(void)
{
        AppConfig.Flags.bIsDHCPEnabled = TRUE;
        AppConfig.Flags.bInConfigMode = TRUE;

//ML using sdcc (MPLAB has a trick to generate serial numbers)
// first 3 bytes indicate manufacturer; last 3 bytes are serial number
        AppConfig.MyMACAddr.v[0] = 0;
        AppConfig.MyMACAddr.v[1] = 0x04;
        AppConfig.MyMACAddr.v[2] = 0xA3;
        AppConfig.MyMACAddr.v[3] = 0x01;
        AppConfig.MyMACAddr.v[4] = 0x02;
        AppConfig.MyMACAddr.v[5] = 0x03;

//ML if you want to change, see TCPIPConfig.h
        AppConfig.MyIPAddr.Val = MY_DEFAULT_IP_ADDR_BYTE1 |
            MY_DEFAULT_IP_ADDR_BYTE2<<8ul | MY_DEFAULT_IP_ADDR_BYTE3<<16ul |
            MY_DEFAULT_IP_ADDR_BYTE4<<24ul;
        AppConfig.DefaultIPAddr.Val = AppConfig.MyIPAddr.Val;
        AppConfig.MyMask.Val = MY_DEFAULT_MASK_BYTE1 |
            MY_DEFAULT_MASK_BYTE2<<8ul | MY_DEFAULT_MASK_BYTE3<<16ul |
            MY_DEFAULT_MASK_BYTE4<<24ul;
        AppConfig.DefaultMask.Val = AppConfig.MyMask.Val;
        AppConfig.MyGateway.Val = MY_DEFAULT_GATE_BYTE1 |
            MY_DEFAULT_GATE_BYTE2<<8ul | MY_DEFAULT_GATE_BYTE3<<16ul |
            MY_DEFAULT_GATE_BYTE4<<24ul;
        AppConfig.PrimaryDNSServer.Val = MY_DEFAULT_PRIMARY_DNS_BYTE1 |
            MY_DEFAULT_PRIMARY_DNS_BYTE2<<8ul  |
            MY_DEFAULT_PRIMARY_DNS_BYTE3<<16ul  |
            MY_DEFAULT_PRIMARY_DNS_BYTE4<<24ul;
        AppConfig.SecondaryDNSServer.Val = MY_DEFAULT_SECONDARY_DNS_BYTE1 |
            MY_DEFAULT_SECONDARY_DNS_BYTE2<<8ul  |
            MY_DEFAULT_SECONDARY_DNS_BYTE3<<16ul  |
            MY_DEFAULT_SECONDARY_DNS_BYTE4<<24ul;
}
```